iFUSE: A development environment for composable, easy-to-assemble information transforms

Robert A. Joyce*, Jennifer P. Cormier
ATC-NY, 33 Thornwood Drive Suite 500, Ithaca NY, USA 14850

ABSTRACT

A crucial component of a net-centric information management system is a set of simple programs or scripts—fuselets—that effect small transformations on available data. Individual fuselets can perform tasks such as filtering, aggregation, monitoring, format conversion, and simple image manipulation. The global effect of a collection of cooperating fuselets is to add value to the system: to transform data into knowledge. Fuselets are also adept at bridging heterogeneous systems, providing consumers the data they need in the format they require—not necessarily the format that was convenient for the original data producer. ATC-NY has created an extensible fuselet development environment, iFUSE, that provides the support fuselet developers need in order to create and discover fuselets, avoid design and efficiency pitfalls, and ensure the appropriate factorization of fuselet code. For the individual fuselet, iFUSE lets the user focus on the information being transformed, not the code needed to implement the transformation. iFUSE also helps the designer understand the environment in which the fuselet operates, automatically detecting potential data flow problems and providing visualization tools such as "fuselet slicing," which allows fuselet authors and infosphere maintainers to assess the effects of additions and changes in context.

Keywords: transformation, net-centric, fuselet, XSLT, information management

1. INTRODUCTION

A key concept in net-centric information spaces is that of the *fuselet*[1], a light-weight, special-purpose, autonomous client program that provides value-added information processing functions under the control of the information platform. Fuselets interact solely with the information space, and do not read from or write to external files or data sources. Further, fuselets do not have a user interface; they are executed within the information platform and managed by information management (IM) staff. Fuselets are a key part of the Air Force Research Laboratory's Operational Information Management[2] (OIM) research effort and fit cleanly within the broader DoD Net-Centric Enterprise Services[3] (NCES) vision.

Typical information spaces will contain many fuselets, perhaps dozens or hundreds, to transform raw data into knowledge. The fuselets might, for instance, be arranged in workflows[4] that parallel established business process workflows. The "information application" driven by fuselets will typically be designed by domain experts, not programmers. Throughout the lifecycle of the information system, IM staff may need to revise fuselets, remove out-of-date fuselets, or create new fuselets. In doing so, they must be able to develop a global picture of the community of interacting fuselets currently running, as well as those fuselets available for re-use or revision should it be needed.

ATC-NY's research in fuselet development has led to iFUSE, a prototype Integrated Fuselet Synthesis Environment. iFUSE is based on the principle that fuselet designers are primarily interested in information and its transformation, not in the "how" or the code required to make the transformation. In addition, domain experts designing fuselets may not be well-versed in the concurrency problems, deadlock, and other issues that can arise in a community of asynchronous fuselets. iFUSE allows domain experts to develop individual fuselets easily, via a graphical environment, and supports information engineering by visualizing and analyzing a collection of fuselets in context. In infospheres containing many fuselets, iFUSE also provides tools to examine individual fuselets' roles, graphically showing the ultimate producers and consumers of its data ("fuselet slicing").

*rob@atc-nycorp.com; phone 1 607 257-1975; fax 1 607 257-1972; www.atc-nycorp.com

In creating iFUSE, we have gained significant insight into the problems associated with designing an information space, and in what tools can assist non-programmers in understanding and improving the information space. We have seen first-hand a number of ways in which fuselets can be used, both for research and real world scenarios. Together with the Air Force Research Laboratory and other contractors, we have been working to expand the applicability of the fuselet concept to include integration middleware, web services, and other domains.

1.1 Anatomy of a fuselet

Every fuselet, whether created by iFUSE or otherwise, contains two parts: *metadata* and *implementation*. The fuselet's metadata describes the identity, author, and purpose of the fuselet. It also declares what types of information the fuselet acts on, and what types of information it produces. Fuselets are most often used in a typed publish-subscribe information system; fuselet inputs are therefore information objects received via a subscription or query result, and fuselet outputs are information objects published back to the information space.

Fuselets can be implemented in a number of ways, each appropriate to a particular transformation need. iFUSE currently supports XSLT-based components (for quick drag-and-drop construction of output objects from inputs), component fuselets (for assembling a fuselet from re-usable parts), Jython fuselets (for rapid fuselet development by programmers), and Java-based fuselets (for experienced programmers or subtle transformation problems). Help is available within iFUSE to select the proper fuselet type for a particular application.

iFUSE also supports *prospective* fuselets, which have no implementation; such fuselets are placeholders for future implementation, and may be used to describe a fuselet-based process or workflow in general terms. Finally, information space client programs that are not fuselets may also be represented in iFUSE; these representations are known as client proxies, and contain metadata similar to fuselets (to describe their purpose, information use, etc.).

1.2 Fuselet execution

Fuselets can operate on both the *metadata* and the *payload* of an incoming information object. The metadata is an XML document, with a well-defined schema, that describes the object. The payload contains the data of the object; some information objects may have XML payload, while others' payloads may be images or other raw data. iFUSE treats all payloads as either XML (if the information object type library dictates a schema for it), or raw opaque data.

Fuselets are executed and managed in concert with the information platform. FREeME[5], a Fuselet Runtime Execution and Management Environment developed by Lockheed Martin Advanced Technology Laboratories, acts as both a fuselet testing facility and a production execution environment for deployed fuselets.

In this paper, we describe the design goals and methodology behind iFUSE in Section 2, outlining the problem it is intended to solve. In Section 3, we detail some of the iFUSE features that assist in the authoring of both single fuselets and cooperating collections of fuselets. Section 4 describes our approach to simplifying the creation and maintenance of fuselets in a number of languages. A few implementation notes of interest are contained in Section 5. Section 6 concludes by reviewing the lessons learned during the iFUSE effort.

2. DESIGN METHODOLOGY

The overall goal of iFUSE is to let information and transformation design predominate, rather than the details of fuselet code. In grounding this principle, we have the following cross-cutting design goals for the prototype:

- Extensibility: We must support fuselets written in a number of different languages and provide a means by which others can support new kinds of fuselets.
- Uniformity: Users must be able to view all fuselet kinds equivalently. All fuselet kinds (and non-fuselet clients) must participate in data flow analyses and searches.
- Ease of use: Where possible, operations and interfaces must be intuitively understandable and not require a reference manual.
- Re-usability of design artifacts: Users with little programming expertise must be able to assemble novel fuselets from components built by experts.
- Realism of testing: Fuselets must be able to be tested in an environment similar to the eventual production fuselet server.

