# Tools to Support Secure Enterprise Computing

*Myong H. Kang, Brian J. Eppinger, and Judith N. Froscher*
Information Technology Division
Naval Research Laboratory

**Abstract**

*Secure enterprise programming is a difficult and tedious task. Programmers need tools that support different levels of abstraction and that track all the components that participate in distributed enterprises. Those components must cooperate in a distributed environment to achieve higher-level goals. A special case of secure enterprise computing is multilevel secure (MLS) computing. Components that may reside in different security domains have to cooperate to achieve higher-level missions.*

*To ease the programmer's burden, we are developing an MLS workflow management system (WFMS), called MLS METEOR. A programmer can specify a distributed programming logic through a GUI-based workflow design tool. Based on the programming logic, MLS METEOR will generate a distributed runtime system that handles communication among different hosts, even those that reside in different classification domains. The multilevel security enforcement of MLS METEOR does not depend on the WFMS itself but rather on the underlying MLS infrastructure and a few security critical components. This paper concentrates on the system organization of MLS METEOR and the rationale for this structure. We explain which portions of the system can be used in generic enterprise computing and which portions are specific to MLS computing.*

## *1. Introduction*

Globalization has replaced the separation that characterized the Cold War era. Unconventional coalitions among businesses and nations and among former adversaries are formed to advance common goals, then quickly dissolve as individual objectives change. Threats now lie in these essential connections among participating enterprises, which also enable profitable cooperation. To facilitate these alliances, businesses and the military rely on distributed information technology (IT) for most operations and must be able to respond quickly to new situations and threats in completely different environments. Hence, supporting IT resources must be flexible to allow for rapid reconfiguration.

The military has additional requirements that stem from the need to pull together coalitions in a short timeframe to achieve a common goal and to protect sensitive national security information. Each mission has different mission logic and deals with different computing resources that can belong to different classification domains. Therefore, distributed programs that support such missions have to deal with multilevel security (MLS) issues.

Another complication of distributed computing arises because the programs are very large. In general, distributed programs are much larger than conventional programs and often involve the integration of existing applications to achieve higher-level goals. Even though distributed object computing standards like CORBA and DCOM, have made a basic level of interoperability among distributed applications possible and have made distributed programming tenable, distributed programming is still a difficult and tedious task. Usually a team of programmers has to work on different parts of a program, which have to be assembled to provide the IT support for the mission. It is often difficult to have a global picture of the whole program and to monitor the progress of the work due to the magnitude of these programs and the wide distribution of resources.

The operational environment and dependence on cooperation among distributed IT resources mean that we need development and runtime tools that

- ease the programming burden of constructing large-scale, distributed systems and promotes reuse of existing components,
- provide a GUI-based distributed programming environment that offers different levels of abstraction so that not only the global picture of the program, but also more detailed views of a component, can be displayed for different users,

- allow easy (re)configuration of design to accommodate and promote integration with coalition partners,
- generate runtime code to handle the complexities of distributed communication (e.g., CORBA, DCOM, HTTP),
- can specify recovery strategies,
- reduce the design time and cost of MLS applications,
- generate secure runtime code to ensure the success of the mission since these systems operate in many different classification domains, and
- provide monitoring capabilities so that users at different classification domain can determine the status of work in progress at their level and the levels that they are allowed to monitor.

Even though there may be many ways to achieve these goals, we have started with the workflow paradigm, to which we can add new capabilities such as multilevel security, distributed scheduling, recovery, etc. In this paper, we view multilevel secure computing as a special case of secure enterprise computing. We may have to guard the connections among different security domains more strongly than the connections among business partners. However, the MLS programming principles is not much different from any other secure enterprise computing. In fact, we have designed the WFMS so that different security infrastructures can be used to facilitate cooperation among several enterprises.

This paper is organized as follows. In section 2, we briefly review our strategy to achieve the above goal for secure enterprise computing. We present the software structure for implementing such a system in section 3. We carefully organize the software so that only a small portion is specific to MLS computing. Section 4 concludes this paper and presents the status of the project and future work.

## *2. A Strategy for Secure Enterprise Computing*

We presented a strategy to pursue the above goals in a separate paper [2]. In this section, we summarize the strategy for the sake of completeness.

The MLS workflow management system (WFMS) that we are building will provide equivalent functionality to a single-level WFMS and hooks into an MLS infrastructure for enforcing the MLS security policy. Tasks that may be single-level individually, but located in different classification domains, have to cooperate to achieve a higher-level MLS mission. Therefore, we need to provide an MLS distributed programming (design) tool that allows programmers to specify their distributed program logic (we sometimes call *mission logic* in this paper). This design tool allows MLS workflow designers to

- divide a design area into multiple domains,
- specify information flow, dependency, and the condition of the dependency among tasks that are in the same or different domains,
- specify dominance relationships among domains (e.g., Top Secret > Secret > Unclassified), and
- specify exception conditions and recovery strategies for exceptions.

