Combining Domain and Type Enforcement with CORBA Security

Gregg Tally

Dan Sterne

Trusted Information Systems

April 3, 1997

Overview

- Mainstream operating system protection mechanisms
- Sigma project
- Domain and Type Enforcement (DTE)
- DTE applied to CORBA
- Prototype implementation
- Possible implementation with security-ready ORB
- Conclusion and discussion

Position Statement

- ORB and Security Service must take advantage of O/S protection mechanisms
- Mainstream operating systems do not provide appropriate protection mechanisms for object-level access control
- Domain and type enforcement provides operating system support for object-level access control

Reasons for Not Using O/S Protection Mechanisms

- Operating system protects system resources; not well-suited to protecting CORBA objects
- Efficiency; impractical to map each object to a unique process (CORBA Security, Sec. 3.9.3)
- Slow IPC discourages placing ORB in a separate process
- Multi-platform support may cause ‘lowest common denominator’ approach to using operating system features

Implications of Current Architectures

- Separately maintained ORB and operating system policies
- CORBA security mechanisms in a static library ORB:
  - cannot protect themselves from tampering or bypass
  - cannot exert control over application behavior

In a More Secure O/S and ORB Combination...

- Operating system and CORBA Security reference models compatible
- Operating system security mechanisms integrated into CORBA Security
- All without impacting performance

Background - Sigma Project

- Research project at TIS to develop and demonstrate security mechanisms for CORBA-based distributed systems
- Funded by Rome Labs, DARPA, and NSA
- Areas of research:
  - Access control based on Domain and Type Enforcement
  - ORB gateways
  - Distributed authentication

What is Domain and Type Enforcement?

- Extension of concept proposed by Boebert and Kain
- Kernel-based configurable mandatory access control mechanism for Unix
- All processes, including root processes, are subject to DTE restrictions
- Policy described in DTE Language (DTEL)
- Controls IPC and access to system resources
- Prototypes developed as part of government-funded research projects
- Additional information on DTE at http://www.tis.com

