Secure Interoperation of Secure Distributed Databases

An Architecture Verification Case Study

Fred Gilham, R. A. Riemenschneider, and Victoria Stavridou

Computer Science Laboratory
SRI International
{gilham,rar,victoria}@csl.sri.com

Abstract. This paper describes the process of implementing an architecture for secure distributed transaction processing, the process of verifying that it has the desired security properties, and the implementation that resulted. The implementation and verification processes provided us with valuable experience relevant to answering several questions posed by our research on transformational development of architectures. *To what extent can implementation-level architectural descriptions be derived from abstract description via application of transformations that preserve a broad class of properties, which includes satisfaction of various access control policies? To what extent can a formal derivation of a non-secure implementation-level distributed transaction processing architecture be reused in derivation of a secure architecture? Are the transformation verification techniques that we have developed sufficient for verifying a collection of transformations adequate for implementing complex secure architecture? Do our architecture hierarchies effectively fill the gap between abstract, intellectually manageable models of a complex architecture and the actual implementation?* Exploring the answers to these questions resulted in a reference implementation of an architecture for secure distributed transaction processing, and an independently interesting demonstration instance of the reference implementation.

1 Introduction

The primary focus of our research efforts is on the problem of producing *dependable system architectures.* We have recently developed the basic technology required to effectively solve this problem in the case of distributed transaction processing systems. Our solution involves extending the ACID properties that drove the development of the X/Open standard for distributed transaction processing by adding security constraints. Specifically, we added the constraint that the system must satisfy a multilevel secure access control policy. Our basic approach to defining a family of architectures that satisfy the extended requirements was described in an earlier paper [8]. This paper is devoted to describing our experiences following that approach to its logical conclusion: an implemented architecture for secure distributed transaction processing that has been proven secure.

2 Architecture Hierarchies

It is common practice to informally describe a single architecture at many levels of abstraction, and from a variety of different perspectives. For example, at a concrete level, the architecture might be described in terms of how a system is assembled from code components using connection mechanisms provided by a programming language or operating system. At a very abstract level, the same architecture might be described in terms of the data and control flow paths that link the functions the system is required to perform. At an intermediate level, the system architecture might be described in terms of clients and servers interacting through message queues.

One reason that multiple descriptions are provided is that each of these architectural descriptions is useful for some purposes, but less useful for others. Often the argument that the system has some desirable architectural property can be greatly simplified by employing an abstract description of the architecture. On the other hand, some properties require detailed analysis of low-level architectural descriptions. Another reason is that there is often a major conceptual gap between abstract architectural descriptions and the actual implementation. Descriptions at intermediate levels of abstraction break this gap down into intellectually tractable pieces, and thus can significantly increase confidence that the abstract description correctly characterizes the as-implemented architecture.

While much of the recent work on more formal architectural description languages (ADLs) has focussed on the problem of describing architectures in a way that provides a suitable basis for various formal analyses, our own work has focussed primarily on the problem of formally linking these several descriptions in an *architecture hierarchy.* The basic idea is to regard each description as a logical theory, the theory of the class of architectural structures that the description describes, and to formalize the links between the descriptions as interpretation mappings. Figure 1 shows the structure of a typical architecture hierarchy. Correctness of the hierarchy can be explicated in terms of constraints on the interpretation mappings. For example, one might require that the interpretations preserve some property of special interest (say, some security property). Or one might require that they preserve every property expressible in some language.

Our tools for constructing architecture hierarchies are based upon an incremental transformation paradigm. The result of applying a transformation to a hierarchy is a hierarchy containing additional or altered descriptions. An additional description is typically either an abstraction of another description in the hierarchy, a refinement of another description in the hierarchy, or a description at the same level of abstraction but from a different perspective (e.g., a data-oriented description that complements an existing function-oriented description). If a hierarchy has been constructed entirely by applications of transformations that are known to be generally correctness-preserving, or whose correctness in the particular instance are checked as they are applied, the hierarchy is known to be correct. Correct hierarchies formally link very abstract formal architectural descriptions, which can easily be demonstrated to satisfy system requirements,