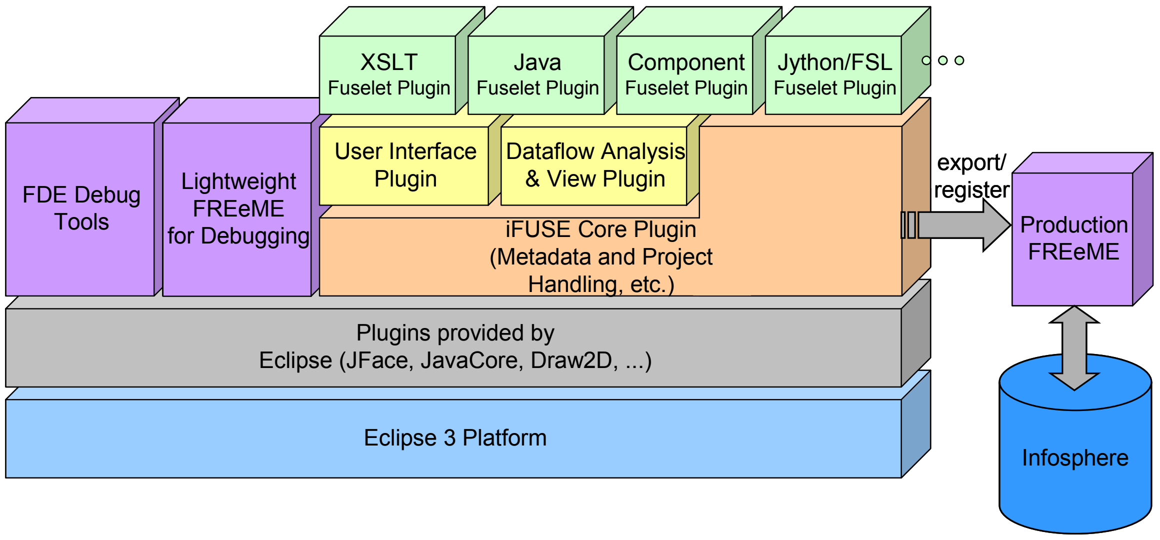

XSLT Fuselet Plugin
Java Fuselet Plugin
Component Fuselet Plugin
Jython/FSL Fuselet Plugin
FDE Debug Tools
Lightweight FREeME for Debugging
User Interface Plugin
Dataflow Analysis & View Plugin
iFUSE Core Plugin (Metadata and Project Handling, etc.)
export/ register
Production FREeME
Plugins provided by Eclipse (JFace, JavaCore, Draw2D, ...)
Eclipse 3 Platform
Infosphere

Figure 1: iFUSE architecture. At the top are plugins supporting various fuselet types. At the bottom are the generic functions provided by the Eclipse platform on which iFUSE is built. On the right, external to iFUSE proper, are the production fuselet server and information platform.

- Interoperability: Generated fuselets must conform to existing OIM standards, such as the Common API[6] (CAPI), and be easily adapted to other information platforms. iFUSE itself must conform to the same standards in communicating with the information platform. Further, iFUSE must generate standards-compliant fuselet metadata, allowing other tools to manipulate iFUSE fuselets.
- Openness: Files and other artifacts generated by iFUSE should be human-readable and editable. Automatically generated code should be visible to users (and modifiable by expert users). The on-disk format of fuselets should be straightforward.

Figure 1 above shows the overall architecture of iFUSE.

2.1 Fuselet metadata standardization

Fuselet metadata allows development, management, and runtime tools to gather information about a fuselet class or fuselet instance without needing to examine the fuselet's implementation (be it in Java, Jython, or another language). In keeping with OIM platform standards and industry conventions, fuselet metadata is in XML format and is described by a W3C XML schema[7].

A fuselet's *class* metadata describes the identity, author, purpose, free parameters, and input/output characteristics of the fuselet. When creating a new fuselet, iFUSE prompts the user for information that will be inserted in the metadata; other metadata fields, such as runtime requirements of the fuselet, are populated automatically. The author can edit the fuselet's class metadata at any time. It is the class metadata that allows iFUSE to analyze collections of interconnected fuselets for the purposes of visualization and data flow problem detection.

A fuselet *instance*—a particular fuselet class set to be executed with a particular set of actual parameter values—is also associated with metadata. The fuselet instance metadata contains the parameter values as well as information on the person or organization that instantiated the fuselet and the particular information platform(s) the fuselet instance will connect with.

The current fuselet metadata standard is the result of three years of revision and feedback from the Air Force Research Laboratory, other fuselet contractors, and outside experts[8].

While not part of the fuselet metadata standard, iFUSE fuselet components, described in Section 4.4, also have metadata. Component class metadata is similar to fuselet class metadata, except that it allows a broader range of input/output options for inter-component communication. A component instance (an instance of a component class, configured in a particular way) uses metadata that is identical in form to a fuselet instance.