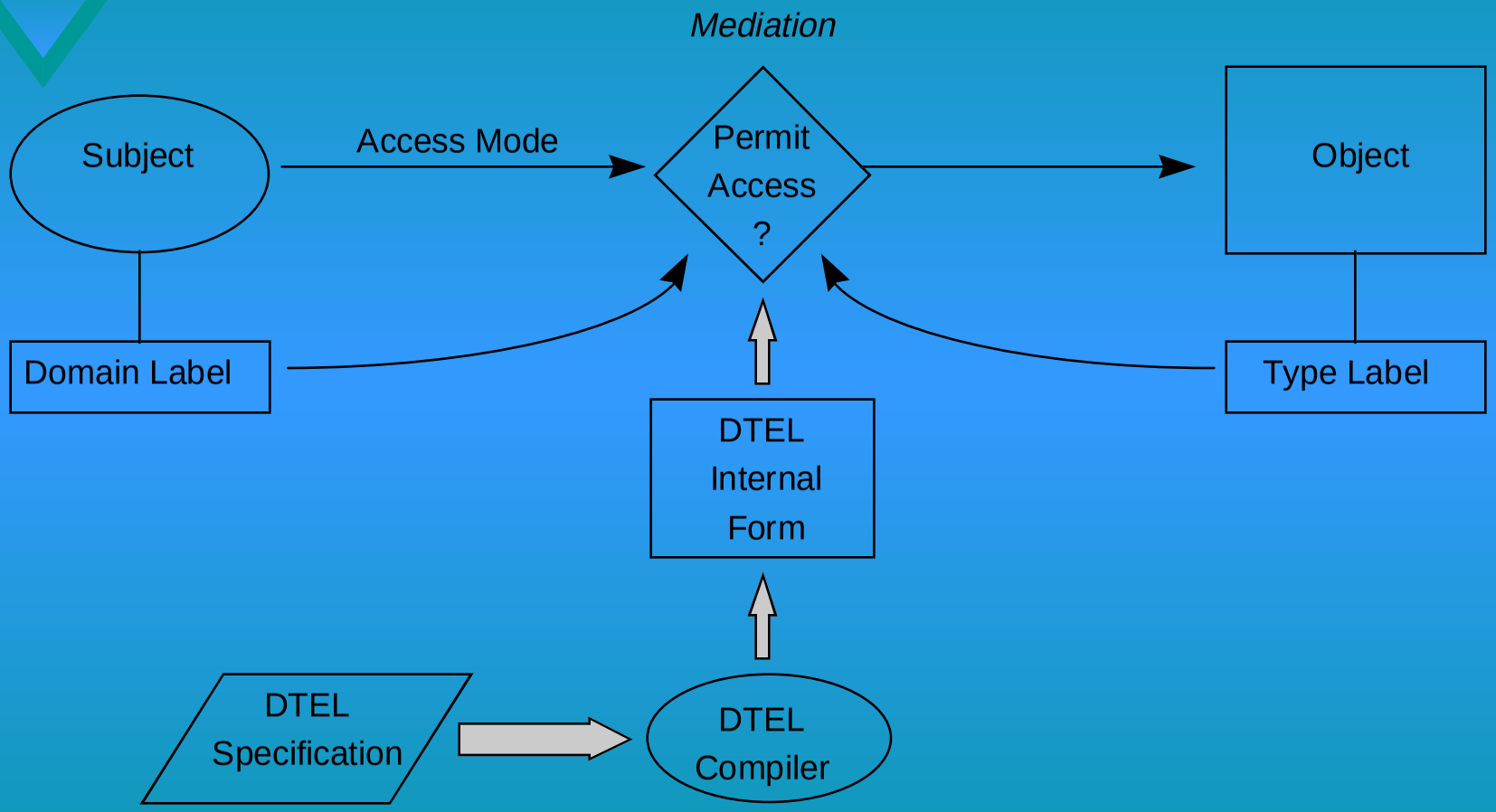
DTE Definitions

- A '*type*' is an attribute of a system resource
- All system resources are immutably typed
  - Files
  - Devices
  - IP Messages
- File and device types established at boot-up according to policy
- New files and directories are typed when created

DTE Definitions (cont'd)

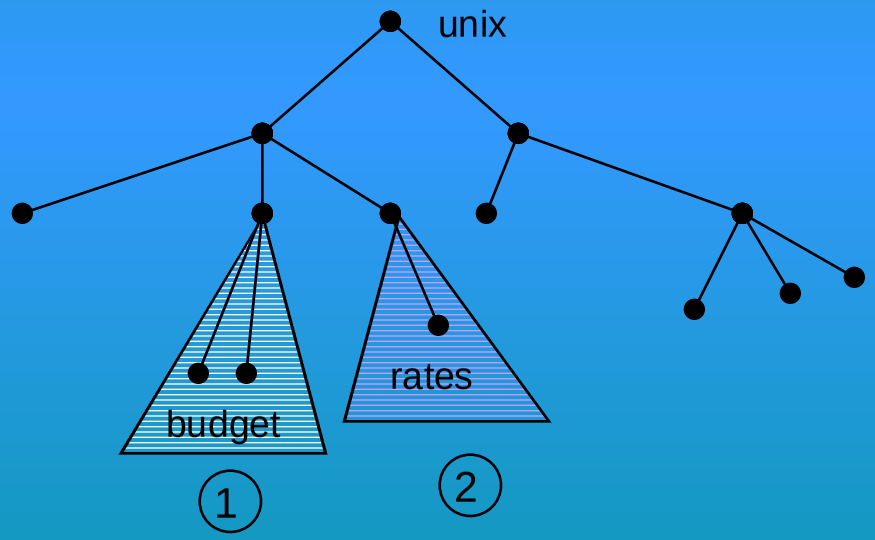
- A DTE '*domain*' is an attribute of a process
- Each process runs in a DTE domain and is constrained accordingly
- A DTE domain's rights include:
  - rights to types (read/write/delete/modify/create)
  - rights to other DTE domains (enter, signal, trace)
  - other privileges
- A DTE domain can be entered only by executing one of its "entry point" programs
- A *'role'* is defined by an initial DTE domain and a user authorization list

How DTE Works - File Access

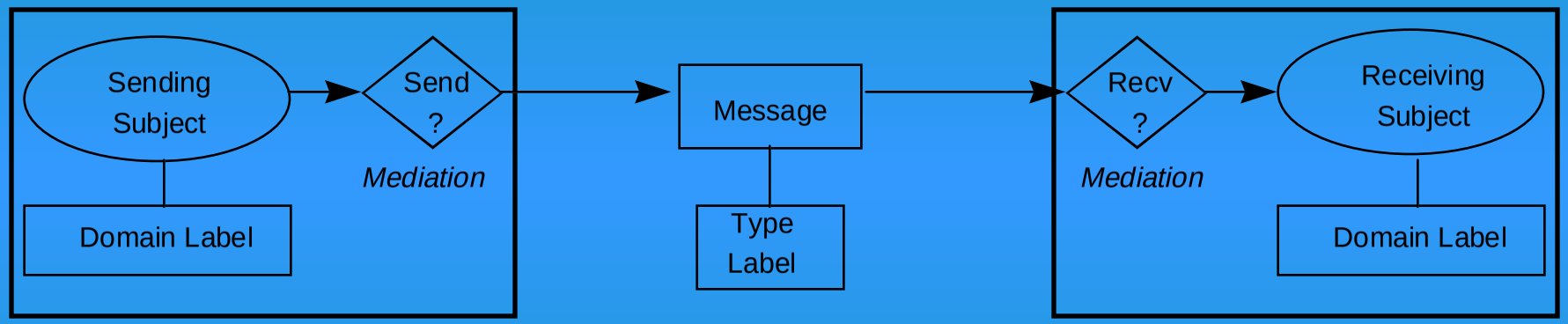


Implicit Type Bindings

Type attributes are “assigned” to directories and implied for all contained files and subdirectories.