On the other hand, the runtime engine needs to provide MLS services in a distributed and heterogeneous computing environment. The MLS runtime system must enforce the following information flow requirements:

- High users may have access to low data and low resources,
- High processes may have access to low data, and
- High data must not leak to low systems or users.

An MLS WFMS should obey this MLS policy. Atluri *et. al.* have investigated MLS workflow in general [1]. The development of high-assurance software, necessary to provide separation between unclassified and TS/SCI information, such as MLS workflow systems, has proven to be both technically challenging and expensive. Today's fast paced advances in technology and the need to use COTS products make the traditional MLS approach untenable. Therefore, we have chosen the approach for building MLS workflow by integrating multiple single-level workflows with an MLS distributed architecture. This is in line with modern distributed computing paradigms that support autonomy and heterogeneity.

To implement an MLS WFMS using the architectural method, the following technical approach has been established:

- Implement the necessary design tool for supporting MLS workflow. Even though this tool allows workflow designers to specify information and control flow among tasks in different domains, the operational environment of the tool will be system-high (i.e., the workflow design tool neither accesses sensitive data in multiple domains nor passes it around). Hence, we can implement this tool without too much concern for multilevel security issues (e.g., information leakage across classification domain boundary). This tool will be run on a single-level system.
- Choose a strategy for dividing an MLS workflow that was designed using the design tool into multiple single-level workflows.

- Choose an MLS distributed architecture where multiple single-level workflows can be executed.
- Choose a single-level WFMS to execute single-level workflow in each classification domain.
- Extend the workflow interoperability model to accommodate cooperation among workflows at different classification domains.
- Extend the single-level workflow enactment service (i.e., runtime engine) to accommodate communication among tasks in different classification domains.

In the following section, we describe the internal structure of the MLS WFMS that we are building using the above strategy.

## *3. System Organization*

There are many ways to satisfy the requirements that were described in section 1. We believe that an MLS WFMS is a good way to meet those requirements. In general, a WFMS consists of two main components: a design tool and runtime tools. Our requirements contain an unusual requirement, which is MLS. However, the rest of requirements are generic enough for use by corporate environments. We believe the MLS requirement and the way we solve this MLS problem actually helps us to look into workflow interoperability from a fresh perspective [2].

We have developed generic platform independent distributed programming tools based on an object-oriented paradigm; hence, Java was chosen as our development language. We also want our design tool to be not only independent of the runtime engine, but also independent of the underling MLS infrastructure. Based on these requirements, we have developed the system organization as shown in figure 1.

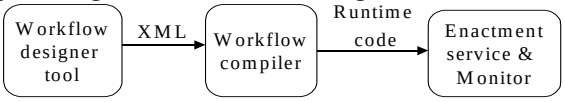


Figure 1: Internal structure of MLS METEOR

A workflow designer specifies mission logic through the workflow design tool (i.e., various workflow editors). The workflow design tool saves the design in XML (eXtensible Markup Language) [5]. When a workflow design is completed, the workflow compiler reads the XML representation of the design and performs the necessary design analysis and validation. Finally it generates runtime code for enactment services. Even though there are many workflow runtime engines, there are very few distributed runtime engines. We believe that OrbWork, an implementation of METEOR WFMS from University of Georgia and a distributed workflow runtime engine, is a good starting point. To satisfy MLS and other requirements, OrbWork has been extended. In the following subsections, we will explain each component in detail.

### 3.1. Workflow Design Tool

The workflow design tool is a generic distributed programming tool that can express programming logic through GUI-based editors. We have developed the workflow design tool based on the MLS distributed computing model.

#### 3.1.1. MLS distributed computing model

MLS distributed programming adds another dimension of complexity to single-level distributed programming, which itself is not a trivial task. Therefore, we need a new programming model for MLS distributed computing that

- eases the burden of MLS distributed programming, especially in the context of large system integration,
- promotes the re-use of existing components,
- facilitates the specification of security requirements (e.g., roles),
- enables secure cooperation among autonomous systems at different classification levels, and
- provides a global picture of the whole mission and a proper view of a mission to users at different levels of abstraction,

In the MLS METEOR model, a *task* represents an abstraction of an activity. A task can be regarded as a unit of work that is performed by a variety of processing entities, depending on the nature of the task. A task can be performed by (*realized by*) a human, a computerized activity that executes a computer program, a database transaction, or possibly a network of interconnected tasks. Hence, a task provides one level of abstraction (view) and its realization provides a lower level of abstraction (view). Since the realization of a task may contain many tasks at different levels of abstraction, a task is a recursive reference in the METEOR model. In other words, one task from a particular user's point of view may be a network of many tasks from another user's perspective.