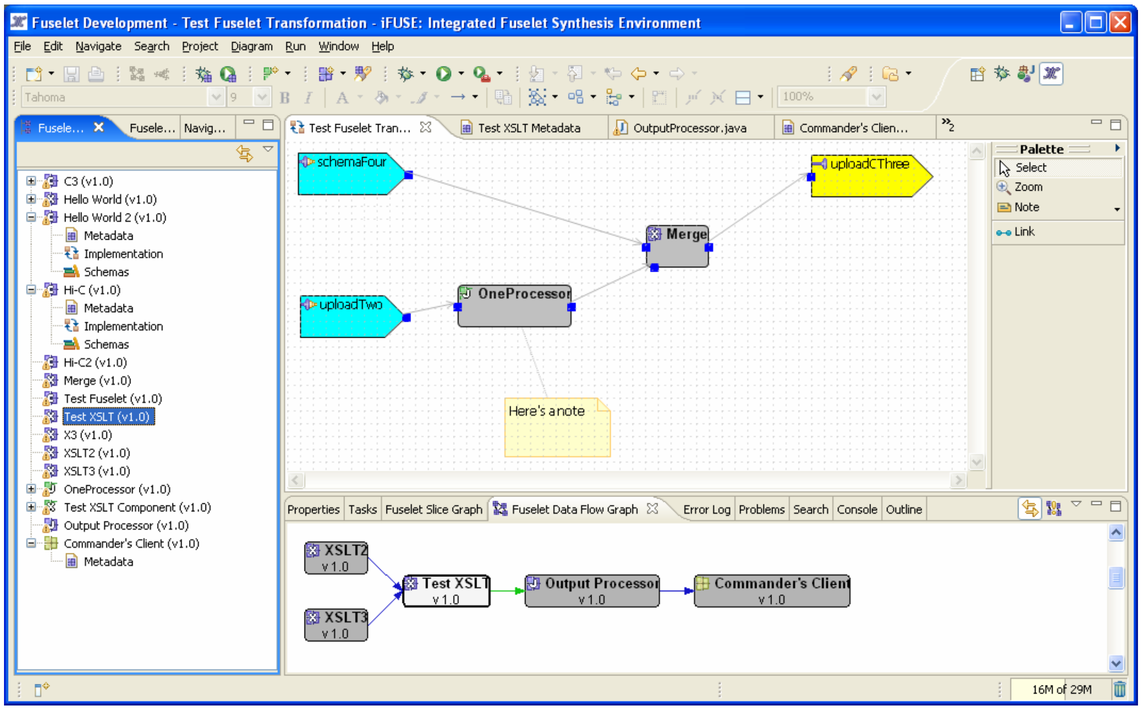


Figure 2: Sample iFUSE screenshot. On the left side of the screen is the fuselet navigator showing all known fuselets, at the bottom is a data flow graph among four fuselets and one client (Section 3.1), and at the top is the graphical component-based fuselet editor (Section 4.4).

3. FUSELET TOOLS

This section describes the salient cross-cutting features of iFUSE. A screenshot of the prototype in action is shown in Figure 2.

3.1 Data flow analysis and visualization

Using the fuselet metadata, iFUSE computes the data flow among all known fuselets. Data flow is defined as one fuselet publishing objects of type X and another fuselet subscribing to objects of type X (or, optionally, a parent type of X). In addition, there is a potential flow of data if one fuselet publishes objects of type Y and marks them for archiving, then another fuselet queries for objects of type Y (or a parent type of Y).

The data flow graph is used for both display and analysis purposes. The global data flow graph is shown at the bottom of the iFUSE window in Figure 2; Figure 3 shows an example with five XSLT fuselets and seven non-fuselet clients. In the application, hovering the mouse over a link shows the information object type, version, and any metadata comment associated with the corresponding data flow between fuselets.

iFUSE continuously analyzes the data flow graph for any potential problems, including duplicates and data flow loops. Duplicate fuselets occur when two fuselets publish, subscribe, and query for the same information object types. Potential duplicates occur when one fuselet publishes, subscribes, or queries for a strict subset of another fuselet's inputs and outputs. (It is impossible to guarantee that the two fuselets are duplicates, but iFUSE warns the user just in case.) A data flow loop—a potentially disastrous resource-wasting situation in a publish-subscribe system—also triggers a warning and is shown in red in data flow graphs.

When there are many dozens of fuselets in the iFUSE workspace, the global data flow graph can become unwieldy. iFUSE offers a fuselet "slice" visualization tool to focus on the pertinent set of fuselets while ignoring others: Analogous to a program slice[9], a fuselet slice is the data flow graph centered on a single fuselet, tracing its input objects recursively back as far as possible through other fuselets and non-fuselet clients. Similarly, the slice traces the fuselet's outputs through other fuselets and clients that consume those outputs, either directly or indirectly. Fuselet slice graphs can be

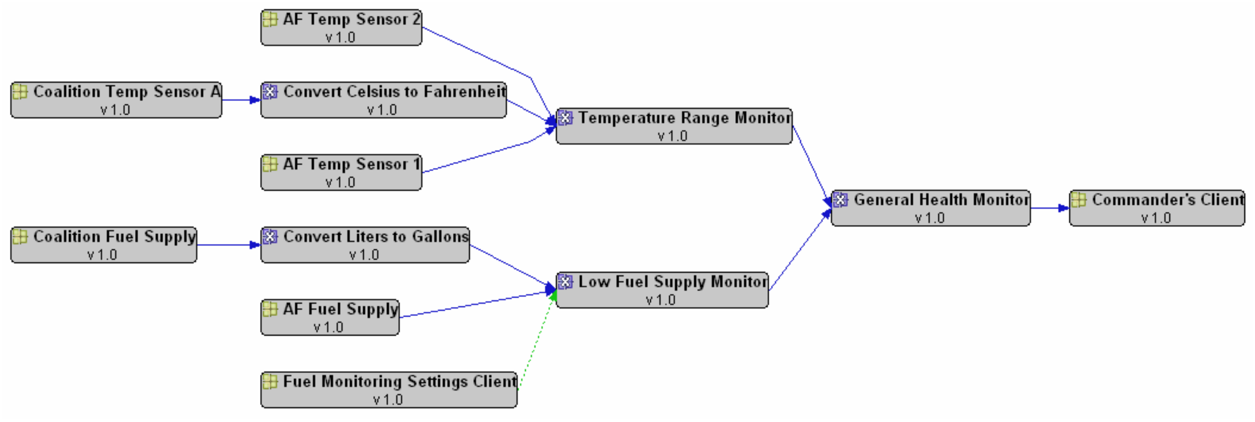


Figure 3: Sample fuselet data flow graph. The type of fuselet is indicated by the icon in the top-left corner of each box. The solid links are due to publish-subscribe "connections" between fuselets, while the dotted link at the bottom is due to objects published by the Fuel Monitoring Settings Client, archived by the information system, then later queried for by the Low Fuel Supply Monitor.

used to assess the effects of design decisions, examine failure modes should an upstream operation fail, or to explore the eventual end-users or clients who will use the data processed by a given fuselet. Such insight is particularly useful in networks of dozens or hundreds of fuselets, where the end (human) consumers of published information may be separated from the fuselet of interest by many intervening transformations.