How DTE Works - IPC



Mediation Occurs During Send and Receive Attempts

Messages May Be:

- Datagrams
- Segments of Streams

How DTE Works

- Kernel APIs unchanged except for new *optional* type and domain parameters; backward compatibility preserved
- A process may ‘exec’ a program in another domain if:
  - it has *exec* or *auto* rights to that domain and
  - the program is declared as an entry point to that domain
- Domain of a process set by:
  - Inheritance of parent process domain, or
  - Specified when process is exec’ed, or
  - Specification in policy

What is DTEL?

- DTEL is a language for expressing the DTE security policy
  - Declares types and domains
  - Specifies entry point programs and rights of each domain with respect to types and other domains
  - Assigns types to system resources
  - Assigns initial domains to roles
  - Assigns roles to users
  - Assigns “pseudo-domains” to non-DTE hosts for IP messaging

Example of DTEL

```
// Default type for all files
assign  -r          generic_t /;

// Protect security information
assign -r -s        dte_t     /dte;
assign    -s        passwd_t  /etc/{master.passwd, pwd.db, passwd};

// Types for executable areas
assign -r           binaries_t  /bin, /sbin, /usr/{bin, sbin};

// Critical device special files
assign    -s        disk_t    /dev/{rsd0a, rsd0b, rsd0g, rsd0h};

domain user_d =     SHELLS,
                    (crwd->generic_t),
                    (rwd->writable_t), (rxd->binaries_t),
                    (rd->readable_t, dte_t, passwd_t),
                    (auto->passwd_d);

// Identify DTE and non-DTE hosts
inet_assign         non_dte_d   0.0.0.0;
dte_systems         (figaro, 10.33.112.31), (thor, 10.33.112.98);
```

Goals of Object-Oriented DTE

- Extend operating system protection mechanisms to CORBA objects
- Consistent operating system and CORBA security administration
- Flexible access control specification:
  - granularity to the module, interface or operation
  - allow different accessability for objects of the same interface
- Improve scalability
- Provide simple mechanism to apply access controls to named groups of objects
- Remain compatible with existing CORBA standards

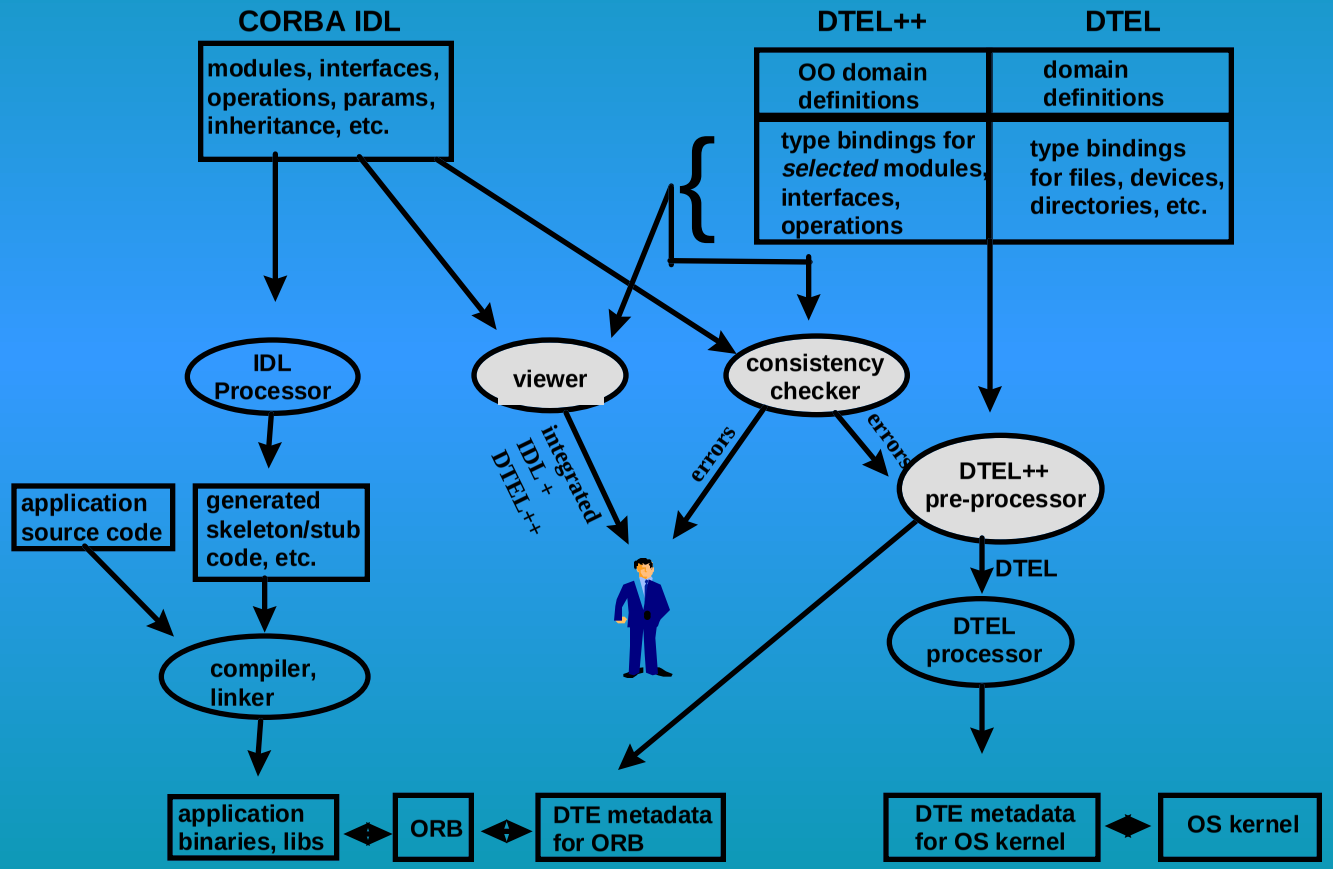
What is DTEL++?

