A Framework for Systematic Synthesis of Transactional Middleware

Apostolos Zarras, Valérie Issarny
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes Cédex, FRANCE
email: {zarras, issarny}@irisa.fr

Abstract
Transactions are contracts that guarantee a consistent, transparent, individual system state transition and their use is widespread in many different kinds of computing systems. Some well known standards (e.g. CORBA) include the specification of services that provide transactional properties. In this paper, we present a formal method for the systematic synthesis of transactional middleware based on the combination of the aforementioned services. The synthesis of transactional middleware is based on (*i*) the formal specification of transactional properties and (*ii*) stub code generation.

Keywords
Atomicity, Consistency, Durability, Isolation, Middleware, Transaction

1 INTRODUCTION

The construction of a large-scale distributed software system is a complex task that involves design decisions relating to different aspects of the system's behavior. In particular, the developer has to address the provision of the system's functional (i.e. algorithmic aspect) and non-functional (i.e. quality of service with which the system's functions are performed) properties. While the provision of functional properties is system-specific, the one of non-functional properties appertains to the underlying Distributed Processing Environment (DPE) and is common to a variety of systems. The concern of improving software quality and reducing the duration of system development has led to a number of standardization efforts that propose specification of middleware services that are common to various systems.

In the framework of the ASTER project * at IRISA/INRIA, we are investigating design and development methods that aid the developer to select the appropriate services and integrate them within the system implementation (Issarny *et al.* 1998). In this paper, we focus on the treatment of non-functional properties relating to transactional processing, as they are among the primary concerns in building large-scale software systems. Transactions provide the warranty for a consistent transparent, individual system state transition (Gray 1981), properties which are indispensable in both centralized and distributed computing systems (e.g. databases, telecommunications, CAD/CAM, operating systems). However, the characteristics of those systems vary a lot and the transaction concept had to adapt with respect to them. As a result, a variety of transaction models (Moss 1982, Pu *et al.* 1988) were developed in order to deal with the newly imposed requirements (e.g. performance, cooperation). Moreover, some well known processing standards (e.g. CORBA (OMG 1995), ODP (ISO/IEC 1994)) include the specification of services that provide transactional properties (e.g. locking, persistence, recovery). However, to ascertain whether or not some required transactional properties are supported by existing standard services is a tedious task. For instance, let us consider the CORBA *Object Transaction Service* (OTS) which provides atomicity but whose support for the nested transaction model is optional in the standard service specification. Then, the developer must be aware of both the variety of standard services installed in the environment and of the particular features provided by their implementations. Furthermore, it is often the case that the property provided by a standard service should be extended in order to meet the current system's requirements. For example, the CORBA *Concurrency Control Service* (CCS) provides locking. In order to obtain the well known two-phase locking protocol, the developer must use the service in a certain way. Finally, there are cases where the non-functional properties provided by standard services should be refined. For instance, the ODP Transaction Function does not define the exact correctness criterion that guarantees a (conflict-)serializable execution. Discharging the developer from the arduous responsibility to manage such details at the low level of implementation is our motive. Towards that goal, we are proposing a systematic procedure that takes as input the transactional requirements for a given system and that takes in charge the selection of corresponding middleware services and their combination to build a middleware platform, customized to the system's needs.

This paper introduces our method for synthesizing transactional middleware from a system description including the specification of transactional non-functional properties. Section 2 gives the core elements of the method. Our method is exemplified

* See URL: http://www.irisa.fr/solidor/work/aster.html.

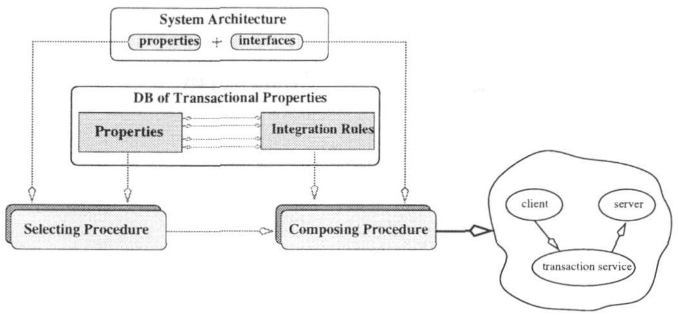

Figure 1 Synthesizing transactional middleware.

in Section 3 in the CORBA framework. Finally, conclusions are given in Section 4, summarizing our contribution and presenting a comparison with related work.

