Imposing Transactional Properties on Distributed Software Architectures

Apostolos ZARRAS and Valérie ISSARNY
{zarras | issarny}@irisa.fr
IRISA - Campus de Beaulieu, 35042 Rennes Cedex, FRANCE

Abstract

Transactions provide the warranty for a consistent, transparent and individual system state transition. In this paper, we describe a method for imposing transactional properties on distributed software architectures. Given the architectural description of a software system, expressed in terms of components, configuration and transactional requirements, a number of basic services are retrieved from a software repository. Selected services are then combined with the components of the system in a fully functioning system. Service retrieval is based on the formal specification of transaction models and basic properties provided by the lower level services (e.g. locking, recovery, time-stamps). Services are also associated with some integration rules, which serve two different purposes: *1)* they guide a code generation procedure that integrates services into the given configuration and, *2)* they extend the basic properties provided by the services in order to meet the required transaction model.

1 Introduction

Separation of concerns and software architecture description were identified as a possible solution for improving software quality and productivity. More precisely, separating the design objectives into those that are specific to a particular computing system (e.g. algorithmic aspects), and those that might be common in many different types of computing systems, simplifies the developer's effort. In particular, the system developer is occupied with the fulfillment of the former, while for the latter, existing software can be reused. Architecture Description Languages (ADLs) provide a clear way to separate concerns. Architectures are described in terms of components, services, and configurations. Components are architecture-specific while services provide certain non-functional properties frequently required for the development of different kinds of computing systems. Providing a systematic way to identify appropriate services that guarantee the initial requirements further simplifies software development.

In this paper, we focus on the demand for transactional processing, because we consider transactions to be among the primary concerns in building large scale distributed computing systems (e.g. databases, telecommunications, CAD/CAM, operating systems). Several different transaction models have been developed in order to meet the requirements of the various types of computing systems [9, 6]. However, all of them could be realized over a common basis of reusable services that provide some primitive properties met in all different models. Several distributed programming environments [7, 15, 4] offer services for concurrency control, failure recovery, stable storage. Moreover, some well-known processing standards include the specification of such primitive services [13, 11] that can be combined in order to realize some required transaction model.

Summarizing the discussion so far, the developer's work is to identify the system's design objectives, and separate them into those that are architecture-specific and those related to the required transaction model. Furthermore, he must select a set of primitive services that can be used for the realization of the model. However, selecting and using primitive services is often a tedious and complicated task. Take for instance the CORBA Object Transaction Service ([11] ch.10), where support for a nested transactional structure is optional in the service standard specification. Hence, the developer has to be aware of both, the available services installed in his system and their particular features. Sometimes, there is also the case where the basic property provided by the service should be extended so as to meet the current requirements. For example, consider the CORBA Concurrency Control service ([11] ch.7), which provides locking. In order to guarantee a serializable execution, the locking protocol must be two-phased and well-formed. Consequently, the developer must use the service in a certain way in order to impose the previously mentioned properties.

Dispensing the developer from such arduous responsibilities is our motive. Our objective is the development of a framework that systematically selects services providing some primitive transactional properties and integrates them into a given configuration, resulting in a fully functioning system. Our proposal was firstly presented in [16] and an overview of the basic concepts is given in Section 2. A more detailed description of the integration of services into a given configuration is presented in Section 3. In Section 4, we provide a simple example that shows the use of our framework. The example is based on services of the ARJUNA platform (a similar example based on the CORBA platform is given in [16]). Finally, in Section 5, we point out our contribution and we compare our framework with related work.

2 Overview

The basic constituents of the proposed framework for the synthesis of transactional middleware are depicted in Figure 1 and further detailed hereafter.

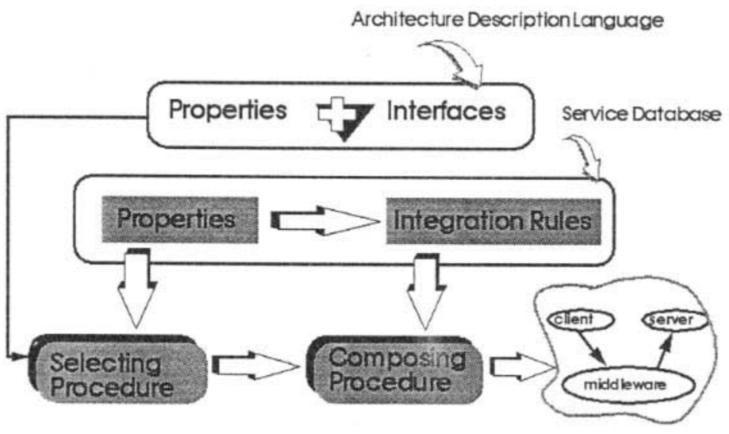

Figure 1: Synthesizing Transactional Middleware

The Architecture Description Language (ADL) allows to specify the structure of a given system in terms of *components, configuration, and properties* required or provided by the system's components. A *component* abstractly defines a unit of computation or data store, in terms of its interface. A component interface contains the operations *provided (i.e. operations that can be used by other components)* and *required (i.e operations bound to operations provided by other components)* by the component. The set of bindings among provided and required operations defines the configuration of the system. Furthermore, the system's ADL description includes the specification of transactional properties, required or provided by the system. The ADL is further detailed in [16] where we proposed our framework.