- Extends DTEL to include CORBA operations
  - *assign* statement binds a type to individual operations
  - a type may be assigned to groups of operations
  - default operation types may be assigned to modules and interfaces
  - optional flags control type assignments for inherited interfaces
- DTE domains given *invoke* and *implement* rights for types
- The *default type template* is the union of assign statements for an interface and its base interfaces
- *Named type templates* allow security aware applications to have different access controls for objects of the same interface

Enforcement Mechanisms

- Access to operations enforced by combination of ORB and kernel
- All operation invocations are '*typed*'; type labels are applied to requests and replies by the ORB and kernel
- Kernel mediates IPC between client and server
- No changes to IDL or application code for security unaware applications

DTEL++ Processing



- Results of Sigma Project

Example 1: Scalability

IDL

```
module Library {
    struct BookDescription {...};
    interface Patron {...};
    interface PatronDatabase {...}

    interface Book {
        readonly attribute BookDescription desc;
        void checkOut (in Patron);
        void checkIn ();
        long numberAvailable ();
        long numberReservations ();
        void reserve (in Patron);
    };

    interface BookDatabase {
        Book newBook (in BookDescription desc,
                      in long copies );
        void removeBook ( in Book );
        Book findByTitle (in string title);
        Book findByAuthor (in string author);
        Book findBySubject (in string subject);
    };
}
```

A library system used by:

- ***patrons*** to find books in the catalog and place reservations for books.
- ***librarians*** to check books in and out, add new books to the system, and help patrons find books

Example 1: Scalability

IDL

```
module Library {
    struct BookDescription {...};
    interface Patron {...};
    interface PatronDatabase {...}

    interface Book {
        readonly attribute BookDescription desc;
        void checkOut (in Patron);
        void checkIn ();
        long numberAvailable ();
        long numberReservations ();
        void reserve (in Patron);
    };

    interface BookDatabase {
        Book newBook (in BookDescription desc,
                      in long copies );
        void removeBook ( in Book );
        Book findByTitle (in string title);
        Book findByAuthor (in string author);
        Book findBySubject (in string subject);
    };
}
```

DTEL++

```
type     safe_t, restricted_t;

assign  restricted_t Library;  // default type
assign  safe_t     Library.BookDatabase.{
                        findByTitle,
                        findByAuthor,
                        findBySubject };
assign  safe_t     Library.Book.{
                        _get_desc,
                        numberAvailable,
                        numberReservations,
                        reserve };
```

- Default all operations to restricted_t
- Some ‘safe’ operations need less restrictive access

Example 1: Scalability

IDL

```
module Library {
    struct BookDescription {...};
    interface Patron {...};
    interface PatronDatabase {...}

    interface Book {
        readonly attribute BookDescription desc;
        void checkOut (in Patron);
        void checkIn ();
        long numberAvailable ();
        long numberReservations ();
        void reserve (in Patron);
    };

    interface BookDatabase {
        Book newBook (in BookDescription desc,
                      in long copies );
        void removeBook ( in Book );
        Book findByTitle (in string title);
        Book findByAuthor (in string author);
        Book findBySubject (in string subject);
    };
}
```

DTEL++

```
type     safe_t, restricted_t;

assign  restricted_t Library;  // default type
assign  safe_t      Library.BookDatabase.{
                        findByTitle,
                        findByAuthor,
                        findBySubject };
assign  safe_t      Library.Book.{
                        _get_desc,
                        numberAvailable,
                        numberReservations,
                        reserve };
```

```
domain  patron_d      PATRON_PROG,
                      invoke->safe_t;

domain  librarian_d   LIBRARIAN_PROG,
                      invoke->restricted_t,
                      invoke->safe_t;
```

```
domain  server_d      SERVER_PROG,
                      implement->{safe_t,
                                  restricted_t};
```