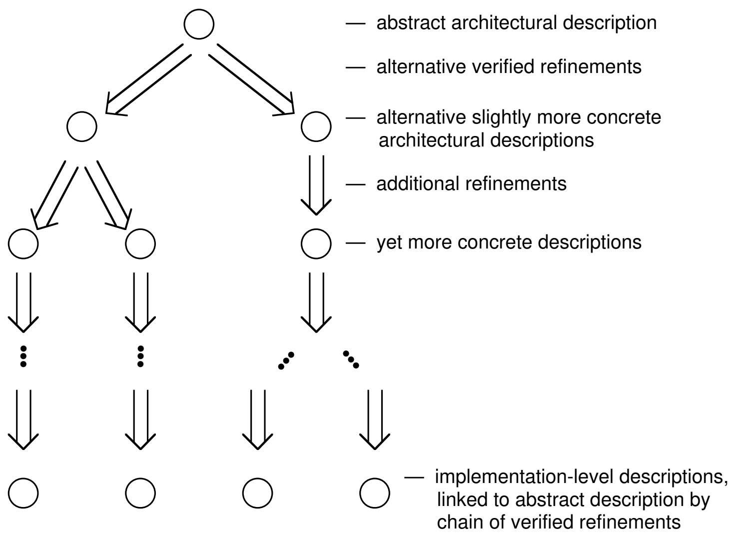

Fig. 1. An Architecture Hierarchy

to implementation-level architectural descriptions in a way that guarantees the implementation satisfies the requirements.

3 Distributed Transaction Processing

In response to a growing demand for increased interoperability of software compoenents, a number of vendor-neutral, open software architectures have been proposed as standards. The X/Open Distributed Transaction Processing (DTP) reference architecture [17] is one of these standards. X/Open DTP is specifically intended to standardize interactions among the components of a three-tiered client/server model: resource (e.g., database) managers (RMs), an application (AP) that accesses those resources, and a transaction manager (TM). A very abstract, informal representation of this architecture — which can be found in the X/Open documents — is shown in Fig. 2.

The role of the TM is to ensure that consistency is maintained when the state of the RMs is changed by the AP. Transactions are collections of operations on the RMs. Multiple transactions can be performed concurrently. In the context of a transaction, all other transaction appear to be executed atomically. In other words, it always appears that all or none of the operations that make up a transaction have taken effect.

The X/Open documentation is informal, consisting of a mix of C header files[1] and English text describing the semantics of the functions declared in

[1] Since a description of how components written in languages other than C, including languages such as COBOL that do not support return values, can be accomodated is

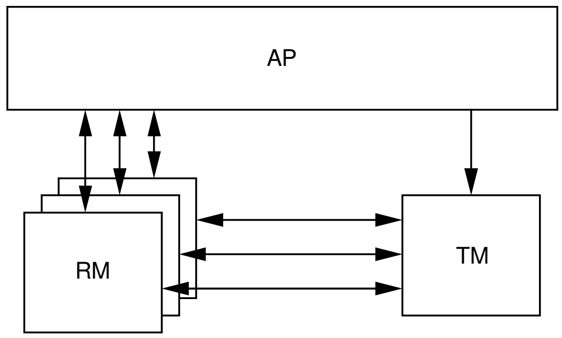

Fig. 2. The X/Open DTP Reference Architecture

the headers. The protocols for using these functions are also only semi-formally characterized.

4 Adding Security to DTP

Security requirements are ubiquitous in transaction processing, both within the defense sector and within the commercial sector. Tremendous leverage can be gained by addressing security concerns at the architectural level, because ensuring security is typically both technically challenging and expensive. Several of the open software architectures have introduced extensions to deal with security issues. For example, recent updates of OMG's CORBA specification have included security services that address fundamental security concerns in a distributed object system, based on a trusted ORB model. These services include credential-based authentication of principals and their clients, several simple privilege delegation schemes, authorization based on access control lists, audit trails, and non-repudiation services based on the ISO non-repudiation model.

One of the key security issues in DTP is enforcement of an *access control policy*, which ensures that classified resoures can only be accessed by clients with appropriate clearances. In terms of the Bell-LaPadula model [4], our proposed multilevel secure (MLS) access control policy for secure DTP can be defined by saying that the distributed transaction processing systems must have the following two properties:

- *The Simple Security Property* — A client is allowed read access to a resource only if the client's clearance level is greater than or equal to the resource's classification level.
- *The* $\star$*-Property* — A client is allowed write access to a resource only if the client's clearance level is less than or equal to the resource's classification level.

included, it is clear that the C code cannot be considered to be a formal specification of the syntax of the interface functions.

The simple security property guarantees that there is no *read-up* of data, while the $\star$-property guarantees that there is no *write-down*.

To say that a component in a DTP system is MLS means that the component satisfies the MLS access policy internally, and that all inputs to and outputs from the component are properly classified. So, for example, to say that a multilevel database management system (DBMS) is an MLS resource manager means that

- any data entered into the database is tagged with a classification level,
- any data retrieved from the database comes tagged with the same classification level that it had when it was entered, and
- no operation performed by the DBMS can result in either read-up or write-down among the subcomponents of the DBMS.