There are two types of tasks in the model:

- *Foreign task:* A task whose realization (i.e., strategy for implementation) is unknown to the workflow designer. It represents a task that is a part of cooperating independent system. It is required for a designer to declare a foreign task explicitly to provide a hint to the runtime code generator. A foreign task should have a minimal information set

(e.g., where to send the request, how to receive output).

– *Native task:* A task for which the realization is known or the realization will be provided before runtime code generation (i.e., all other tasks except the foreign tasks).

For example, foreign tasks can be used to define communication and synchronization with a task in other classification domains. If an MLS workflow is created at the highest classification domain, then the complete MLS workflow with realizations of all its tasks can be specified. However, if the workflow designer creates an MLS workflow that requires input from (or output to) higher classification domains, then he may know only the interfaces to the tasks at the higher levels but not the detailed workflow process at higher levels.

A native task can be either a simple task or a network task. A simple task is a task that cannot be broken down further from a workflow designer's point of view. A *network task* represents the core of the workflow activity specification. Since a network task is one of the realizations of a task, it is always associated with a task called its *parent task*. A single network of tasks defines a relationship among workflow tasks, transferred data, exception handling, and other relevant information. It is a collection of either foreign or native tasks and transitions from one task to another. Figure 2 shows a simplified version of two levels of abstractions (views) where Task2 is the parent task of the projected workflow $W_1$, which contains tasks 4, 5, 6, and 7, and transition $t_j$ represents a transition from Task1 to Task2. In Figure 2, Task1, Task2, and Task3 may belong to different classification domains. Hence, the MLS METEOR model can be thought of as follows: along the *xy*-surface, there are tasks in different domains and along the *z*-axis, there are different levels of abstraction.

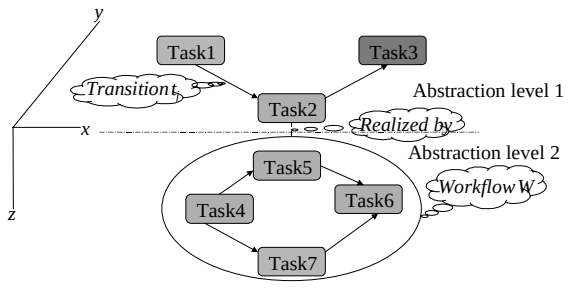


Figure 2: MLS METEOR Model

A task may play the role of a source task or a destination task (e.g., Task1 is the source task and Task2 is the destination task of the transition $t_j$ in Figure 2) for a number of transitions. All of the transitions for which a task is the destination task are called the *input transitions* for that task (e.g., transition $t_j$ is an input transition for Task2). Likewise, all the transitions for which a task is the source task are called its *output transitions* (e.g., transition $t_j$ is an output transition of Task1). A transition may have an associated Boolean condition called its *guard*. A transition may be activated only if its guard is true. When there is a transition from task $T_i$ to task $T_j$ where $T_i$ and $T_j$ are in different classification domains, we call this an MLS *transition* from $T_i$ to $T_j$.

An *external transition* is a special type of a transition in which the two participating tasks (source and destination) are not in the same workflow (i.e., transition to and from a foreign task). An external transition may lead to a start task of another workflow. Similarly, an implied transition leads from the final task and is used to notify the external entity that the network has terminated. Note that an MLS transition is turned into an external transition when an MLS design is divided into multiple single-level workflows for runtime.

External transitions are also used to specify synchronization points with some external events. Typically, external transitions may be used to specify communication and synchronization between two independent workflows. Here, an external transition leading into a task in the workflow is assumed to have an implied source task (outside the workflow). Similarly, an external transition leading out of a task in the workflow is considered to have an implied destination task (outside the workflow). External transition is a cornerstone of our strategy to support MLS workflow.

The classes (i.e., types of objects) that are associated with an input transition to a task are called the task's *input classes*, and those appearing on an output transition are called *output classes* of that task. If an output class is also not an input class then the class is *created* by the task. Specifically, an object instance of the specified class is created by the workflow runtime. An input class that is not an output class is *dropped (consumed)*. When input classes are unused by the task, they are transferred to the task's successor(s).

A group of input transitions is called an *AND-join* if all of the participating transitions must be activated for the task to be *enabled* for execution. An AND-join is called *enabled* if all of its transitions have been activated. All the input transitions of a task may be partitioned into a number of AND-joins. A group of input transitions is called an *OR-join* if the activation of one of the participating transitions enables the task.

A group of transitions is said to have a *common source* if they have the same source task and all lead either from:

- Its success state or
- Its fail state.

A group of common source transitions may form one of the following:

- *AND-split*: Each of the transitions in the group has the condition set to `true`. This means that all of the transitions in the group are activated once the task is completed.
- *OR-split* (selection): An ordered list of transitions where all but the last transition may have arbitrary conditions (i.e., the last transition on the list has the condition set to `true`). The first transition whose condition is satisfied will be activated.
- *Loop*: A special case of an OR-split, where the list is composed of exactly two transitions: continue and break. Continue implies branch taken and break implies branch not taken (i.e., fall through).