Example 2: Order Dispatching

IDL

```
module Dispatch {
    interface OrderFactory {
            Order createOrder ();
    };

    interface Order {
            // Omitted operations to set address, time, etc.
            void assignDriver ( inout Driver assignedDriver );
            void accept ();
            void complete ();
    };

    interface ExpeditedOrder : Order {  // Derived interface
            void set_priority ( in long priority);

    }
```

A system for dispatching parcel delivery and pickup orders to drivers

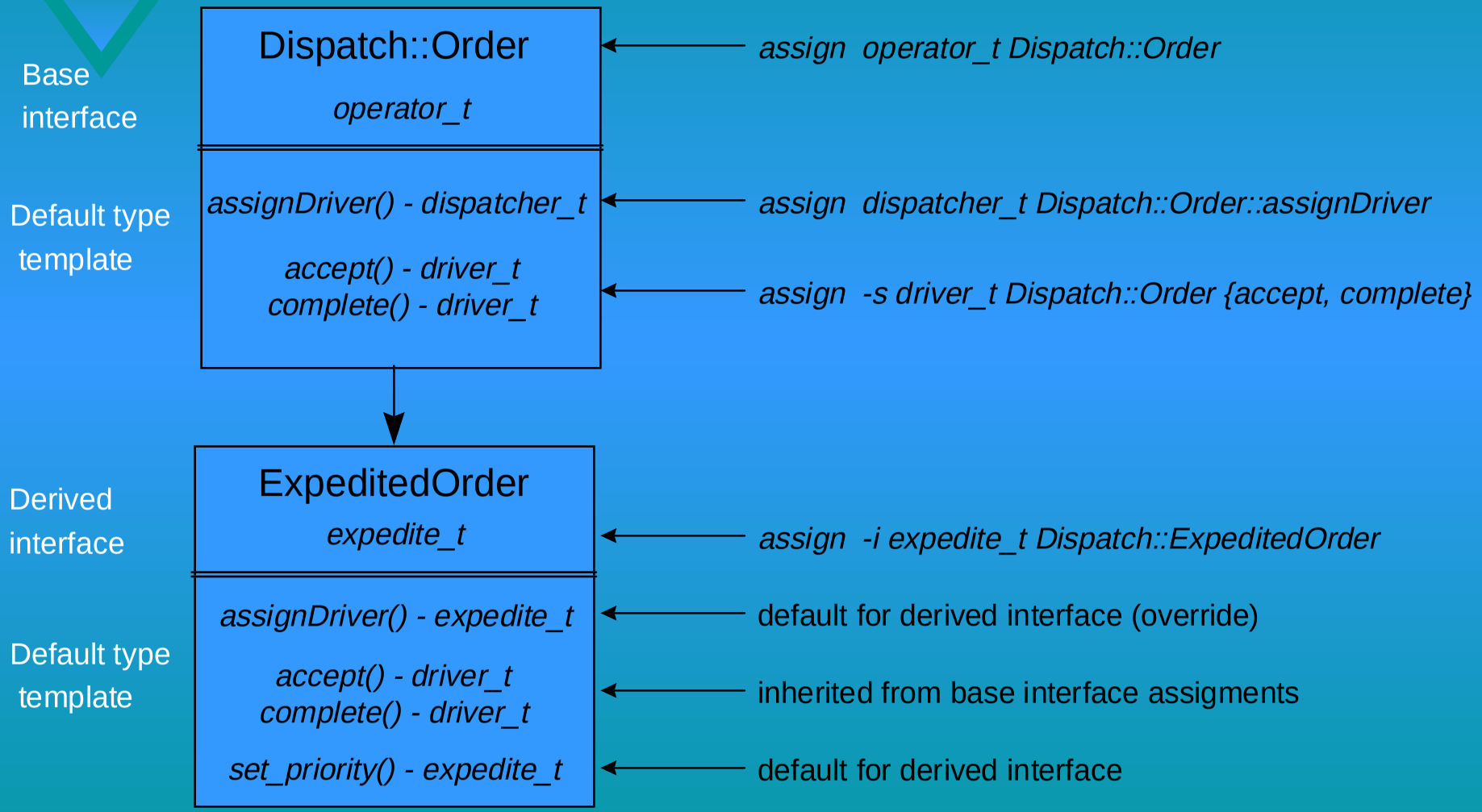
Users are:

- Operators - create orders
- Dispatchers - assign orders to drivers
- Drivers - accept/reject assigments, complete orders