(Note that a single-level database is trivially an MLS resource manager, provided single-levelness is maintained.) Similarly, to say that the entire DTP system is MLS means that no read-up or write-down occurs between components and that inputs and outputs are properly classified.

The challenge in adding an MLS access control policy to X/Open DTP is to define an architecture which guarantees that, if the components used to build the system are MLS, then the DTP system as a whole is MLS. That this is not a trivial problem can be seen by considering a DTP system composed of an application that accesses several single-level databases, each of which has a different classification level, a very common situation. A secure DTP (SDTP) architecture must ensure that the client application does not read data classified above the application's clearance level, that it does not write data that it obtained from a highly classified database to a database classified at a lower level, and so on.

One complicating factor is that no single architectural structure is well suited to every SDTP system. It is easy to see that a single-level client AP, a single-level TM, and a collection of single-level RMs all at the same level can be linked by secure connectors to build a system that is automatically MLS secure. Extending the X/Open DTP architecture to handle such systems can be accomplished simply by adding appropriate encryption to the communication protocols. But varying single-level systems will require that the connectors linking the AP to the RMs somehow enforce the access control policy (at least in the absence of a certification that the AP satisfies security constraints beyond being MLS). One way of doing so is to introduce a Security Manager component that mediates communication between the RMs and the MLS AP, as shown in Fig. 3.

5 Defining the SDTP Hierarchy

As part of an earlier effort to develop an architectural description language suitable for defining architectural hierarchies, we defined a formal hierarchy for the X/Open DTP standard consisting of 17 SADL [9][2] specifications, linked by

[2] SADL, pronounced *saddle*, is an acronym for 'Structural Architecture Description Language'. The name was suggested by the fact that SADL emphasizes description of structure rather than description of the behavior of connectors, a topic quite adequately addressed by other ADLs.

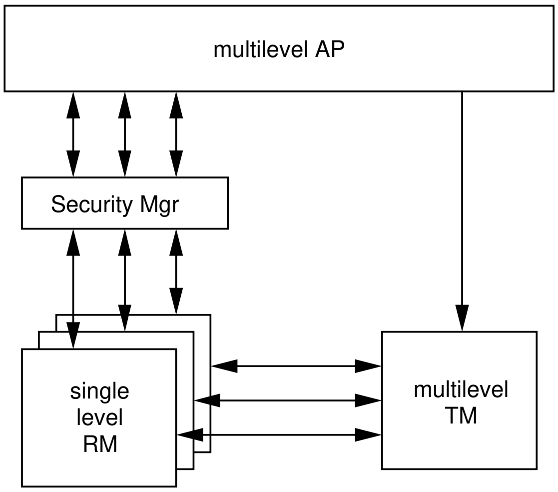

Fig. 3. SDTP Architecture for Varying Single-level Resource Managers

SADL-specified interpretation mappings. The most abstract description, shown in Fig. 4, approximately corresponds to the informal diagram shown in Fig. 2. Six levels below the most abstract description is a description at roughly the same level of abstraction as the code in the X/Open documentation. (Excerpts from this code are shown in Fig. 5.) Below this level, the hierarchy branches in several directions, reflecting, e.g., the various choices that can be made in allocating low-level interprocessor communication functions to processors.

Our basic approach to developing a similar hierarchy for SDTP architectures was to treat SDTP as a *rearchitecting* problem: given an architecture that satisfies a set of requirements, how can that architecture be modified to satisfy an additional requirement, viz., that the MLS access policy is satisfied. The idea is to attempt to "replay" the derivation of the implementation-level architecture descriptions from the most abstract architectural description. An architecture hierarchy includes a record of the transformations used to generate the interpretations that link the descriptions. If a transformation used in the original X/Open DTP hierarchy can be shown to preserve satisfaction of the additional requirement — i.e., in this case, if the transformation can be shown to preserve the simple security property (no read-up) and the $\star$-property (no write-down) — then the same transformation can be employed in deriving an SDTP implementation.[3]

A priori, it was not known what percentage of the design history encoded in the X/Open DTP hierarchy could be reused in the SDTP hierarchy. We therefore