All tasks that we define in this paper are single-level tasks. What we mean by single-level is that the task receives input from one classification domain and produces output at the same classification domain. There are four special tasks: *begin*, *success*, *failure*, and *synchronization*. The synchronization tasks represent external transitions to and from other workflows. In general, workflow designers do not manipulate synchronization nodes directly. They are automatically generated by the system based on the specification of foreign tasks and input and output transitions to and from the foreign tasks.

An MLS *workflow* is a network of interconnected single-level (foreign or native) tasks from more than one classification domain. Note that we call a task single-level from one particular level of abstraction (view). Since a single-level task may be realized by an MLS workflow at a lower level of abstraction, it may have side-effects on different classification domains at lower abstraction levels. Hence, our distinction between single-level and multilevel is purely from the perspective of a specific abstraction level.

Let $CL(T_i)$ represent the classification domain of task $T_i$. The relationships between the classification domains form a lattice. An MLS workflow that is the realization of task $T_i$ where $CL(T_i) = S_a$ must obey the following constraints:

- The *begin*, *success*, and *fail* nodes of the MLS workflow must be CL(*begin*) = CL(*success*) = CL(*failure*) = $S_a$.
- It may have tasks in other classification domains; however, if the $CL(T_j) = S_b$ where $S_a$ does not dominate $S_b$, then $T_j$ must be a foreign task. In other words, only tasks in $S_c$ where $S_a \geq S_c$ may have realizations.

### 3.1.2. Design Editors

The workflow design tool should provide an easy way to capture the control and data flow among components. It should also provide an easy way to import existing designs or components to the current design environment. We provide various platform independent GUI-based editors to support the MLS distributed computing model. There are two starting points into a specific design process (see figure 3). They are the task and network editors, both of which will create an initial top level component (task). The arc and operator editors are mainly used in conjunction with the network and task editors to specify data and control flow. There are three additional editors that aid MLS workflow design: data, domain, and role editors. These three editors can be used independent of the specific workflow. For example, domain structure, especially in MLS, may be predefined based on physical separation. Also role hierarchy [6] may be predefined by organizations. This semi-independence enables a workflow designer either

- to use predefined data, and domain and role structures from a previous design or
- to define necessary data, domain and role structures during a workflow design.

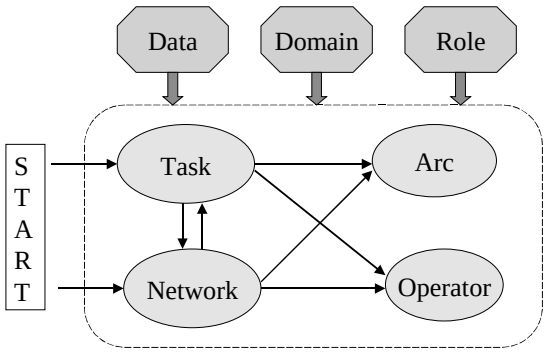


Figure 3: A typical design process and relationship among editors

The description of each editor follows.

***Network Editor***

A network editor, alternatively called a map editor, is a graphical programming tool that allows users to lay out the control flow of the intended mission logic. Hence, at the highest abstraction level, it provides a global picture of a mission. In this editor, the designer can

- divide the drawing area into many classification domains,
- drop tasks in the different domains, and

– draw arcs that represent control flow between tasks.

A designer can traverse the different abstraction levels to observe or specify different workflow logic with this editor. It also provides links to all other editors to refine a design. For example, if a designer wishes to specify operators (e.g., AND-split, loop) for a specific task, then he can do so by accessing the operator editor.

An ability to prescribe recovery routes and alternative tasks in case of failure is an important feature for an MLS WFMS. Our designer provides this capability through various editors. First, the network editor supports two types of arcs that represent transitions: one is success arc and the other is fail arc. Second, METEOR also supports system and user-defined exceptions that can be specified through the task editor. Using exceptions and fail arc, a workflow designer can specify a recovery strategy for predictable failures. Figure 4 shows a snapshot of the network editor.