2 SYNTHESIS METHODOLOGY

The basic constituents of our method for synthesizing transactional middleware are (see Figure 1):

- A declarative language, known as *Architecture Description Language* or ADL for short (Shaw *et al.* 1995), for the description of systems software architectures. The ADL allows the developer to describe the gross organization of his software system in terms of the interconnection of functional components (i.e. computation unit or data store) defined by their interface. The ADL further supports the specification of transactional properties either provided or required by software components.
- A *selecting procedure* that takes as input a set of transactional requirements, reflecting the demand for a particular transactional model, and that verifies whether the requirements are supported by existing middleware services.
- A *composing procedure* that combines the middleware services meeting the system's transactional requirements with the system's functional components.

2.1 Specifying Transactional Requirements

The specification of transactional requirements relies on: (*i*) the description of software architectures and (*ii*) the formal specification of transactional properties used for systematic selection of middleware services. We examine these two issues in turn.

Describing system architectures

Using our ADL, the software architecture of a system is described in terms of the *component* and *configuration* abstractions. A *component* abstractly defines a unit of computation or data store in terms of its interface, which gives the operations that are imported and exported by the component. A *configuration* defines a system architecture by stating interconnections among a set of of components, i.e. by binding the imported operations to the exported ones. The description of components and configurations may further embed the specification of required and/or provided transactional properties. To simplify the developer's task, this specification is given in terms of properties names corresponding to formal specifications stored in an expandable database of transactional properties, as detailed in the next paragraph. An example of software architecture description is given below:

```
COMPONENT client {
  IMPORTS: VOID op();
  DEFINES TRANSACTIONAL Trans:
    MODEL = Tmodel;
    BEGIN = t_begin(); COMMIT = t_commit(); ABORT =  t_abort();
  REQUIRES: op: Trans
}
COMPONENT server {
  EXPORTS: VOID op(); REQUIRES: NOTHING;
}
CONFIGURATION system {
  USES: client, server;
  BINDS: client::op: server::op;
}
TRANSACTION_MODEL Tmodel {
  PROPERTIES: atomic AND isolated;
}
```

It is a simple client/server system where the client component requires the invocation of the operation op of the server to be executed within a transaction under the `Tmodel` model. Furthermore, the operations used within the client's source code to perform significant transactional events are `t_begin`, `t_commit`, and `t_abort`. The `Tmodel` transactional model is the conjunction of the `atomic` and `isolated` properties. These two properties may be refined into a number of more specialized ones (e.g. rollback recovery for atomicity and serializability for isolation), and hence a large number of middleware services are *a priori* eligible to meet these requirements.

Components of a configuration declare in their interface, their requirements for transactional processing. Two components involved in a nested control flow at runtime (e.g. a client component instance invokes the operation *op* of a server which invokes another server during the execution of *op*) may require distinct transactional models. In a first step, we address this issue by enforcing the requirement for a single transactional model at the level of a configuration. Given a configuration of components that require distinct transactional models, the developer should specify the model that applies for all the components (although they may not be involved in a nested control flow). By default, the transactional model that is enforced for a configuration is the conjunction of the strongest transactional properties among the specified ones. However, for the sake of flexibility, components that do not specify any transactional requirement are handled in a different way. The selected transactional model will be enforced for the operations they invoke only if the invocations happen within a transaction whose model is specified to be distributed.

Let us notice that our synthesis method applies to static configurations and hence is not directly suited for configurations based on dynamic bindings (e.g. using a trading service). The treatment of dynamic configurations is an open issue for future work.

Specifying transactional properties

For the specification of transactional properties, we use a refinement approach where each of the ACID (Atomicity, Consistency, Isolation, Durability) properties can be gradually refined into more specialized properties. The properties of a given transactional model correspond to the conjunction of one refinement of each of the ACID properties that are enforced by this model. Additionally, the transactional property provided by a given middleware service corresponds to one of the refinements of the ACID properties.

Our specification of transactional properties is based on temporal logic where only the precedence operator among predicates is used in addition to the operators of first order logic. This allows us to keep the notations simple. The following notations are used: $\wedge$, $\vee$, and $\Rightarrow$ denote logical and, or, and implication, respectively; σ (possibly primed or with subscript) denotes a system state; $[\sigma]$ denotes the predicate that holds if the system is in state σ, $[\sigma] \prec [\sigma']$ holds if $[\sigma]$ is verified before $[\sigma']$*. In addition, t (possibly primed or with subscript) denotes a transaction, $pre(t)$ (resp. $post(t)$) denotes the system's state before (resp. after) the execution of transaction t, and $\sigma \rightharpoonup_t \sigma'$ is the specification of t. In that context, a property P *refines* another property P' if $P \Rightarrow P'$.