Example 2: Applying Fine-Grained Control

```
// Define possible types for application
type generic_t, internal_t, dispatcher_t, driver_t, operator_t, expedite_t;

// Default typing for all application objects
assign   generic_t        CORBA::Object;    // Default to 'world' access

// Module and interface default typing
assign   internal_t       Dispatch;         // Internal users only

// Single and multiple operation typing
assign   dispatcher_t     Dispatch::Order::assignDriver; // Dispatchers only
assign -s driver_t        Dispatch::Order::{accept, complete}; // Drivers only

// Strict typing and inherited interface typing
assign -s operator_t      Dispatch::OrderFactory::createOrder;  // Operators
assign -i expedite_t      Dispatch::ExpeditedOrder;    // Special handling
```

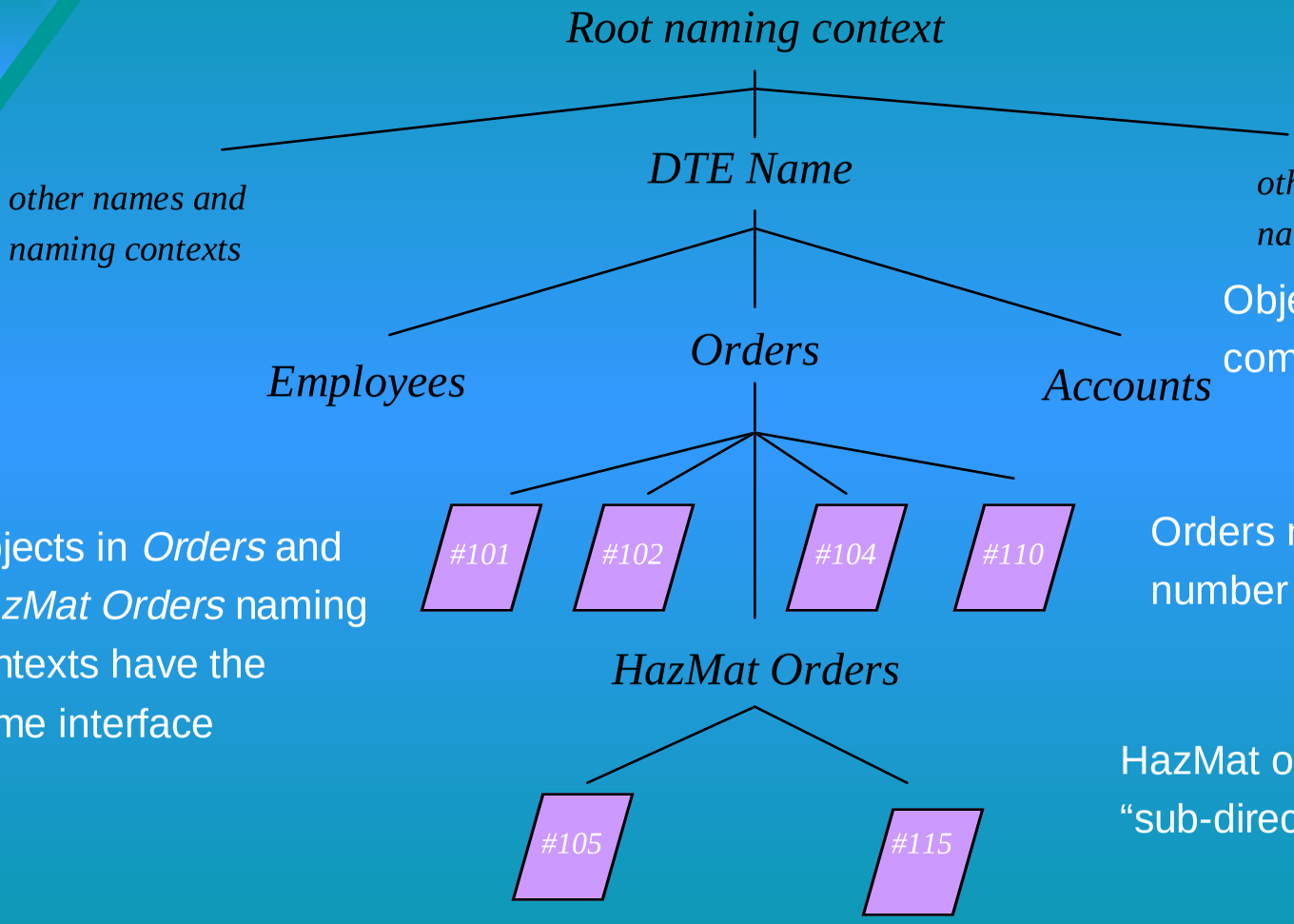
Example 2: DTEL++ and IDL



Named Type Templates

- Allows objects with the same interface to have different type-to-operation bindings
- *Named type templates* contain groups of *assign* statements mapping types onto operations
- Named type templates are assigned to objects with *DTE Names:*
  - DTE Names are CORBA Naming Service names
  - Policy can assign type template to a specific object name or a name context
  - A DTE Name is a Property of an object
  - DTE Name property can only be set at object creation