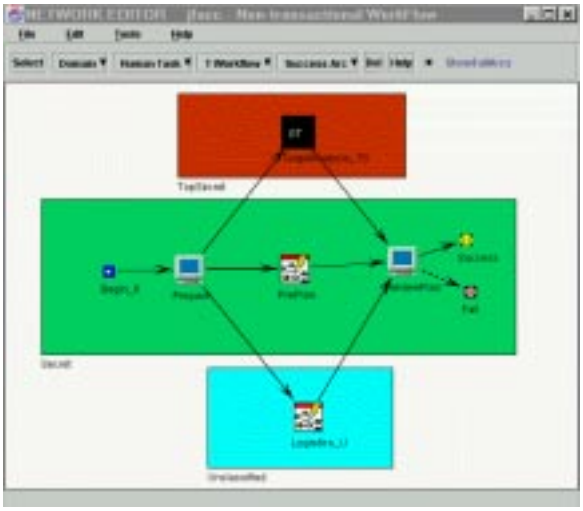

Figure 4: A snapshot of the network editor

***Task Editor***

The task editor provides the designer with a tool for describing the interface, operating environment, and implementation of the task. Since tasks can be connected together in the implementation of another task (i.e., network task), the task editor also provides information about connections to other tasks and their corresponding editors. In other words, the task editor provides a task-centric view of the workflow.

To describe the task interface, the designer provides a unique name for identifying the task, along with a textual description. The task's type specifies how the task is implemented. The two primary task types are network and simple, where simple tasks include: human, transactional and non-transactional. To complete the interface to a task the designer must specify the input data objects necessary to invoke the task along with the task's output data objects. The designer can specify multiple invocations, but during runtime all the data objects for one of the invocations must be available for a task to start. If the designer wishes to specify failure states for a task then special data objects, called exceptions, are used.

The designer must also describe the environment within which the task has been designed to operate. Of critical importance to an MLS design is what classification domain the task will be operating in. The designer can use compartments (e.g. data restrictions) to further restrict access to the task and its data. In addition the designer can specify what organization owns the task and what roles (which will map to a list of users at runtime) are allowed to perform the task. The designer can also specify the host where a task should be located and specify any system and operational constraints (e.g., allocated time to completion) for the task. During workflow design it is important that the designer considers constraints which have been specified in parent tasks.

The designer must also specify the task implementation (realization). The task's realization is highly dependant upon the type of the task. For a network task, the designer can use the network editor to describe the underlying workflow. For a simple human task, the workflow design tools will generate a generic html page based upon the inputs and outputs of the task. The designer can then specify an html editor and viewer that he can use to customize the HTML page for the desired result. When implementing a simple transactional task, the designer will be able to enter the database query commands that are necessary to carry out the commands. For a simple non-transactional task, the designer can enter the code necessary for invoking the task (e.g., executable code, CORBA invocation) or can enter a description of what needs to be done, and the runtime designer can actually implement the functionality. Both transactional and non-transactional tasks can be connected to existing legacy applications.

There are two other special tasks, which the task editor can edit. They are the abstract and the foreign tasks. A designer uses an abstract task to describe the interface and security of a task that some other designer will complete before runtime code generation. But a foreign task is used to describe the interface to a task that will be implemented by another designer and will be available only at runtime.

To achieve a task-centric view of the workflow, the task editor provides the necessary connectivity to look at the entire design. If the task is used within a workflow, the task editor provides the designer with a view of all the task's connections (arcs) whether input, output, or failure arcs. For each connection, the editor provides the designer with quick access to the associated arc, operator, and task editors. And the task editor provides the ability to view down to the implementation details.

***Arc Editor***

An arc in the network editor represents a transition from a source task to a destination task. In our implementation, arcs specify the data transferring from the source task to the destination task (i.e., input and output classes). The arc editor provides an easy way to map outputs from one task to inputs of another task in a given workflow. For example, one task has three outputs, type1, type1, and type2. Another task has three inputs type1, type1, and type2. Since there is an ambiguity of matching two type1 outputs to two type1 inputs, an arc editor provides a handy way for the designer to specify which output of a task corresponds to the input of another task.

***Operator Editor***

The new model uses operators to specify the input and output transitions for a task. Hence, a designer needs a capability to edit the structure of these operators. Due to the complexity of workflow design for most applications, it does not seem practical to attach complex operator structures to each task (i.e., three operator structures per task; input, success output, and failure output) in the network editor. So we provide a separate editor to organize the input transition operator and two output transition operators. The input operators are organized using a structure of AND-joins and OR-joins to combine transitions from other tasks. The two output transitions (one for success and one for fail) are organized using a structure of AND-splits, OR-splits, and a LOOP to distribute transitions to other tasks.

***Domain Editor***

The domain editor allows a designer to specify attributes of each domain (e.g., name, description). As mentioned in section 3.1.1, the dominance relationship among classification domains form a lattice. The domain editor allows the designer to specify the dominance relationship among classification domains. It also lets users change the GUI properties of classification domains (e.g., color). This editor provides a convenient place to specify receive and release policies between pairs of domains. This policy information can be used as a view into a complete list of policies that are described in a more comprehensive policy definition and enforcement tool.

***Data Editor***

Data for the workflow design tool is specified as an object interface. The data editor provides a graphical interface for a designer to specify new data and access already defined data. All data objects must extend an existing workflow data object, since the root workflow data object implements functionality required by the runtime of data object management. This is similar to the Java concept where the "Object" class is the root of the entire Java class hierarchy. The data editor allows the designer to specify the package, class name, what class the current class extends (single inheritance only), fields, and methods.