Whenever the user creates a new fuselet, iFUSE automatically checks for duplicates and loops and prompts the user to continue creation in spite of the potential problems. Users need not remember to check this themselves; it happens automatically. iFUSE assumes users may not be aware of these design subtleties, either because they do not have a programming mindset or because they are not familiar with all the fuselets available in an information space (e.g., if they are maintaining a fuselet collection crafted by someone else).

3.2 Fuselet creation wizards

iFUSE offers a number of wizards for creating new fuselets, guiding users step-by-step through the process of specify fuselet inputs, outputs, required metadata, and other type-specific attributes. Casual users, in fact, need not even know about fuselet metadata per se; they will be asked for all the required data automatically. Before concluding, the wizards check for potential data flow problems and duplicates. When the user clicks "finish", the wizard generates the underlying XSLT, Java, or Jython code needed to implement the fuselet, then opens the appropriate editor on the implementation. Users of XSLT and component-based fuselets never need to touch the underlying code.

3.3 Fuselet metadata editor

While iFUSE creates the fuselet metadata automatically in the fuselet creation wizards, the user may wish to modify the metadata (say, to add more publications or to add an author). iFUSE provides a graphical metadata editor, allowing the user to make the necessary changes. The editor has multiple "tabs," each for a particular facet of fuselet metadata. One of the tabs displays the raw, XML metadata; this can be used to make low-level modifications, or to observe the effect of changes made from other tabs.

3.4 Fuselet search

An important feature for fuselet re-use is fuselet search: it allows the user to find existing fuselets that are similar to one he may be creating, or that may contain important functionality. Fuselet search also lets the user find all fuselets created by a certain author, using a certain information object type, etc. Component-based fuselets can also be searched for by the components they contain. Simple search criteria can be combined to perform complex searches, such as "all fuselets with names containing X, version at least Y, that were not written by author Z." As the search operates only on fuselets' metadata, fuselets can be found even if the particular kind of fuselet (XSLT, Java, etc.) is unknown to iFUSE.

3.5 Refactoring tools

The author of a fuselet, or the maintainer of previously-written fuselets, may want to *refactor* one or more fuselets. For instance, a user may decide that two existing fuselets need to communicate, or that a pair of fuselets should be combined into one to improve performance. iFUSE offers two sets of refactoring tools:

- Generic refactoring tools that can operate on any kind of fuselet (Java, XSLT, etc.). These tools make no assumptions about the core logic of the fuselet, and base all operations on the metadata.
- Type-specific refactoring tools that operate on only one kind of fuselet. Type-specific tools make use of insight into the logic of the fuselet to provide more dramatic transformations. iFUSE provides a number of XSLT-specific refactoring tools, such as Split and Merge. iFUSE also supports the Java-specific refactoring tools provided by Eclipse itself.

3.6 Execution and debugging tools

Executing a fuselet involves setting any free parameters for the fuselet instance, generating instance metadata, possibly uploading the fuselet to an execution engine such as FREeME, and finally running the fuselet. iFUSE performs all these tasks automatically. All the fuselet types supported by iFUSE can be executed, and multiple fuselets can be run at the same time.

iFUSE uses the standard Eclipse "launching" mechanisms to run and debug fuselets, so users already familiar with Eclipse will find iFUSE's launch support very familiar. Typical debugging operations, such as setting breakpoints, examining variables, and stepping through code, are supported.

Fuselets are executed using FREeME if FREeME support is installed in iFUSE (normally the case). By default, a local copy of FREeME embedded in iFUSE is used to run fuselets; this copy is started automatically when the first fuselet is run, and stops when the user exits iFUSE. Fuselet instances can, of course, be run on any FREeME server; iFUSE provides a graphical interface for selecting one. Fuselet classes can be registered with a production FREeME server for later instantiation as well.

If FREeME plugins are not installed in iFUSE, or the user indicates that FREeME is not to be used, iFUSE reverts to launching fuselets directly as Java processes.