Let us first examine the specification of the basic ACID properties. A transaction is said to be *atomic* (also known as the *all-or-nothing* property) if it appears to be indivisible. In other words, either the execution of the transaction results in a state transition that goes from the initial state to the final state specified for the transaction or, the system appears as if it had never left the initial state. The following formula captures this property:

$$atomic(t) \quad \equiv \quad ((pre(t) = \sigma) \wedge (\sigma \rightharpoonup_t \sigma')) \Rightarrow ((post(t) = \sigma) \vee (post(t) = \sigma'))$$

The *consistency* property guarantees that given a system in a consistent state and a transaction that executes alone and to completion, the system passes to another consistent state. Hence, for a transaction t and the system states before and after

*Let us also note that we have: $p_1 \prec p_2 \prec \ldots \prec p_n \equiv \bigwedge_{i=1,..,n-1}(p_i \prec p_{i+1})$.

the execution of t, the corresponding system state transition must be dictated by the transaction specification. This property is captured by the following formula:

$$consistent(t) \quad \equiv \quad ((pre(t) = \sigma) \wedge (post(t) = \sigma')) \Rightarrow (\sigma \rightharpoonup_t \sigma')$$

The third basic property is the one that guarantees the *isolated* execution of a transaction, i.e., a transaction t always behaves as if it is the only one executing. Hence, for every transaction t', $t' \neq t$, either t' views a system state that follows the state produced by t or t views a system state that follows the state produced by t'. We have:

$$\begin{aligned} isolated(t) \quad \equiv \quad & ((pre(t) = \sigma) \wedge (post(t) = \sigma')) \Rightarrow (\forall\, t', t' \neq t: \\ & ((pre(t') = \sigma'') \wedge (\sigma' \prec \sigma'')) \vee ((post(t') = \sigma''') \wedge (\sigma''' \prec \sigma))) \end{aligned}$$

The final basic transactional property is the one that guarantees *durability* for the results of completed transactions. This means that the system state that results from the execution of transaction t will survive subsequent failures. We get:

$$durable(t) \quad \equiv \quad (post(t) = \sigma) \Rightarrow (\forall \sigma' : [\sigma] \prec [\sigma'] \Rightarrow \sigma \rightharpoonup_S \sigma')$$

where $\rightharpoonup_S$ denotes a system state transition that is correct with respect to the system's specification.

Refining transactional properties

The above transactional properties should be refined to be actually helpful for the synthesis of transactional middleware. Refined properties then serve to characterize the behavior of various transactional models as well as the behavior of available middleware services. Up to now, we managed to refine our basic properties regarding several well-known models (e.g nested, split/join transactions). A detailed description of those refinements is not given here due to the lack of space. In the following, we focus on the refinement of properties that will be used for illustration purpose. These are the transactional properties provided by the CORBA OTS and CCS which respectively provide support for atomicity and isolation. The refinements of basic transactional properties characterize the behaviors of transactional systems, and hence define the behavior of the operations on transactions, i.e., begin, commit, and abort of a transaction. Furthermore, properties are given with respect to the execution of the DPE's operations. Such executions are characterized using the following base predicates in the case of a distributed transactional DPE:

- EXPORT(C_1, C_2, d) holds if component instance C_1 exports data d towards component instance C_2.
- IMPORT(C_1, C_2, d) holds if component instance C_2 imports data d from component instance C_1.
- FAILURE(C, σ) holds if component instance C fails while the system is in state σ.
- BEGIN(C, t) holds if component instance C requires the initiation of transaction t.

- COMMIT(C, t) holds if component instance C requires the commitment of transaction t.
- ABORT(C, t) holds if component instance C requires the abortion of transaction t.

Let us first give a refinement of the *atomic*(t) property under the distributed flat transaction model where we use the notation σ_C to denote the state of component instance C. This refinement consists of refining the value specified for $pre(t)$ and $post(t)$. We get the following *globalAtomic* property:

$$\mathit{globalAtomic}(t) \quad \equiv \quad ((pre_f(t) = \sigma) \wedge (\sigma \rightharpoonup_t \sigma')) \Rightarrow ((post_f(t) = \sigma) \vee (post_f(t) = \sigma'))$$

with:

$$\begin{array}{lcl}
pre_f(t) & = & \bigcup_{C:begin(C,t)} : (pre(C,t) = \sigma_C) \\
pre(C,t) & = & \sigma_C \equiv [\sigma_C] \wedge begin(C,t) \\
post_f(t) & = & \bigcup_{C:begin(C,t)} : post(C,t) = \sigma_C \\
post(C,t) & = & \sigma_C \equiv ([\sigma'_C] \wedge commit(C,t) \wedge ([\sigma_C] \prec [\sigma'_C])) \vee \\
 & & (pre(C,t) = \sigma_C \wedge abort(C,t))
\end{array}$$

and where:

$$\begin{array}{lcl}
begin(C,t) & \equiv & \text{BEGIN}(C,t) \vee \\
 & & (\exists C' \colon (\text{IMPORT}(C',C,d) \wedge (begin(C',t) \prec \text{IMPORT}(C',C,d)) \wedge \\
 & & (\nexists\ \text{IMPORT}(C'',C,d') \colon \\
 & & \quad (begin(C'',t) \prec \text{IMPORT}(C'',C,d') \prec \text{IMPORT}(C',C,d))) \wedge \\
 & & (\nexists\ abort(C'',t) \colon \\
 & & \quad (begin(C''',t) \prec abort(C'',t) \prec \text{IMPORT}(C',C,d))) \wedge \\
 & & (\nexists\ commit(C'',t) \colon \\
 & & \quad (begin(C''',t) \prec commit(C'',t) \prec \text{IMPORT}(C',C,d)))) \\
commit(C,t) & \equiv & \text{COMMIT}(C,t) \vee \\
 & & ((commit(C',t) \wedge (begin(C',t) \prec commit(C',t))) \wedge \\
 & & (\nexists\ abort(C'',t) \colon (begin(C'',t) \prec abort(C'',t) \prec commit(C',t)))) \\
abort(C,t) & \equiv & \text{ABORT}(C,t) \vee \\
 & & (\text{FAILURE}(C,\sigma_C) \wedge (begin(C,t) \prec [\sigma_C])) \vee \\
 & & (abort(C',t) \wedge (begin(C',t) \prec abort(C',t)))
\end{array}$$

Let us notice here that providing other refinements for the *atomic* property consists of defining the formulas for *begin*, *commit*, and *abort* according to the behavior of the associated actions. Let us now give a refinement of the *isolated* property, we have:

$$\begin{array}{lcl}
\mathit{globalIsolated}(t) & \equiv & \bigwedge_{\forall C:begin(C,t)} \mathit{IsolatedComponent}(C,t) \\
\mathit{isolatedComponent}(C,t) & \equiv & ((\nexists t' \colon begin(C,t) \prec begin(C,t') \prec commit(C,t)) \Rightarrow \\
 & & \quad (post(C,t) = \sigma_C)) \Rightarrow \\
 & & (\forall t'' \colon begin(C,t) \prec begin(C,t'') \prec commit(C,t)) \Rightarrow \\
 & & \quad (post(C,t) = \sigma_C))
\end{array}$$

The above formulas may further be refined to distinguish between the various implementations of mechanisms providing them.

2.2 The Selecting Procedure

Given the proposed description of a configuration and formal specification of transactional properties, selecting middleware services that will be used to enforce the transactional model required for a configuration C, is direct. Let $\{R_i\}_{i=1,\ldots,n}$ be the set of transactional properties characterizing the required transactional model, and S be the set of available middleware services S_j, $j \leq 1 \leq m$, such that each S_j provides the transactional property P_{S_j}. Then, a transactional middleware can be built for C if:

$$\forall R_i, 1 \leq i \leq n : \exists \{S_k\}_{k=1,..,m} \subseteq S | \wedge_{k=1,..,m} P_{S_k} \Rightarrow R_i$$

Selecting middleware services to enforce transactional properties becomes simplified if the developer is provided with a tool that implements systematic retrieval of software according to the above formula. Such a tool can be based on the adequate organization of the database of transactional properties, in a way similar to the work presented in (Mili *et al.* 1994). The database is organized into a lattice structure* that encodes the refinement relation among the formulas defining transactional properties, i.e., any descendant of a property P refines P. Systematic retrieval of middleware services is then based on the following management of the database: *(i)* Upon the availability of a new middleware service, the property it provides is inserted in the database and the service is connected to it; *(ii)* Given the request for a transactional property P, the middleware service that will be returned among those available in the database is –if any– the one that provides a property P' that refines P such that there is no service providing P'' where P' refines P'' refines P. Construction of the database together with the selection of a middleware service with respect to a transactional property requires a tool based on theorem proving technology. There exist various such tools that have been exploited successfully for the retrieval of software components with respect to functional properties expressed in first order logic (Mili *et al.* 1994, Zaremski and Wing 1995, Schumann and Fischer 1997). We further have implemented our own in the ASTER project for the retrieval of software components providing non-functional properties, still expressed in first order logic (Issarny *et al.* 1998). Up to now, we have not yet experimented the use of a tool that deals with transactional properties based on temporal logic. However, we do not see this as a major difficulty since this has already been addressed for richer logics such as real-time temporal logic (Blair *et al.* 1997).

Following the selection of middleware services for a given configuration, a functioning configuration is obtained by composing the configuration's functional components with the selected services. This raises the issue of components and services compatibility with respect to the underlying communication protocol (known as the *architectural matching* problem (Garlan *et al.* 1994)). In the proposed method for transactional middleware synthesis, architectural matching is simplified by request-

*More precisely, there is a lattice structure for each of the basic ACID properties.

ing the developer to specify, for each component, the middleware platform (e.g. CORBA) for which it is targeted.