[3] The bindings of the variables in the transformation can, and usual do, change. But the bindings in the replay can be automatically determined from the bindings in the original derivation by tracking the correspondence between objects in the two hierarchies as the new hierarchy is constructed from the old.

```
x_open_abstract: ARCHITECTURE [ -> ]
  IMPORTING ALL FROM Dataflow_Relations_style
BEGIN

CONFIGURATION
  ap: TYPE <= Function
  rms: TYPE <= Function
  tm: TYPE <= Function

  the_ap: ap
  the_rms: rms
  the_tm: tm

  ar: CONSTRAINT = Dataflow(the_ap, the_rms)
  tx: CONSTRAINT = Dataflow(the_ap, the_tm)
  xa: CONSTRAINT = Dataflow(the_tm, the_rms)

END x_open_abstract
```

Fig. 4. Dataflow-Level SADL Description of X/Open Architecture

developed implementation-level descriptions of the three abstract architectures for SDTP that we have described in an earlier publication [8]. It turned out that over 90% of the design decisions made during the implementation of each of the three SDTP architectures could be based on decisions recorded in DTP hierarchy; conversely, every design decision in the DTP hierarchy was reused in the development of every one of the three SDTP hierarchies. In the most extreme case, the hierarchy for the "single-level, same level RMs" case is nearly identical to the DTP hierarchy, the only real difference being additional low-level decision to use an RPC mechanism that performs encryption. Thus, one important result of this case study was evidence that confirms our hypothesis that derivation replay is often an effective rearchitecting technique.

6 Verifying Security

We believe that our implementation-level SDTP architectural descriptions define architectures that have the simple security property and the $\star$-property, because we have formally proved that these properties follow from a dataflow-level description of the SDTP architectures in terms of components linked by channels that enforce security and we are in the process of formally confirming our belief that the design decisions in the SDTP hierarchies preserve security. The proofs that the design decisions preserve the desired security properties use two different verification techniques we have developed.

Many of the transformations employed in development can be shown to preserve a broad class of structural properties by showing that the implementation links they introduce into the hierarchy are *faithful interpretations* of the theory

```
x_open_std_level: ARCHITECTURE [ -> ]
%% X_Open_style defines relevant
%%  properties of procedure calls;
%%  this description covers how things
%%  are hooked together
  IMPORTING ALL FROM X_Open_style , ...
BEGIN
  ...
COMPONENTS
  tm: TYPE <= ARCHITECTURE [ -> ]
     EXPORTING ALL
     BEGIN
       register: AX_Register_Procedure
         [id: X_Id, rmid: INT, flags: INT
                 -> ret: INT]
       begin: TX_Begin_Proc [ -> ret: INT]
       ...
     END tm
  ...
  the_tm:  tm
  ...

CONFIGURATION
  tx: CONSTRAINT =
         Called_From(the_tm.begin, the_ap)
           AND ...
  xa: CONSTRAINT =
        (FORALL y: rm)
          [Called_From(the_tm.register, y)
            AND ...
  ...
END x_open_std_level
```

Fig. 5. X/Open Standard-Level SADL Description of X/Open Architecture (Heavily Elided)

of the more abstract description in the theory of the more concrete description. To say an interpretation μ of the language of theory T_1 in theory T_2 is a faithful (theory) interpretation means that, for every sentence (i.e., closed formula) $\mathbf{A}$ in the language of T_1,

$$T_1 \vdash \mathbf{A} \quad \text{if and only if} \quad T_2 \vdash \mu(\mathbf{A})$$

In general, access control policies prohibit certain dataflow paths in the system. Simply by omitting the undesired dataflow channels from the abstract architectural description, we guarantee that no formula saying *there is a dataflow from* C_1 *to* C_2, where a flow from component C_1 to component C_2 would violate the access control policy, can be proved from the theory of the abstract description. Faithful interpretations cannot introduce new dataflow channels, but can only further elaborate existing channels. Therefore, transformations that can be shown to always introduce faithful interpretations can freely be used in the process of implementing a design that has been verified to be secure, without fear that security will be violated. The details of our model-theoretic method for proving that transformations always introduce faithful interpretation links has been described in a series of earlier papers [7, 13, 15]. Currently, we manually prove that transformations introduce faithful theory interpretations, though we are actively investigating the possibility of using the SRI's PVS verification system to provide automated support for the process.

On the other hand, not every interpretation link in the SDTP hierarchies has been shown to be a faithful theory interpretation. In fact, some of the transformations employed in building the hierarchies do not have the property that they always introduce faithful interpretations. For these interpretations, we apply a *checking* technique to show that the interpretation preserves the desired security properties. The basic idea is to use the interpretations that refine the architectural description to refine the proof that the descriptions have the desired security properties as well. If the result of applying the interpretation to the security proof at the more abstract level can be used to automatically obtain a security proof at the more concrete level, the implementation step has been shown to preserve security. (This technique has been described in greater detail in an earlier paper [14].) Currently, we apply the transformations to proofs manually, then use PVS to check the results. A tool that automates proof transformation is under development.