DTE Names



Objects grouped by common characteristics

Objects in *Orders* and *HazMat Orders* naming contexts have the same interface

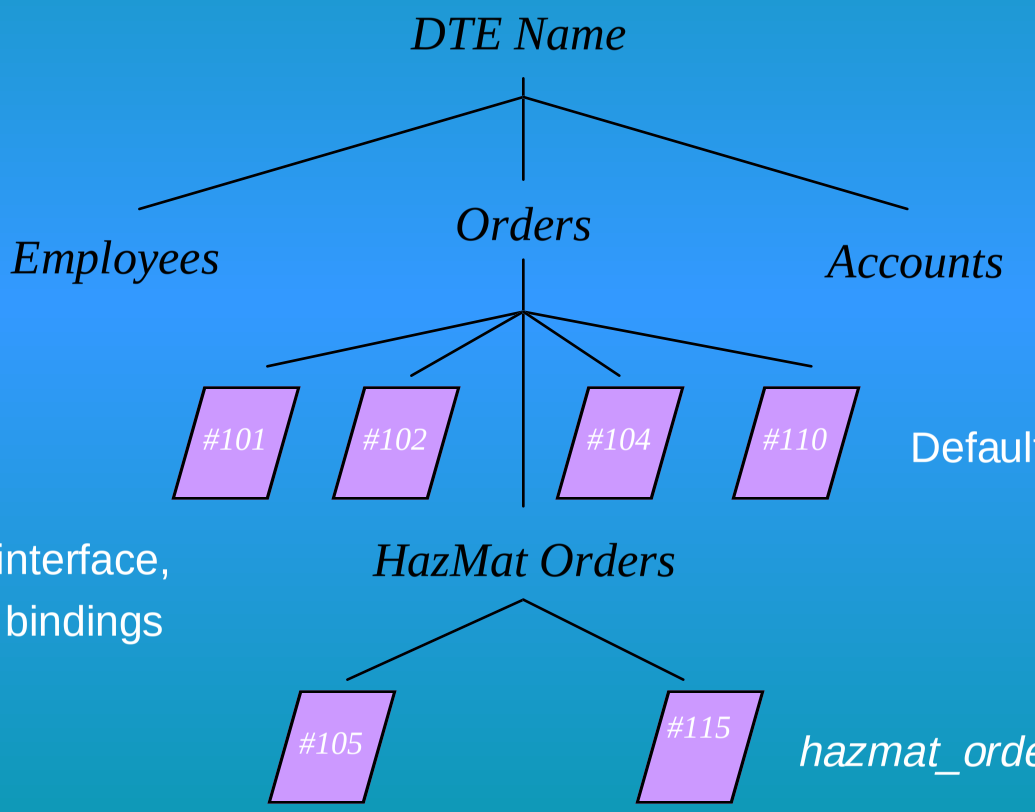
Orders named by number

HazMat orders in "sub-directory"

Example of DTEL++ Per-Object Type Bindings

```
// Template defined
template hazmat_order_template {
   assign hazmat_t     Dispatch::Order::{accept, complete};
}

// Template bound to portion of object namespace
assign hazmat_order_template  .Orders.HazMat;
```