2.3 The Composing Procedure

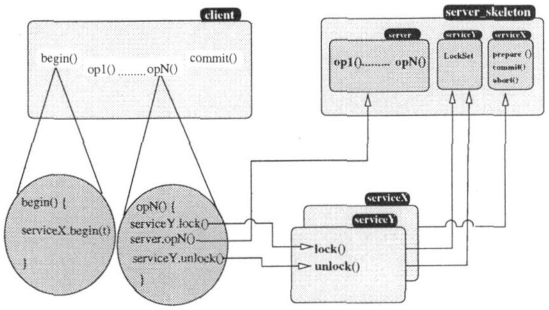

Figure 2 A client/server system configuration and its transactional middleware.

Given a configuration description and the set of selected services for the configuration, the final step of middleware synthesis is the composing procedure. The goal is to generate stub code for both the components that initiate transactions and/or issue requests within a transaction (referred to as client components in the following) and the components that serve requests issued within a transaction (referred to as server components in the following). The produced stub code integrates the configuration's functional components and the underlying middleware services in a full functioning system (see Figure 2). Stub code generation relies on the definition of *integration rules* for each transactional middleware service. The notion of integration rule completes the definition of transactional properties provided by a service. Basically, integration rules specify the code that needs to be added to client and server components. The integration rule for server components amounts to state the piece of code that implements the targeted transactional feature. From the client side, the integration rules subdivide into rules for the following actions: creation (i.e. *begin rule*), validation (i.e. *commit rule*), abortion (i.e. *abort rule*), and operation calls. Rules relating to operation calls subdivide into rules that specify the code to be executed before the call (i.e. *pre_call rule*), when the called is issued (i.e. *op_stub rule*), and after the call returns (i.e. *post_call rule*).

For illustration, the following figure gives the integration rules for a service, called `lock`, providing the isolation property according to the *two-phase locking* algorithm.

```
INTEGRATION lock ::=
 SERVER RULES:
   FORALL(S: SERVER) SKELETON = S ⊎ (lock::LockSet lockSet); END
 CLIENT RULES:
   CONTEXT BEGIN() ::=
     CONTEXT ctx; RETURN ctx;
   COMMIT(CONTEXT ctx) ::=
     FORALL(srv IN ctx) srv.lockSet.unlock(); END
   ABORT(CONTEXT ctx) ::=
     FORALL(srv IN ctx) srv.lockSet.unlock(); END
   PRE_CALL(CONTEXT ctx) ::= SKELETON srv; srv = BINDING(C, op);
     srv.lockSet.lock(); ctx.insert(srv);
   POST_CALL(CONTEXT ctx) ::=
     NULL
   OP_STUB(..., CONTEXT ctx)::=
     SKELETON srv; srv = BINDING(C, op); srv.op(..., ctx);
```

In the figure, words in capital letters denote keywords of the language used for the specification of integration rules. In particular, `SERVER` denotes server components of the configuration, i.e., components that export operations which are bound in the configuration; `SKELETON` denotes the skeleton to be produced for a server component; `CONTEXT` denotes the data structure that characterizes the context of a transaction; and `BINDING(C, op)` returns the instance of the server component that is bound to the imported operation op of client component C, in the configuration that is processed. The integration rules given for the `lock` service have the following meaning. The one for server components is trivial: components import features of the service using aggregation, which is specified by the symbol $\uplus$. The rules for client components consist of acquiring a lock on each server that is invoked within a transaction, passing the transaction context on each call to a server, and releasing locks upon the transaction termination.

Given the integration rules provided for middleware services, the stub code for the components of a given configuration is generated by appending, for each type of integration rules, all the corresponding rules of the selected services. In addition, the execution of the stub code generated for client components must preserve possible existing precedence constraints among middleware services (e.g. releasing locks is done after committing the produced results). Hence, the combination of integration rules relevant to client components is accomplished by employing a topological order graph algorithm (Cormen *et al.* 1990). Possible precedence constraints are set upon the insertion of a new service in the service repository. They are defined regarding all possible related services. This last issue gives the impression of an intolerable administration effort. Even so, we must consider that transactional services can be well classified in certain categories and dependencies are only formed between different classes of services providing different transactional properties. For instance, there exist three different classes of concurrency control services providing isolation, namely, serializability graph testing, locking, and timestamp order preserving services. For each one of them, there are different precedence constraints regarding failure recovery services providing atomicity, which can also be categorized in certain classes (e.g. undo, redo, ...).