7 Implementing an SDTP Reference Architecture

In this SDTP reference implementation the RMs consist of a Security Manager (SM) that enforces the desired multilevel security policies — each of which implements a piece of the Security Manager component shown in Figure 3 — and a set of single-level RMs, which are arbitrary single-level databases. A multilevel-secure RM consists of the combination of the SM and the single-level RMs. Note that the single-level RMs are not required to provide the X/Open DTP standard services themselves; they must, however, support transaction processing.

The X/Open DTP standard specifies various services that the components of the architecture must provide and make use of. An SDTP application will make use of two services: the **TX** (Transaction Demarcation) service, and the **AR** service (the service that allows the application access to shared resources). The **TX** service is provided by the TM while the **AR** service is provided by the (multilevel) RMs the application makes use of.

The TM and RMs also make use of X/Open DTP-specified services. The service the RMs provide to the TM is called the **xa** subservice of the **XA** service. This subservice implements the interface the TM uses to perform the two-phase commit protocol. The **ax** subservice of the **XA** service is provided by the TM and allows RMs to dynamically register themselves as being under the TM's management, and unregister themselves when that management is no longer desired.

The X/Open DTP standard gives a detailed specification of the **TX** and **XA** services. The **AR** service is considered implementation-dependent because RMs may provide services other than database management, and thus the **AR** interface must allow its clients to make use of those services.

Our reference implementation of SDTP was written in Common Lisp. The advantage of using a language quite different from X/Open's choice of C is that it helped us identify and eliminate some of the "C-centric" implementation decisions that crept into the X/Open design. Common Lisp was also attractive because it enabled rapid implementation and debugging of the system. The Common Lisp package system was used to globally identify the various services and prevent name conflicts (e.g., since `open` is a standard Common Lisp function for file I/O as well as appearing in both the **TX** and **XA-xa** service, a name conflict would occur unless some method of distinguishing between these functions was employed).

A remote procedure call (RPC) facility native to the Common Lisp version we used was employed for component-to-component communication. Communication between the RMs and the single-level databases was done by means of a foreign function interface (FFI) to the native database programmatic interface library.

7.1 The TX Service

The **TX** service allows the application to open and close resource managers, start, finish or roll back transactions, obtain information about the state of a transaction, and set certain transaction characteristics.

A typical chain of events involving the application making use of the **TX** service is:

1. Tell the TM to open the RMs with the `tx:open` call.
2. Connect to the RMs and use the **AR** services of the RM for preliminary functions (such as authentication and key exchange).
3. Tell the TM to start a transaction with the `tx:begin` call.
4. Use the **AR** service of the RMs to exchange data
5. Finish the transaction by using the TM's `tx:commit` call; alternatively, abort and roll back the transaction using the `tx:rollback` call.
6. Once the desired set of transactions is completed, tell the TM to close the RMs using the `tx:close` call.

7.2 The XA Service

The **XA** service is used by the TM to communicate with the RMs and by the RMs to communicate with the TM. It is divided into two subservices as described above.

The **xa** subservice is used by the TM to communicate with the RMs to negotiate the two-phase commit protocol. A typical chain of events is:

1. The application informs the TM that it wishes to begin processing transactions (via the **ax** subservice, described below).
2. The TM uses the `xa:open` call to tell the RMs to begin listening for connections from the application.
3. The application tells the TM that it is beginning a transaction.
4. The TM issues `xa:start` calls to the RMs, telling them that a new transaction is being started.
5. After issuing one or more commands to the RMs, the application informs the TM that it wants to commit its current transaction.
6. The TM makes an `xa:prepare` call to each of the RMs. Each RM returns a value indicating that it is ready to commit.
7. The TM makes an `xa:commit` call to each of the RMs, indicating that it should commit its work.

The **ax** subservice is used by the RMs to dynamically register with the TM. Once an RM registers with the TM, it is then managed along with all the other RMs that the TM currently knows about. This feature is intended to permit RMs that are infrequently used to only be managed when they are doing actual work, thus avoiding the necessity for them to engage in the transaction protocol when they aren't processing transactions.

7.3 The AR Service

The **AR** service is the service an RM provides to allow applications to access its shared resources. The X/Open DTP standard does not specify the form this interface must take. It allows the RMs to provide standard interfaces, such as SQL, and/or custom or proprietary interfaces. The SDTP RM reference implementation currently provides a custom interface built on top of SQL.