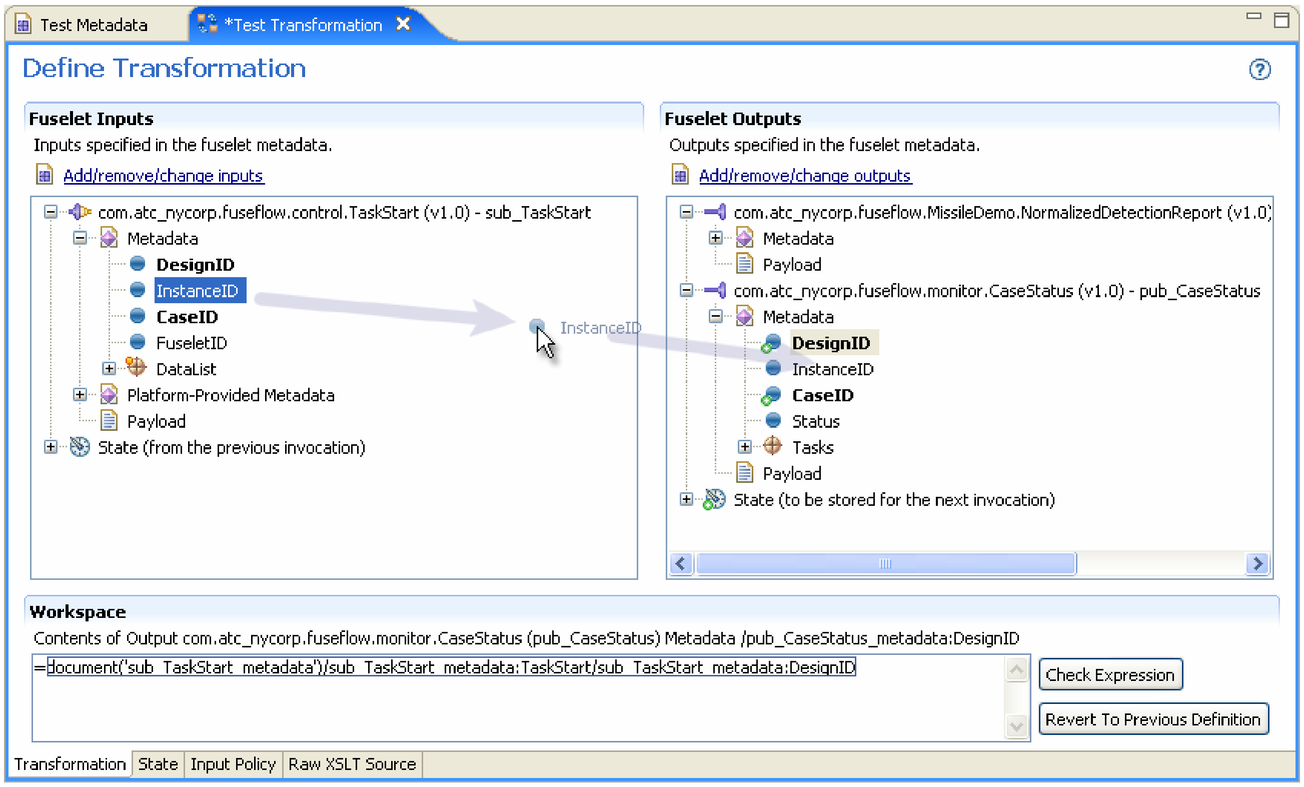
4. KINDS OF FUSELETS SUPPORTED

Different transformation needs—stateful, security-critical, real-time, etc.—are best met with different kinds of fuselets. Further, authors of varying skill levels and capabilities are best served by authoring tools that operate within their area of expertise. iFUSE supports the creation and revision of fuselets written in a number of different languages, including Java, Jython, and XSLT, as well as fuselets created by graphically assembling components.

4.1 XSLT fuselets

XSLT is a language for describing XML to XML transformations, making it a natural way to express fuselet operations[10]. XSLT, however, is very cumbersome to write by hand, even for experienced programmers. iFUSE provides graphical tools to create and manipulate XSLT-based fuselets, as well as additional functionality (such as state and automatic triggering) not available in pure XSLT. The philosophy is not to express every possible intricacy and nuance of XSLT graphically, as do XSLT editors such as Altova's MapForce[11] and Stylus Studio's XML Mapper[12]. Rather, the goal is to simplify the task of domain experts, for whom XML is *not* a native language. Central to XSLT fuselet support is iFUSE's drag-and-drop graphical transformation editor, shown in Figure 4.

The basic principle behind this editor is to fill in output (publication) metadata and payload elements by dragging input (query/subscription) elements to them. The Workspace section at the bottom acts like the Microsoft Excel equation bar: it shows the contents of the currently-selected output element. If the workspace text begins with "=", it is a formula (XPath[13] expression), whose result is put into the output. (iFUSE "smoothes over" some of the unintuitive aspects of XPath, where unambiguous, such as allowing "`/`" where strict XPath requires "`div`" for division. The resulting expressions behave just as a novice user would expect, e.g., "`(a + b) / 2`" for an average.) Dragging elements automatically generates the appropriate XPath expression references. Users can also drag input elements to the Workspace to add an XPath reference to an existing formula. Element references in the workspace are highlighted, and any output elements with malformed formulae are tagged with a red "X" badge and an entry in the Eclipse "Problems" to-do list.


Test Metadata | *Test Transformation

Define Transformation

Fuselet Inputs

Inputs specified in the fuselet metadata.

Add/remove/change inputs

```
com.atc_nycorp.fuseflow.control.TaskStart (v1.0) - sub_TaskStart
  Metadata
    DesignID
    InstanceID
    CaseID
    FuseletID
    DataList
  Platform-Provided Metadata
  Payload
State (from the previous invocation)
```

Fuselet Outputs

Outputs specified in the fuselet metadata.

Add/remove/change outputs

```
com.atc_nycorp.fuseflow.MissileDemo.NormalizedDetectionReport (v1.0)
  Metadata
  Payload
com.atc_nycorp.fuseflow.monitor.CaseStatus (v1.0) - pub_CaseStatus
  Metadata
    DesignID
    InstanceID
    CaseID
    Status
    Tasks
  Payload
State (to be stored for the next invocation)
```

Workspace

Contents of Output com.atc_nycorp.fuseflow.monitor.CaseStatus (pub_CaseStatus) Metadata /pub_CaseStatus_metadata:DesignID

```
=document('sub_TaskStart_metadata')/sub_TaskStart_metadata:TaskStart/sub_TaskStart_metadata:DesignID
```

Figure 4: Graphical XSLT fuselet editor.

Entire trees can be copied by simply dragging them from the input side to the output side; any incompatibilities in the subtrees' XML schemas will be called out for the user. In addition, any publication can be made conditional simply by assigning a Boolean workspace expression to it.

iFUSE automatically and continuously generates XSLT 2.0 code based on the graphical description. The code can be reviewed and hand-modified if desired. Code changes induced by any unsaved graphical operations are automatically highlighted in the XSLT code text editor, making it easier to learn XSLT and audit iFUSE's code generation.

Many fuselet transformations are stateful in nature: they use more than just the most recent values of their input objects. Examples are fuselets that maintain running statistics, those which maintain an internal data model, and those that check for anomalous inputs. iFUSE offers two approaches to adding state between invocations of XSLT fuselets: buffering input data, and maintaining a set of state information from the output of one invocation, providing it to the input of subsequent ones. The two alternatives are shown in Figure 5, below.

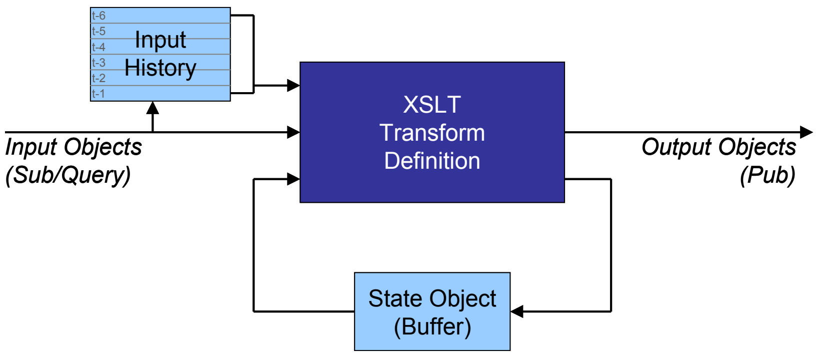


Figure 5: Stateful XSLT fuselet model. The stateless XSLT transform definition, executed with an XSLT engine such as Saxon[14], is surrounded by input and state object buffers.