Workflow data is used for task invocations, outputs, and exceptions. In the case of exceptions, the data must extend an existing user exception or the root "UserException". Data is also used for the guards (conditional statements in operators). In a conditional statement, the designer will have access to all the fields and methods defined in the interface of the data object. The relationship among workflow data is shown in figure 5.

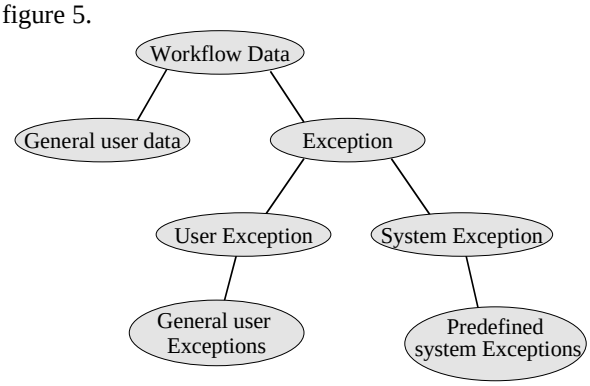


Figure 5: Data inheritance tree

***Role editor***

The role editor is similar to the classification domain editor in the sense that it allows a designer to define roles and the relationships among them (i.e., role hierarchies). In defining a role the designer can specify the name of the role, its description, and the privileges associated with the role. In general, role hierarchies reflect an organization's line of authority and responsibility. For example, if role A is higher than role B, then role A may have all the permissions that role B has and more. A designer can specify which role is more privileged than another in a given organization through this editor. The role editor will generate XML files on a per organization basis. These XML files can be used by an external application to assign users to roles and enforce permissions in the runtime system.

### 3.1.3. Coordination among Editors

Editors share common workflow related information and several of them might be displayed at the same time. If an object is modified after an editor displays information, then the editor needs to know about the changes so that it can refresh its display. We use a very simple scheme to ensure consistent display. There is an editor registry that maintains a list of active editors. When a user opens an editor, it registers itself to the registry. When a user closes an editor, it drops itself from the registry. When an editor modifies an internal workflow object, it notifies other active editors. It is each editor's responsibility either to update display if the editor uses the modified object or ignores the notification if the editor does not use the object.

The workflow design tool is not only a generic GUI-based, distributed programming tool, but also a good documentation tool that can capture the architecture of a complex distributed system design. Since the tool has to handle various inputs and outputs (e.g., mouse movement, context sensitive menu display), it becomes a fairly complex system. Since we do not want to make the design tool any more complex than is necessary, we created another module, the workflow compiler, to handle some functions that do not require much user interaction. In the next section we present the modules that bridge the gap between the workflow design tool and enactment services.

## 3.2. Workflow Compiler

There are two main reasons that we decided to separate the workflow compiler from the workflow design tool. First, even though the workflow design tool performs limited local design validation (e.g., task name conflict), it is logical to move global design analysis and code generation out of the workflow design tool for maintainability and extendibility reasons. Second, the workflow design tool is a generic distributed programming and documentation tool that can be used for many different purposes. Therefore, a different user community may have different analysis and validation requirements (e.g., vulnerability analysis). Also a different user community may prefer different enactment services. Hence we need to create a simplified (i.e., no GUI components) data structure that can be used by other people to write a new analyzer or runtime code generator.

Based on the above requirements, we structured the workflow compiler to perform three important tasks. They are:

- Analysis and validation of the design,
- Splitting an MLS workflow into multiple single-level workflows, and
- Generation of runtime code for enactment services.

The workflow compiler is organized in such a way that new analyzers or runtime code generators can be easily integrated. The internal structure of the workflow compiler is as shown in figure 6.

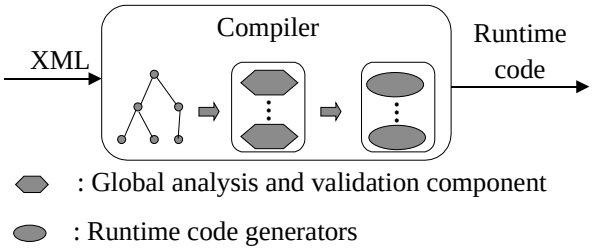


Figure 6: the structure of MLS METEOR compiler

The workflow compiler reads an XML representation of a design and converts it to an internal tree data structure. The analyzer and runtime code generator receive the internal data structure and perform their tasks. Runtime code generation largely depends on

- the specific runtime engine that will be used and
- the MLS infrastructure (see section 3.3.1) where the runtime engine is executed.