A typical call to an **AR** service looks like the following:

```
(wire:remote-value
  rm-wire
  (ar:update *tid*
    (encrypt-string "inv_id" common-key)
    (encrypt-string new-inv-id common-key)
    (encrypt-string " " common-key)
    (encrypt-string " " common-key)))
```

where `rm-wire` is the wire or connection the application has to the resource manager, `ar:update` is the actual RPC to the **AR** service, `*tid*` is the application's thread ID and the rest of the arguments are mapped into an `update` SQL statement.

The SDTP reference implementation of the **AR** service uses encryption to secure the contents of its network communications (as do the other services). The current encryption package uses a Diffie-Hellman key exchange protocol and RC4 as its encryption mechanism.

8 Coöperating Law Enforcement Databases

As a test application for the SDTP reference implementation, we built an MLS law enforcement tracking system. This system was inspired by an FBI system called "FOIMS" (Field Office Information Management System), though of course it bears no relationship to the actual working of that system.

Our intent in building this application was to demonstrate multilevel security in a database system that was both distributed and replicated. For this reason we decided to distribute the information for State investigations and replicate the Federal information. The motivation for this was that each State would tend to query and update its own information the vast majority of the time and so it would be more efficient to keep that information on local resource managers and allow remote queries; on the other hand, Federal information would likely be queried by many States and so it would be more efficient to replicate that information across the resource managers belonging to the States.

An example of a table that contains information classified at varying security levels was the Agent Table. This table contained a unique ID number for each agent as well as the agent's name. The ID number was needed at all security levels, both to keep track of agent workloads and to ensure that each investigation had a valid agent assigned to it. However, the agent's name was considered Top Secret and so it was only available when the user making the query was authenticated at the Top Secret security level.

Another example of multilevel tables is the Investigation Tables. Each investigation was classified at a particular security level — even the fact that an investigation was being conducted on a particular individual might itself be sensitive information. For this reason, investigations that might have, say, important political implications were classified at Top Secret, while investigations that had resulted in court cases would ordinarily be public knowledge and therefore Unclassified.

Our demonstration implementation contained resource managers for two States, each of which also contained a replica of the Federal portion of the database. The application was capable of querying, updating and adding information to the relevant tables.

The SDTP reference architecture enforces the simple security property (*no read-up*) and the $\star$-property (*no write-down*) by using a distributed Security Manager layer. These Security Managers each used a set of single-level databases for storage; the combination of the Security Manager layer and the single-level databases produced a virtual multilevel resource manager.

The Security Managers communicated with the single-level databases using the native interface provided by the database. The Security Manager would create a set of communications links to the databases and manage them so as to maintain the security constraints. When an operation was requested by a component, the Security Manager would select the set of links that were appropriate for that operation. For example, when a component requested a read operation, the Security Manager would select the links to databases serving the current security level or lower. A write operation would result in only the current security level's link being selected.

The screen dump in Fig. 6 shows the demonstration instance in operation.