Of the two methods, the state object method is more general in that it allows arbitrary stateful transformations to be performed, but is less intuitive to non-programmers. The input buffer method restricts authors to the unaltered input history, but is more amenable to understanding by domain experts. Everything that can be done with the input buffer method can also be done with the state object method, albeit in a more complex fashion. (There is nothing specific to XSLT in this design; it applies to state machines in general. The input history buffer has parallels with FIR (fixed impulse response) digital signal processing, while the state object method is similar to a Mealy state machine or IIR (infinite impulse response) digital filter.)

iFUSE XSLT fuselets can use either approach to adding state, or can combine the two:

- Input History: A fixed number of previously-received input objects is stored in a buffer (one buffer per fuselet input). For instance, if a given fuselet is to publish average temperatures over the past five received temperatures, the user could specify to always keep a history of four old elements. Then, on receiving `temp[t]`, the fuselet would also have access to `temp[t-1]`, `temp[t-2]`, `temp[t-3]`, and `temp[t-4]`. These "old" temperature values appear as pseudo-inputs in the left pane of the Transformation tab of the XSLT editor.
- State Object: A special "state" object is added, which can receive the results of custom transforms, as well as serve as the source of transforms for later executions. The first time a transformation is triggered, the state pseudo-input is empty. On each subsequent trigger, the state pseudo-input is identical to the state pseudo-output of the previous transformation. This state object is not published to the infosphere, and is not necessarily of a type known to the metadata repository.

4.2 Java fuselets

Java fuselets offer tremendous flexibility, and are intended for cases where XSLT or component-based fuselets do not offer needed functionality. The drawback is that Java programming experience is required to effectively develop and modify Java fuselets. iFUSE provides automatically-generated template code and a number of tools to simplify the job. The substantial power of Eclipse's Java development environment is also available to the Java fuselet author.

4.3 Jython fuselets

Jython fuselets allow rapid implementation of complex fuselet tasks, but do require knowledge of the Python programming language. Jython extends Python by allowing direct access to Java classes and objects. iFUSE relies on the open-source PyDev[15] package to provide much of the Jython language support. Like for Java fuselets, the fuselet wizard for Jython fuselets automatically generates template code based on the publications, subscriptions, and queries declared by the author. Jython snippets can also be arranged into canned "functions" which allow novice users to construct a fuselet simply by selecting one or more functions within the wizard.

4.4 Component-based fuselets

In addition to the three basic types of fuselets iFUSE supports—Java, Jython, and XSLT—iFUSE allows you to compose fuselets from *components*. Components are high-level functional objects, typically of meaning to an information expert. Fuselets are a prime example of components: existing fuselets can be combined to create a new fuselet. The concept of a component, however, is broader than a fuselet: components can interact more tightly than fuselets as they are not restricted to the CAPI interface and information object publication/subscription/query. For instance, a component can "publish" a raw XML document, binary data, or even a Java object reference. Moving data between components inside a single component-based fuselet is potentially much more efficient than publishing the same data to an infosphere and subscribing to it from another fuselet: there is no network latency and no need to serialize XML documents. The drawback of using a component-based fuselet is that inter-component objects cannot be archived in the infosphere, potentially hindering after-action analysis and forensics. Table 1 summarizes the capabilities of fuselets and components.

Remember, fuselets are a special case of components; *all fuselets are components*. Component-based fuselets, therefore, can act as components in other fuselets! This allows fuselets to be built up hierarchically from base functionality. (Components cannot perform queries for the data produced by other components in the same fuselet: intra-fuselet data is transient and queries only operate on persistent, stored data. A simple "data storage" component could be crafted if there is such a need.)

Table 1. Fuselet versus component capabilities.

Capability	Fuselet	Component
Information Object inputs and outputs (known types)	✓	✓
Information Object inputs and outputs (types not known to the infosphere)		✓
Java object reference inputs and outputs		✓
XML document inputs and outputs		✓
Binary inputs and outputs		✓
Ability to perform queries	✓	
Custom Graphical Configuration		✓

Like fuselets, components are described by class metadata declaring their inputs and outputs. In fact, paralleling the set of iFUSE new fuselet wizards is a set of "new component wizards" which operate similarly but permit the additional component-specific features.

Component instances (within a component-based fuselet) also have associated instance metadata, which is in exactly the same format as fuselet instance metadata. iFUSE provides a graphical component configuration tool, which effectively writes out this instance metadata.

Inside a component-based fuselet, components are *tightly coupled* to one another; this is in contrast to fuselets in an infosphere, which are only "connected" because one subscribes to an information object type that another publishes. In an infosphere, new fuselets can "hook up" to an object stream by subscribing to the correct type. Inside a component-based fuselet, objects from a producer component will only ever be received by the consumer component drawn in the component diagram.

A notional diagram of the component concept is shown in Figure 6, and the iFUSE graphical component-based fuselet editor is shown in Figure 7. The editor was built using Eclipse's Graphical Modeling Framework[16] (GMF) and maintains an internal XML-based model of the transform. This model can be used for later analysis of the transformation or automated refactoring. It is also used at runtime to implement the component-based fuselet. The GMF-based editor also allows non-functional information, such as call-outs and comments, to be added to the transform model, simplifying the job of later fuselet maintainers.

5. IFUSE PROTOTYPE IMPLEMENTATION

iFUSE is a set of "plugins" to the Eclipse[17] platform, an open-source, extensible development environment supporting a number of popular languages and application frameworks. Eclipse's flexibility allows iFUSE to integrate with other development environments or information applications in a single package. One Eclipse installation, for example, could include iFUSE for fuselet development as well as third-party tools for fuselet registration, fuselet execution, and information space monitoring. Further, support for additional kinds of fuselets can be easily added to iFUSE using Eclipse's extension mechanism (documentation of this extension point is available for developers).

Familiarity with Eclipse is not necessary to use iFUSE. However, previous experience with Eclipse can speed a fuselet designer's job tremendously: Eclipse has a number of valuable shortcuts and tools to assist developers, with which the fuselet designer will gradually become familiar.

The following sections describe a few of the noteworthy implementation details of iFUSE.