3 EXEMPLIFYING MIDDLEWARE SYNTHESIS WITH CORBA

This section details the use of the proposed synthesis method through the example of a software system based on CORBA. The example is the same one that goes along with most of the CORBA transactional services implementations. It is a traditional client/server banking system that consists of a number of distributed server components instances (or *objects* using CORBA terminology) associated with a database. Transactional clients issue request operations to the server components for reading and updating the system's state. In order to simplify the example and without loss of generality, we assume that each server component corresponds to a single bank account, which provides operations for withdrawal and deposit. In such a system, it is essential to perform compound operations (e.g. transferring an amount from one account to another is realized as a withdrawal followed by a deposit) within a transaction, in order to preserve the system's consistency. Hence, the traditional ACID properties are required for distributed transactions. The underlying database provides *durability* (i.e. a recovery protocol) and *local atomicity* (i.e. atomic operations) Furthermore guaranteeing *consistency* is system-specific, the remaining requirements are those for *global atomicity* and *isolation*. The following is the ADL description of the system's architecture:

```
COMPONENT BankClient {
  SOURCE: xxx/BANKclient_i.cc; PLATFORM: MIDDLEWARE = CORBA; LANGUAGE = C++;
  EXCEPTIONS: InsufficientFunds {};
  IMPORTS: VOID deposit(IN FLOAT balance);
    VOID withdraw(IN FLOAT balance) RAISES (InsufficientFunds);
  DEFINES TRANSACTIONAL Trans:
    MODEL = BankTrans; BEGIN = t_begin(); COMMIT = t_commit(); ABORT =  t_abort();
  REQUIRES: withdraw, deposit : Trans; }
COMPONENT BankServer {
  SOURCE: xxx/BANKserver_i.cc; PLATFORM: MIDDLEWARE = CORBA; LANGUAGE = C++;
  EXCEPTIONS: InsufficientFunds {};
  EXPORTS: VOID deposit(IN FLOAT balance);
    VOID withdraw(IN FLOAT balance) RAISES (InsufficientFunds);
  REQUIRES: NOTHING; }
CONFIGURATION BankSystem {
  USES BankClient, BankServer;
  BINDS BankClient::withdraw: BankServer::withdraw;
     BankClient::deposit: BankServer::deposit; }
TRANSACTION_MODEL BankTrans {PROPERTIES: globalAtomic AND globalIsolated; }
```

Clients import the `deposit()` and `withdraw()` operations which are exported by the server components. In addition, clients require the aforementioned operations to be transactional according to the transaction model defined as the conjunction of the *globalAtomic* and *globalIsolated* properties. Finally, clients specify that the initiation, validation and abortion of a transaction are notified in the client source code through the `t_begin()`, `t_commit()`, and `t_abort()` pseudo-operations, respectively. The source code for these operations is generated during the composing procedure, together with the stub code for the `deposit()` and `withdraw()` operations.

Middleware Synthesis

Two standard CORBA Common Object Services can be used to synthesize a middleware that conforms to the banking system requirements. The OTS service (chapter 10 in (OMG 1995)) provides the *global atomicity* property through a distributed commitment protocol, while the CCS service (chapter 7 in (OMG 1995)) is associated with two-phase and well-formed locking. Given the behavior of two-phase and well-formed locking, it can be formally proved that they imply an isolated execution and hence the *isolatedConfig* property. It follows that the CCS and OTS services can be selected using our method exposed in Subsection 2.2, to meet the banking system's transactional requirements. In order to use these services, including the appropriate extension of the locking functionality provided by the CCS service, the corresponding integration rules must be followed.

The CCS service exports the `LockSet` CORBA IDL interface that provides the `lock()` and the `unlock()` operations. The former operation acquires a lock on the corresponding `LockSet`. If the lock is already held in an incompatible mode by another client, the current operation is blocked until the lock is dropped, using the latter operation. The integration rules that correspond to the CCS service are then the ones given below:

```
INTEGRATION CosConcurrencyControl ::=
  SERVER RULES:
    FORALL(S: SERVER)
       SKELETON = S ⊎ (CosConcurrencyControl::LockSet lockSet); END
  CLIENT RULES:
    CONTEXT BEGIN() ::= CONTEXT ctx; RETURN ctx;
    COMMIT(CONTEXT ctx) ::= FORALL(srv IN ctx) srv.lockSet.unlock(); END
    ABORT(CONTEXT ctx) ::= FORALL(srv IN ctx) src.lockSet.unlock(); END
    PRE_CALL(CONTEXT ctx) ::= SKELETON srv; srv = BINDING(C, op);
      srv.lockSet.lock(); ctx.insert(srv);
    POST_CALL(CONTEXT ctx) ::= NULL;
    OP_STUB(..., CONTEXT ctx) ::=
       SKELETON srv; srv = BINDING(C, op); srv.op(..., ctx);
```