DTE Names with Templates

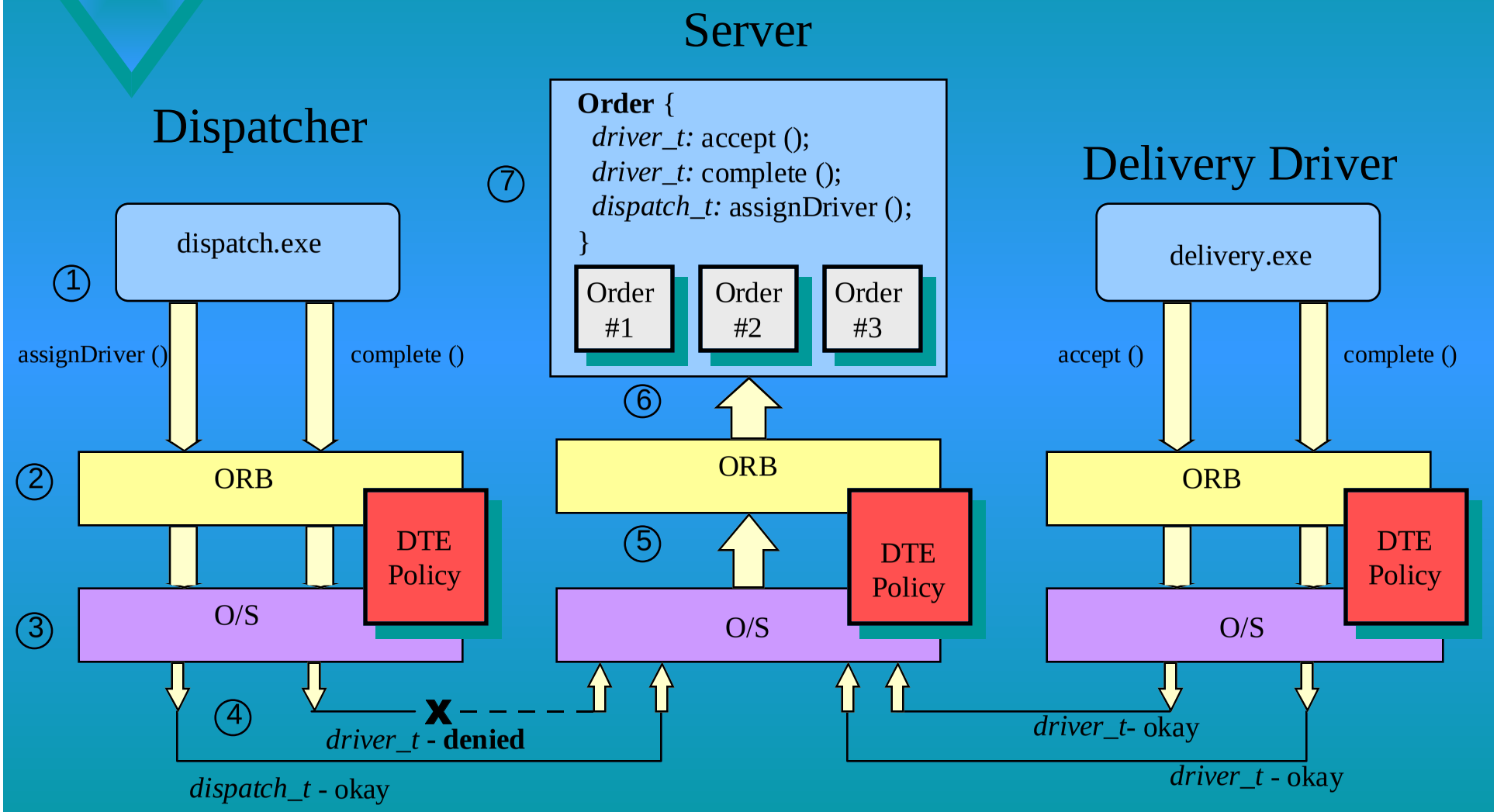


Objects of same interface,
but different type bindings

Overview of ORB Supporting DTE

- For each type bound to an operation, the DTEL++ pre-processor creates a pair of types for the request and reply messages
- Kernel mediates communication between client and server processes through DTE-enhanced IPC services:
  - Process's DTE domain requires write permission for message type in order to send message
  - Process's DTE domain requires read permission for message type in order to receive message
  - Server kernel performs access control for non-DTE hosts
- Sending ORB determines type for request and reply messages; receiving ORB validates type

Access Mediation



Prototype Implementation of Per-Object Type Bindings

- Uses Naming Service, Property Service, LifeCycle Generic Factory
- Policy assigns a DTE type template to objects under a “DTE_Name” naming context
- Factory creates objects with a “DTE_Name” property provided as a criteria to the create_object operation and binds newly created object to the given name under the “DTE_Name” naming context
- On operation invocation, ORB uses the type template assigned to the object’s “DTE_Name” to determine the type for the operation

Description of ILU Implementation

- ILU uses a static library implementation of the ORB
- TIS modified the ILU library to perform type assignments and validation
- Internal ILU structures extended to hold the type information
- Runs on DTE-enhanced BSD/OS

Initial Results of Sigma Experience

- Modified ILU on DTE kernel provides role based access control for small demonstration application
- No application changes required to accommodate DTE when only the default type templates are used
- Types and mediation applied to all operation requests and replies
- Both DTE and non-DTE clients can interoperate with DTE servers
- Negligible impact on performance
- Named type templates in progress

Possible Implementation with Security-Ready ORB

- Pass template name, sending domain, and message type in the generic security mechanism tag using the *mech_specific_data*
- Use request-level interceptor (client_invoke) to add domain and type information to IOR tags and perform access check on client
- Use request-level interceptors (target_invoke) on server to perform access check
- If kernel supports DTE, add type and domain information to IP header through message interceptor (requires access to target IOR)

Combining DTE and CORBA: Potential Benefits

- Access Control Policy
  - Cohesive, unifed representation of CORBA and OS policies
  - Scalable or fine-grained control
  - Supports interface inheritance
- Enforcement
  - CORBA Security supported by non-bypassable, tamper-proof kernel mechanisms
  - Trojan horse protection; kernel can limit code that runs with user's access rights
  - Object servers need not be cognizant of users and user authorization

Planned Enhancements

- IPSEC for transport security
- Dynamic DTE policies
  - Load/unload policy modules
  - Automated distribution of policy updates across hosts
- OO-DTE without DTE kernel
- OO-DTE plug-in for COTS ORB via replaceable security service interfaces