5.1 Type library

Early on, we required that iFUSE be able to run even if no connection to a running infosphere is available. Such a situation might arise if fuselet development occurs before "standing up" the final infosphere, or if the developer is working remotely. iFUSE therefore keeps its own library of known information object types (metadata schema repository, MSR). Users can import types from a running infosphere to iFUSE, and can export iFUSE's library to an infosphere. Further, users can add, remove, or modify information object types known to iFUSE. With these capabilities, iFUSE becomes a full information space *design* tool.

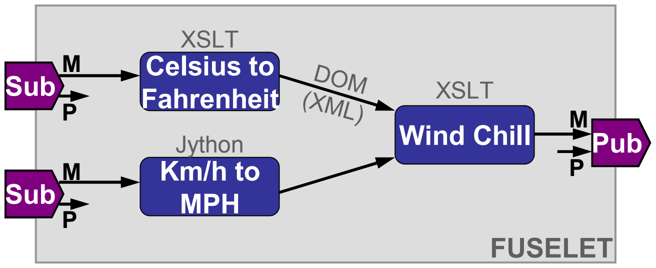
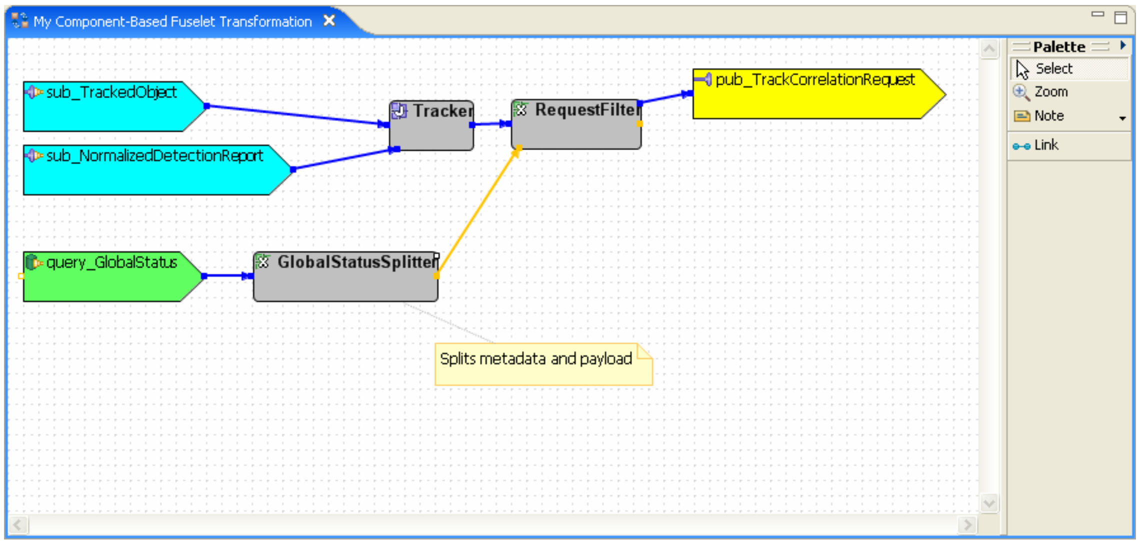


Figure 6: A fuselet assembled from components. In this case, the components communicate with one another via parsed DOM objects representing XML data. This saves time by not requiring a serialization/de-serialization round trip for XML text between components.



My Component-Based Fuselet Transformation
sub_TrackedObject
sub_NormalizedDetectionReport
Tracker
RequestFilter
pub_TrackCorrelationRequest
query_GlobalStatus
GlobalStatusSplitter
Splits metadata and payload
Palette
Select
Zoom
Note
Link

Figure 7: iFUSE component-based fuselet editor, built using Eclipse's Graphical Modeling Framework.

5.2 Fuselet project format

In order to support multiple types of fuselets, and to provide separation for fuselets that contain multiple files, each fuselet is an Eclipse project in iFUSE. The projects are semi-standardized, in that they must have a `metadata.xml` file at the top level and have any source files in a `src/` directory. Any "compiled" files generated by the auto-build process must be put in a `bin/` directory. Beyond those requirements, the format of a fuselet project is type-dependent.

5.3 Fuselet type support via plugins

iFUSE supports multiple types of fuselets by building on Eclipse's extension and plugin mechanisms. We have defined and documented an API for extending iFUSE to support a new fuselet type. This API was tested many times over, in developing plugins to support XSLT, Java, Jython, and Component-based fuselets. (Jython fuselet support was developed by Lockheed Martin using the API documentation.)

There is also a provision to support "un-typed" fuselets as placeholders. A fuselet of a type not known to a particular installation of iFUSE behaves just as if it were un-typed: it participates in data flow analyses and its metadata can be edited, but the transformation itself cannot be altered.

5.4 Client proxies

Often, is it useful to include non-fuselet clients in data flow graphs, searches, and other fuselet tools. A client could, for example, access an external database or provide a user interface, but otherwise behave like a fuselet. iFUSE represents such clients by fuselet class metadata, with one exception: clients are allowed to not publish any outputs (e.g., if they interface to external systems or a user; a fuselet without publications would be useless). In fact, client proxy support is provided by a nearly-trivial fuselet type plugin, just as described in the previous section.

5.5 Data flow analysis algorithms

As described in Section 3.1, iFUSE continually computes the data flow graph among all known fuselets, based on the fuselets' metadata. Matching of fuselets' publications and queries or subscriptions is done solely based on information object type and version. Some information spaces have information object types arranged hierarchically, with parent and child relationships between types. If the user desires, matching can use type inheritance, matching a publication of type `X` to any subscription of a parent type of `X`. (Use of the type hierarchy is a configuration option in the Air Force Research Lab-oratory's OIM Reference Implementation platform.)

In searching for data flow loops, certain pathological cases can yield an exponential number of loops (exponential in the number of fuselets involved). In order to keep the user interface responsive, iFUSE halts the search after 1000 data flow loops are found in a particular set of fuselets.