The process of splitting an MLS workflow design into multiple single-level workflows that we described in section 2 and also in [2] precedes runtime code generation. Initially, the extended version of OrbWork that is an implementation of METEOR is used as our target runtime engine. However, the structure of the MLS Meteor allows other runtime engines to be easily incorporated.

## 3.3. Runtime-system

An MLS workflow runtime management system accesses information in many classification domains. Therefore, the MLS requirement needs to be addressed by the runtime system. Our strategy for providing an MLS workflow management capability that can reduce MLS trust requirements is through composing multiple single-level WFMSs on a particular MLS infrastructure.

### 3.3.1. MLS Infrastructure

Composing an MLS workflow from multiple single-level workflows is the only practical way to construct a high-assurance MLS WFMS today. In this approach, the multilevel security of our MLS workflow does not depend on a single-level WFMS, but rather on the underlying MLS distributed architecture. The MLS distributed architecture will:

- Host multiple single-level workflows to be executed and

– Provide conduits for passing information among tasks in different classification domains.

A generic MLS distributed architecture is shown in Figure 7.

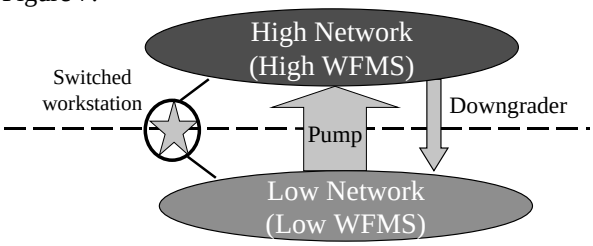


Figure 7: An MLS Distributed Architecture

In this architecture, switched workstations (e.g., "Starlight") enable a user to access resources in multiple classification domains and create information in domains that the user is authorized to access. One-way devices (e.g., a flow controller such as "A Network Pump") together with information release and receive policy servers provide a secure way to pass information from one classification domain to another. An information release policy server resides in a classification domain where the information is released, and an information receive policy server resides in a classification domain where the information is received as shown in Figure 8.

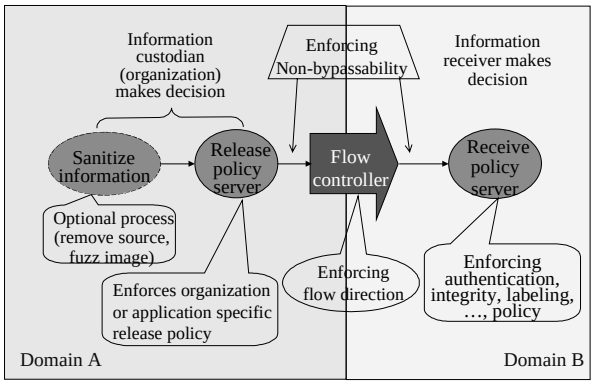


Figure 8: Information Release and Receive Policies in Conjunction with a Flow Controller

### 3.3.2. Single-level Enactment Services and Monitor

There can be many enactment services for a given workflow design. Currently we are using modified OrbWork [4] for our runtime engine. OrbWork is a single-level distributed workflow engine implemented in Java. It does not have a central scheduler for the whole workflow, rather there is a distributed scheduler per task that the workflow designer defined in the network editor. Each scheduler only knows its predecessors and successors.

Briefly, OrbWork works as follows. OrbWork schedulers are CORBA servers. Hence, they communicate with each other through CORBA's IIOP. OrbWork schedulers are also HTTP servers. When a human operator has to interact with a scheduler (e.g., human task), he can do so through the HTTP protocol. Also when a human workflow manager needs to intervene for some reasons, he can do so through the HTTP protocol.

OrbWork has two other CORBA servers: data servers and workflow monitor servers. Data servers act as a repository for data that needs to be passed among schedulers. The workflow monitor server receives progress reports from schedulers. Simplified communication paths among different components in OrbWork are shown in figure 9.

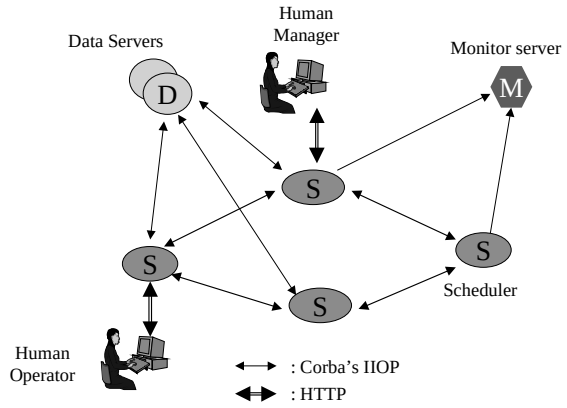


Figure 9: Communication among OrbWork components

### 3.3.3. MLS Enactment Services and MLS Monitor

As we presented in section 2 and in [2], our strategy for achieving MLS workflow is through the interoperability of single-level workflows that were generated from an MLS workflow design. For interoperability, the workflow runtime engine should be able to pass and receive the necessary information across domain boundaries. We extend OrbWork with the *synchronization node*. We can categorize synchronization nodes into release and receive synchronization nodes following our MLS infrastructure shown in figure 8. The responsibilities of synchronization nodes are as follows:

– To act as proxies for a task in another domain,
– To serve as exit and entry points to pass necessary information from one domain to another domain, and
– To ensure only proper information is passed to another domain (i.e., make sure release and receive policies are enforced).