The OTS service provides the `Current` CORBA IDL interface that includes operations allowing a client to initiate (i.e. `begin()`) and complete transactions (i.e. `commit()` and `rollback()`). Moreover, for a server component to be transactional (i.e. the server's ability to retrieve information about the transaction that is associated to an operation being served), it must inherit from the `TransactionalObject` interface, also provided by OTS. The resulting integration rules for OTS are given below:

```
INTEGRATION CosTransactions ::=
  SERVER RULES:
    FORALL(S: SERVER)
       SKELETON = S ◁ CosTransactions::TransactionalObject; END
  CLIENT RULES:
    CONTEXT BEGIN() ::=
       CONTEXT ctx; CosTransactions::Current::begin(); RETURN ctx;
    COMMIT(CONTEXT ctx) ::= CosTransactions::Current::commit();
    ABORT(CONTEXT ctx) ::= CosTransactions::Current::rollback();
    PRE_CALL(CONTEXT ctx) ::= NULL;
    POST_CALL(CONTEXT ctx) ::= NULL;
    OP_STUB(..., CONTEXT ctx) ::=
      SKELETON srv; srv = BINDING(C, op); srv.op(..., ctx);
```

The combination of the integration rules of CCS and OTS given for client components further adheres to the precedence constraint that must be preserved whenever combining a locking service with a service that provides atomicity: locks should be released only after the results produced by a transaction become permanent. After combination of the integration rules, the concluding step of the middleware synthesis is to interpret the rules in stub code, regarding the banking system's architecture. The tool used for achieving interpretation is platform specific and a more detailed description of it can be found in (Zarras and Issarny 1998).

4 CONCLUSION

This paper has presented a method for the systematic synthesis of transactional middleware based on the use of existing standardized services. The main contribution of our work comes from combining the following two points:

(*i*) Synthesis of transactional middleware based on an *open* repository of standard services associated to an *expandable* set of properties.

(*ii*) The whole procedure is *systematic* and *adapts* the primitive properties provided by the standard services.

Let us further point out our contribution through a comparison with related work. The VENARI/ML (Nettles and Wing 1991) environment decomposes the transaction concept in a set of properties realized by object classes offered by the environment. The developer specifies application objects to be atomic, durable etc. However, the limitations existing in such kinds of environments are: the non-expandable property set and the static association between objects implementations and properties (i.e. an object cannot be used both as transactional and non-transactional, because its properties are defined at compile time). Additionally, there is no automation in composing an application that conforms to certain transactional requirements.

In the same spirit of decomposing the transaction concept, we meet the RAVEN system (Finkelstein *et al.* 1994) and the approach proposed in (Ranson 1995) for the TINA architecture. The novelty is that properties are dynamically associated with the objects. In RAVEN, this is done through inheritance and object migration while in the other proposal, the properties are part of the request message issued to a particular object.

An approach that overcomes the closed property set limitation is the combination of ACTA (Chrysanthis and Ramamritham 1994) and ASSET (Biliris *et al.* 1994). ACTA is a formal method for the specification of transaction models and ASSET is a programming library that realizes the features provided by the former (e.g. provides routines to create dependencies between transactions). Nevertheless, implementing a model that is specified in ACTA, through the use of ASSET is up to the developer.

An approach that overcomes the closed property set limitation and also provides automation in building the actual transaction model, is presented in (Georgakopoulos and Hornick 1996). According to that, the developer can require a particular transaction model, expressed in an ACTA-like language. A formal framework is used to verify whether or not a set of application objects support this model. In case the model is not supported, a Transaction Management Mechanism (TMM) can be used to enforce the required constraints. Nevertheless the limitation is the non-expandable set of services. The whole framework is based on the TMM unit. Thus, it lacks the flexibility of using primitive transactional services and exploiting the different features they provide (e.g. optimistic *vs.* pessimistic concurrency control).

The synthesis method presented in this paper aims at overcoming the limitations of the aforementioned approaches. Our goal is the synthesis of transactional middleware based on an open service repository associated to an expandable set of properties, that serves as the basic formal proving theory. Finally let us point out the benefit of our method from the standpoint of software correctness due to the use of formal specification, and from the standpoint of software reuse. In order to better point out the applicability of our framework we plan to examine various case studies including different transaction models and services that come with different distributed programming environments. Finally in the context of our work we wish to provide the specification and implementation of flexible services that can easily adapt to various transaction models.