Matching of publications and subscriptions or queries currently ignores any XPath predicates on the input/output metadata. This is done for three reasons. First, CAPI 1.5 does not support publication predicates, so predicate-based matching is meaningless for any CAPI-compliant platform. (With no predicate on the output, iFUSE could only assume that at least one output object would match a given query or subscription predicate.) Second, true XPath predicate matching is an exponentially difficult problem; deciding whether two XPath expressions overlap (i.e., they intersect) is an even more difficult problem than Boolean satisfiability[18]. Finally, even if iFUSE could efficiently compute the intersection of two XPath predicates, there is no guarantee that any real, *published* object will actually meet both predicates; iFUSE has no way to determine the scope of object metadata that is actually published by a fuselet. For these reasons, all data flow "links" determined by iFUSE are *potential* links, and even if iFUSE could compute actual predicate intersections, they could still be no better than potential links.

6. CONCLUSIONS AND LESSONS LEARNED

iFUSE provides the support that fuselet developers need in order to discover existing fuselets and rapidly create new ones, avoid design and efficiency pitfalls, and ensure the correctness and maintainability of the information system as a whole. In doing so, iFUSE allows information management staff to rapidly respond to new, unanticipated information needs in a timely manner without the need to bring in an engineering team. iFUSE also enables the rapid integration of heterogeneous systems, allowing non-programmers to make use of components crafted by experts. In addition, iFUSE helps the designer of an individual fuselet understand the environment in which the fuselet operates, and thus address issues of efficiency and the global effect of the fuselet community. Without such a tool, net-centric information spaces and integration middleware would eventually suffer from performance and maintainability problems that would be hard to detect and correct.

Across all fuselet types, certain "idioms" become apparent. These common aspects of each fuselet deserve more formal support in order to further simplify the task of fuselet creation. Examples of common, idiomatic fuselet features are (1), query-then-subscribe, for fuselets that must fetch the current state of the world and receive automatic updates; (2), data-dependent queries, where the query predicate depends on some aspect of the incoming data; and (3), reference data, which are relatively static objects repeatedly consulted by fuselets. While all of these idioms can be implemented in each fuselet type, users would benefit from canonical, generalized implementations, such as simplified data-dependent query support in our XSLT editor, and automatic, behind-the-scenes lookup and caching of reference objects.

We have also shown that the analysis of fuselet data flow on-the-fly is feasible, even testing many dozen fuselets together. The fact that iFUSE stops searching after 1000 data flow loops turns out not to be a problem. In such cases, the user really only needs to know of the existence of (large numbers of) data flow loops involving their new fuselet; enumerating all possible loop combinations is not necessary. Data flow computations using information object types (and ignoring predicates) are generally sufficient, but depend on the information space using descriptive types (e.g., types that have real world meaning). An information space with only one type, `baseObject`, may function, but permits little analysis, monitoring, or long-term maintenance.

More generally, during the iFUSE development and testing effort, we observed that information space design, especially by non-programmers, is a very challenging task. While iFUSE allows domain experts, who are not programmers, to create fuselets, such non-programmers need to understand the information space as a whole. Simplifying the development tool is not sufficient; users need to understand the fundamental information space operations of publication, query, and subscription, and the semantic consequences of each. The design of an efficient information space involves many domain-specific tradeoffs (efficiency, simplicity, storage requirements, bandwidth requirements, etc.), and is intimately dependent on the information platform (OIM, web services, etc.) in use. Internal testing of iFUSE at ATC-NY, by staff who are technically-savvy but are not programmers, highlighted this issue: while they could understand the operation of iFUSE per se, they were not cognizant of the larger information space design implications of many of their actions within iFUSE.

Overall, we believe that fuselets hold a promising future, both as a research vehicle and as a framework for deployable information applications. The decomposition of information needs into small, managed transformations speeds development, enables automated analysis and visualization, and simplifies long-term maintenance of the information space.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the input of Mr. James Milligan and Mr. James Nagy of the Air Force Research Laboratory, Rome NY, in the fuselet metadata standardization process and in guiding fuselet research more broadly. We also acknowledge the significant contributions of Lockheed Martin's Advanced Technology Laboratories ISX Lab in providing Jython fuselet support as well as the fuselet runtime execution and management environment (FREeME) for use in iFUSE.

Support for the research described herein was provided by the Air Force Research Laboratory under contract FA8750-04-C-0070.

REFERENCES

1. Operational Information Management (OIM) fuselet technology web site, http://www.fuselet.org/
2. Air Force Research Laboratory Information Directorate (AFRL/IF) Operational Information Management program (formerly the Joint Battlespace Infosphere), http://www.rl.af.mil/programs/jbi/
3. Defense Information Systems Agency (DISA) Net-Centric Enterprise Systems (NCES), http://www.disa.mil/nces/
4. M. Stillerman, "Final Report – FuseFlow: A Workflow-Aware Fuselet Management Environment," ATC-NY Tech Report TR06-004, March 23 2006.
5. Lockheed Martin Advanced Technology Laboratories ISX Lab's Fuselet Runtime Execution and Management Environment (FREeME), http://jbi.isx.com/freeme/
6. OIM Common API (CAPI) version 1.5, http://www.infospherics.org/api/
7. World Wide Web Consortium (W3C), "XML Schema Part 0: Primer," May 2001, http://www.w3.org/TR/xmlschema-0/
8. Fuselet Focus Group materials, October 2004, http://www.fuselet.org/briefings/minnowbrook-2004/
9. S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs," *ACM Transactions on Programming Languages and Systems*, 1990. 12(1): p. 26-60.
10. World Wide Web Consortium (W3C), "XSL Transformations (XSLT) Version 2.0," January 2007, http://www.w3.org/TR/xslt20/
11. Altova MapForce, http://www.altova.com/products/mapforce/xml_to_xml_mapping.html
12. Stylus Studio XML Mapper, http://www.stylusstudio.com/xml_to_xml_mapper.html
13. World Wide Web Consortium (W3C), "XML Path Language (XPath) Version 2.0," November 1999, http://www.w3.org/TR/xpath20/
14. Saxonica Limited, Saxon XSLT and XQuery processor, http://www.saxonica.com/
15. PyDev Python Development Plugins for Eclipse, http://pydev.sourceforge.net/
16. Eclipse Graphical Modeling Framework, version 1.0, http://www.eclipse.org/gmf/
17. Eclipse tool platform, version 3.2. http://www.eclipse.org/
18. L. Zhang and S. Malik, "The Quest for Efficient Boolean Satisfiability Solvers," *Proc. 8th Intl. Conf. on Computer Aided Deduction (CADE 2002)*, July 2002.