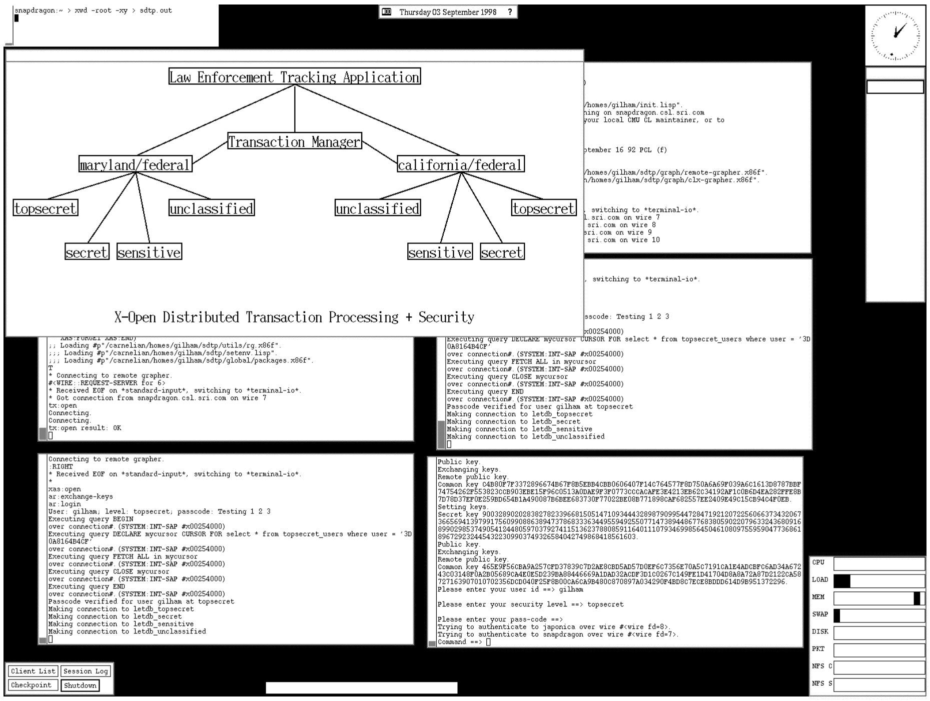

Fig. 6. SDTP Demonstration Instance in Operation

9 Related Work

A great deal of work has been devoted to transformational development of software artifacts and to (informal) architecture refinement, but very little has been done that combines both. Other than our own papers on the subject, some of the best examples are Brinksma, Jonsson, and Orava's work on connector refinement [2], work on component refinement [3], and Philipps and Rumpe's work on refinement of information flow architectures [12]. However, work on relating models of connector behavior at different levels of abstraction by Abowd, Allen, and Garlan [1], Luckham, et al. [5], and Moriconi and Qian [6] is closely related.

Similarly, a great deal of work has been devoted to developing verification of transformations, but we are unaware of any work specifically devoted developing techniques for showing that transformations always introduce faithful interpretations other than our own, with the exception of the sort of very general techniques for proving the existence of faithful interpretations found in references such as Turski and Maibaum's book [16].

Generally speaking, the security community has not been successful in formally linking security proofs to actual implementations. A notable exception is Neely and Freeman's verification of a secure gateway [10, 11]. The technique used in Neely and Freemen's case study is not adequate to deal with the SDTP architecture, because it treats only horizontal refinement (i.e., "bubble decomposition") and not vertical refinement (i.e., change in style of representation).

10 Conclusions

The SDTP case study was performed with several objectives in mind. First, we wanted to determine whether implementation-level SDTP architectural descriptions could be derived for the three implementation approaches defined in our earlier paper [8] by the application of transformations that introduce only faithful interpretations. While we do not have a definitive answer to this question, our experience suggests that such a derivation would be difficult or impossible. Therefore, we attempted to determine whether implementations could be derived using only transformations that always preserve security. The answer to this question is certainly *yes*. But we discovered that defining generally useful transformations that always preserve security can be quite difficult, in some cases. Preservation of security sometimes requires the addition of strong preconditions to the transformation. These strong preconditions can seriously reduce the transformation's generality. We therefore developed an alternative approach which allows transformations that only *sometimes* preserve security to be employed. Rather than building some specific set of sufficient conditions for security-preservation into the transformation, we check whether, in each individual application of the transformation, security has been preserved. Our approach to checking is based on a notion we call *proof-carrying architectures* [14]: the same transformation used to implement the architectural description is used to "implement" the security proof, which is carried along with the architectural description. Using a combination of showing that some transformations always introduce faithful interpretations and checking that the others preserve security in the particular case, we succeeded in manually verifying the security of all 12 implementation-level architectural descriptions (four per implementation approach).

A second objective in performing the case study was to begin to determine whether our "rearchitecting via replay" model works as well for global architectural changes (such as introducing security requirements) as it has worked in the 23 small-scale local rearchitecting case studies we have performed based on the X/Open DTP architecture.[4] As we pointed out above, the X/Open DTP derivation was very effectively reused in the defining the SDTP hierarchies. In addition, our techniques for quickly estimating the cost of changes, and conservatively bounding the scope of changes, required to restore consistency in a hierarchy after a change has been introduced continued to work effectively. Therefore, we have begun the process of designing a rearchitecting tool based on this model that we believe will provide very substantial automated support for the process.

Our third objective in performing the case study was to assess the effectiveness of our transformation verification techniques on a larger stock of transformations. Although we wound up abandoning the idea of using only generally valid transformations in the development of the SDTP hierarchies, all the transformations that we believe to be generally valid were successfully verified. So, as mentioned above, we have begun to explore the process of automating these manual verifications using SRI's PVS verification system.