We are currently working on the integration of the proposed synthesis method in the ASTER development environment based on the software architecture paradigm (Issarny *et al.* 1998). The current version of the ASTER environment supports systematic customization of the ORBIX ORB for enforcing basic non-functional properties, which may be specified using first order logic. Integration of our synthesis method in ASTER requires to address selection of middleware components with respect to properties specified in temporal logic. Another enrichment to be made relates to the automated generation of stub code using integration rules. The current ASTER environment realizes component integration by requiring middleware services to implement a given set of appropriate operations for their combination with other components. On the other hand, the procedure for the combination of services defined by our method provides a way to remove these constraints, and may further be exploited for the synthesis of middleware providing non-functional properties other than transactional. Our final goal would be to provide a uniform approach for building middleware regarding the needs for several kinds of properties. Hence we intend to explore the combination of services that provide different types of properties (e.g transactional, fault tolerance, security) and identify ways to deal with cases where those properties interfere.

REFERENCES

Blair G.S., Blair L. and Stefani J. (1997) A specification architecture for multimedia systems in open distributed processing. *Computer Networks and* ISDN *Systems*, 29.

Biliris A., Dar S., Gehani N., Jagadish H. V. and Ramamritham K. (1994) ASSET: A System Supporting Extended Transactions. *Proceedings of ACM-SIGMOD International Conference on Management of Data*, 44-54.

Cormen T., Leiserson C. and Rivest R. (1990) *Introduction to Algorithms*, MIT Press.

Chrysanthis P.K. and Ramamritham K. (1994) Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3), 450-491.

Finkelstein D., Acton D., Coatta T., Hutchinson N. and Neufeld G. (1994) Object Properties in the RAVEN System. *Proceedings of the 14th International Conference on Distributed Computing Systems*, 502-509.

Garlan D., Allen R. and Ocklerbloom J. (1994) Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 17-26.

Georgakopoulos D. and Hornick M. (1996) Customizing Transaction Models and Mechanisms in a Programmable Environment Supporting Reliable Workflow Automation. *IEEE Transactions on Knowledge and Data Engineering*, 8(4), 630-649.

Gray J. (1981) The Transaction Concept: Virtues and Limitations. *Proceedings of the 7th International Conference in VLDB*, 144-154.

Issarny V., Bidan C. and Saridakis T. (1998) Achieving Middleware Customization in a Configuration-based Development Environment: Experience with the ASTER Prototype. *Proceedings of the International Conference on Configurable Distributed Systems*, 207-214.

ISO/IEC (1994) Reference Model of Open Distributed Processing. ISO/IEC Document 10746.

Mili A., Mili R. and Mittermeir R. (1994) Storing and Retrieving Software Components: A Refinement Based System. *Proceedings of the 16th International Conference on Software Engineering*, 91-100.

Moss E. (1982) Nested Transactions and Reliable Distributed Computing. *Proceedings of the International Conference on Reliability in Distributed Software and Database Systems*, 33-39.

Nettles S. and Wing J.M. (1991) Persistence + Undoability = Transactions, School of Computer Science, Carnegie Mellon University, CMU-CS-91-173.

OMG (1995) CORBAservices : Common Object Services Specification. OMG Document.

Pu C., Kaiser G. and Hutchinson N. (1988) Split-Transactions for Open-Ended Activities. *Proceedings of the 14th International Conference in VLDB*, 26-37.

Ranson R.D. (1995) Less-Than-Transactional Semantics for TINA. *Proceedings of TINA'95*, 2, 243-257.

Schumann J. and Fischer B. (1997) NORA/HAMMR: Making Deduction-based Software Component Retrieval Practical. *Proceedings of the International Conference on Automated Software Engineering*, 246-254.

Shaw M., DeLine R., Klein D., Ross T., Young D. and Zelesnik G. (1995) Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4), 314-335.

Zaremski A.M. and Wing J.M. (1995) Specification Matching of Software Components. *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, 6-17.

Zarras A. and Issarny V. (1998) Automated Synthesis of Middleware for Distributed Transactional Applications. *Proceedings of the ACM SIGOPS European Workshop.*

5 BIOGRAPHY

Apostolos Zarras was born at Ioannina, Greece. He received his B.Sc. and M.Sc. degrees from the University of Crete, Hellas in 1994 and 1996 respectively. He is currently working at IRISA/INRIA (ASTER project), while pursuing a Ph.D. at the University of Rennes I. His current research interests include, software architectures, computer aided software engineering and computer architectures.

Valerie Issarny is an INRIA Researcher at IRISA (Rennes, France) since 1993. She obtained her Ph.D. thesis from the University of Rennes I in 1991, on concurrent exception handling. She spent a one-year post-doctoral at the University of Washington in 1992 where she worked on the design of a single address space operating system. She is in charge of the ASTER research project that addresses the construction of distributed systems using the software architecture paradigm.