For example, if there is an MLS transition from task A to task B as in figure 10, the release synchronization node acts as a proxy for task B and the receive synchronization node acts as a proxy for task A. Also release synchronization nodes serve as an exit point of the release domain and the receive synchronization node serves as an entry point for the receive domain. Therefore, release and receive policies can be enforced there.

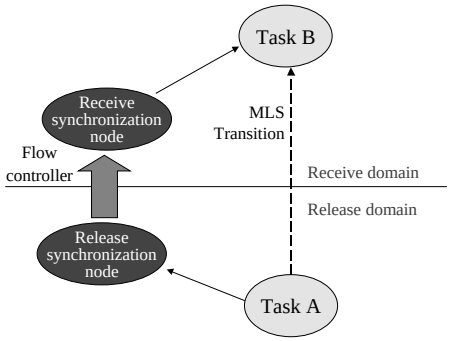


Figure 10: Break down of an MLS transition into multiple transitions

In our MLS workflow system, information exists as objects. Hence, there is the potentially that objects that contain variables and methods have to be passed across domain boundaries. Passing an object across classification domains can cause integrity and security violations. To mitigate risks associated with passing entire objects (i.e., state and definition of objects) across domain boundaries, currently, we only pass the necessary state information so that the objects can be reconstructed on the receiving domains. When a release synchronization node receives an object, it extracts the necessary information from the object and sends it through a flow controller. When a receive synchronization node receives, it instantiates the object in the receiving domain.

An MLS WFMS requires an MLS monitor. Workflow managers in a classification domain may need to know the progress of work in their classification domain and other domains that they are authorized to access. In other words, users of MLS workflow in different classification domains may have different views of the workflow they are running. Hence, an MLS WFMS should provide the capability to monitor activities in all domains the workflow manager is authorized to access. We are providing MLS monitoring capabilities that use similar techniques to those for data transfer from one domain to another. In other words, we place a monitoring proxy that receives monitoring information in a lower domain. This monitoring proxy corresponds to a release synchronization node and transfers the monitored information to a higher domain. In the higher domain, there is another monitoring proxy which corresponds to a receive synchronization node. This monitoring proxy relays the lower-level monitoring message to a higher-level monitor server. As in the case of data transfer, information must satisfy release and receive policy and must go though a boundary controller.

### *4. Conclusion*

In this paper, we presented the system organization of MLS METEOR and the rationale behind the organization. MLS METEOR is an example of an MLS application that can run on the MLS infrastructure that was presented at this conference last year [3]. In this approach, multilevel security enforcement of MLS METEOR does not depend on the single-level WFMS but rather on the underlying MLS infrastructure and a few security critical components (i.e., synchronization nodes that enforce release and receive policy, and boundary controllers). Therefore, MLS METEOR allows a workflow designer to concentrate on the functionality of the system he is building.

Prototypes of all components (i.e., workflow design tool, workflow compiler, and extended OrbWork as an enactment service) have been implemented. We are in the process of integrating those components. Future work includes adding more advanced features and addressing other aspects of workflow (e.g., survivability).

### References


1. V. Atluri, W-K. Huang and E. Bertino, “A Semantic Based Execution Model for Multilevel Secure Workflows,” Journal of Computer Security, To appear.

2. M. H. Kang, J. N. Froscher, B. J. Eppinger, and I. S. Moskowitz, “A Strategy for an MLS Workflow Management System” To appear in 13th IFIP Conference on Database Security, Seattle, WA, 1999.

3. M. H. Kang, J. Froscher, and B. Eppinger, “Toward an Infrastructure for MLS Distributed Computing,” 14th Annual Computer Security Applications Conference, Scottsdale, AZ, 1998.

4. K. Kochut, A. Sheth, and J. Miller, “ORBWork: A CORBA-Based Fully Distributed, Scalable and Dynamic Workflow Enactment Service for METEOR,” UGA-CS-TR-98-006, Technical Report, Department of Computer Science, University of Georgia, 1998.

5. Extensible Markup Language (XML) 1.0,” World-wide-Web Consortium, http://www.w3.org/TR/1998/REC-xml-19980210.html

6. R. Sandhu, E. Coyne, H. Feinstein and C. Youman, “Role-Based Access Control Models,” IEEE Computer, Vol. 29, No. 2, 1996.