The final principal objective was to determine whether the lowest level architectural descriptions in the SDTP hierarchy were, in fact, implementation-level. In other words, we wanted to confirm our belief that no significant design decisions would have to be made when turning these low-level descriptions into executable code. This is crucial, because the hierarchy only effectively links the abstract architecture to the actual implementation if the gap between the lowest level description and the code is very small. If the gap is too large, confidence that the "implementation-level" descriptions of SDTP are secure provides only weak evidence that the actual implementation is secure. Our experience was that writing the code from these low-level descriptions is completely straightforward, thanks to formalizing information about implementation language facilities as SADL styles. Although this approach makes the low level implementation decisions in the SDTP hierarchies implementation language-dependent, it should hardly be surprising that choice of a programming language for the implementation affects low-level architectural design. Designs appropriate for other choices of programming language can be included by introducing additional branches in the hierarchies (although we have not done so in the case of the SDTP hierarchies we have developed). It should be noted that all these branchings would be at a level of abstraction lower than the X/Open standard, and hence they do not reduce the general utility of an SDTP standard at that same abstraction level. Based on this experience, we are planning to extend our architecture transformation toolset by including a facility for automatic generation of code for the architecture from an implementation-level description in SADL.

So it can be seen that development of the three SDTP hierarchies and the one reference implementation and demonstration instance served as a useful case study in

– transformational development of architectures,

[4] The 23 studies are based on making a variety of small changes in system requirements (e.g., adding an RM) and/or implementation infrastructure (e.g., reducing the cost of interprocessor communication).

– rearchitecting after adding a new "global" system requirement,
– transformation verification, and
– linking architectural descriptions to code.

References

1. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. Technical Report CMU-CS-95-111, School of Computer Science, Carnegie Mellon University, January 1995.
2. E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In S. Abramsky and T. S. E. Maibaum, editors, *Proceedings of TAPSOFT '91*, pages 71–80. Springer-Verlag, 1991.
3. M. Broy. Compositional refinement of interactive systems. Technical Report No. 89, Digital Systems Research Center, Palo Alto, July 1992.
4. L. J. LaPadula and D. E. Bell. MITRE technical report 2547, Volume II. *Journal of Computer Security*, 4(2,3):239–263, 1996.
5. D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):314–335, April 1995.
6. M. Moriconi and X. Qian. Correctness and composition of software architectures. In *Proceedings 2nd ACM Symposium on Foundations of Software Engineering (SIGSOFT '94)*, pages 164–174. ACM Press, 1994.
7. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995. Available at `http://www.csl.sri.com/sadl/tse95.ps.gz`.
8. M. Moriconi, X. Qian, R. A. Riemenschneider, and L. Gong. Secure software architectures. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 84–93, May 1997. Available at `http://www.csl.sri.com/sadl/sp97.ps.gz`.
9. M. Moriconi and R. A. Riemenschneider. Introduction to SADL 1.0: A language for specifying software architecture hierarchies. Technical Report SRI-CSL-97-01, Computer Science Laboratory, SRI International, March 1997. Available at `http://www.csl.sri.com/sadl/sadl-intro.ps.gz`.
10. R. B. Neely and J. W. Freeman. Structuring systems for formal verification. In *Proccedings of the IEEE Symposium on Security and Privacy*, pages 2–13, April 1985.
11. R. B. Neely, J. W. Freeman, and M. D. Krenzin. Achieving understandable results in a formal design verification. In *Proceedings of the Computer Security Workshop II*, pages 115–124, June 1989.
12. J. Philipps and B. Rumpe. Refinement of information flow architectures. In *Proceedings of the First IEEE International Conference on Formal Engineering Methods (ICFEM '97)*, pages 203–212, November 1997. Available at `http://www4.informatik.tu-muenchen.de/papers/icfem_rumpe_1997_Publication.html`.
13. R. A. Riemenschneider. A simplified method for establishing the correctness of architectural refinements. SRI CSL Dependable System Archiecture Group, Working Paper DSA-97-02. Available at `http://www.csl.sri.com/sadl/simplified.ps.gz`., November 1997.
14. R. A. Riemenschneider. Checking the correctness of architectural transformation steps via proof-carrying architectures. In *Proccedings of the First Working IFIP*

Conference on Software Architecture, Kluwer Academic Press, 1999. Available at `http://www.csl.sri.com/sadl/checking.ps.gz`.
15. R. A. Riemenschneider. Correct transformation rules for incremental development of architecture hierarchies. SRI CSL Dependable System Architecture Group, Working Paper DSA-98-01. Available at `http://www.csl.sri.com/sadl/incremental.ps.gz`. February 1998.
16. W. M. Turski and T. S. E. Maibaum. *The Specification of Computer Programs.* Addison-Wesley, 1987.
17. X/Open Company, Apex Plaza, Forbury Road, Reading, Berkshire RG1 1AX, U.K. *Distributed Transaction Processing: Reference Model*, November 1993.