The Service Database contains information about all the existing services installed in the developer's environment. Every such service is connected to a set of properties, which are expressed as formulas based on temporal logic (again more details can be found in [16]). Each property comes along with a set of guidelines, which must be followed in order for the corresponding property to hold, during the lifetime of a system. These guidelines basically point out how to combine the selected services with the components of the system, so as to impose the properties that are associated to them.

The Selecting Procedure accepts a set of transactional requirements and verifies whether they are supported by existing services. Given that the requirements are expressed in temporal logic the selecting procedure works just like a theorem prover. Considering the requirements, $\{R_i\}_{i=1,..,n}$, as a theorem, and the properties P_{S_j} of the services S_j, $j \leq 1 \leq m$, stored in the database, as existing theory, the following formula must hold in order to verify that the given theorem holds:

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

In the context of the ASTER framework [5] we implemented a tool for efficient software retrieval of reusable services, based on the formal specification of their non-functional properties [14]. This idea is related to the one proposed in [8] where abstract properties (e.g. isolation) are refined into more specific ones (e.g. two-phase-locking) and the whole refinement relation is enclosed in a lattice structure used during the selection. The current ASTER selecting tool is based on first order predicate logic but we do not see any significant difficulties in extending it so as to support a richer logic (i.e. temporal logic).

The Composing Procedure takes over after the selection of the services that meet the system's requirements so as to integrate them into a given configuration. The whole composing procedure is based on the notion of *integration rules.* The integration rules *abstractly define the code to be added to the components in order for them to obtain certain features provided by the services.* The integration rules can be divided in a number of actions that must take place before and after the execution of an operation and during the initiation and the termination of a particular transaction. Hence, the integration rules are used to specify those actions, regarding the functionalities provided by existing middleware services. The composing procedure generates the corresponding source code that implements a number of predefined hooks. These hooks are used by the system developer and hide from him any details related to the underlying middleware. Thus, the responsibility for service integration passes from the developer to the service repository administrator who is supposed to give the integration rules for each newly installed service.

In order to further promote service reuse, the integration rules should be expressed in a way independent from any low level implementation details. For that reason, we decided to adopt an abstract specification language for the definition of the integration rules. Hence, the rules become reusable abstractions that can be easily interpreted while taking into account several implementation details, specific to the given computing system (e.g. implementation language, underlying communication platform etc.). Since in our execution model, components interact only through their interfaces, our specification language ends up to be quite simple. Its basic constituents are: declarations of components' interface instances, and operation calls on components' interface instances.

3 Design

In this section we detail the basic design concepts relating to the framework's composing procedure.

3.1 Composing Steps

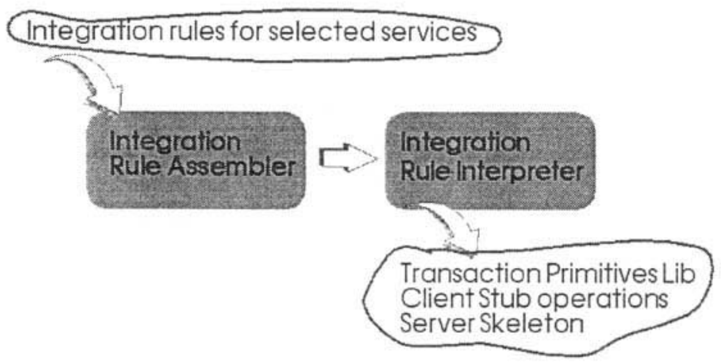
Integration rules for selected services
Integration Rule Assembler
Integration Rule Interpreter
Transaction Primitives Lib
Client Stub operations
Server Skeleton

Figure 2: Composing Procedure

Figure 2 depicts the two basic steps of the composing procedure. During the first step, the *Integration Rule Assembler* assembles the rules of all the selected services and deals with problems of interference and precedence order among them. The former issue raises in the case of using the composing procedure in the general context of a framework, like ASTER [5], that deals with several different kinds of properties (e.g fault tolerance, security, transactions). The latter issue appears often in the current case study of synthesizing transactional middleware. Take for instance the combination of a service that provides locking with a service that provides atomicity. In this case, held locks must be released after the validation of the results produced by the transaction. If a service that provides time-stamps is selected instead of the locking one, timestamp validation must take place before the validation of the results produced by the transaction. During the second step, the *Integration Rule Interpreter* translates the assembled set of integration rules, resulting in the stub code that contains the hooks, which combine the services with the components of the system.

3.2 Integration Rules

In the general case, we consider that integration rules are subdivided into the following:

- Rules for the components that require operations (denoted by the keyword **exportRules**). Export rules are further subdivided into:
  - Rules that define the actions that take place before issuing a request, denoted by the preExport keyword (e.g pack a platform specific request).
  - Rules that define the actions that take place in order to issue a request, denoted by the Export keyword. In the common case, this is the piece of code that calls a primitive action provided by the underlying communication platform so as to issue a request.
  - Rules that define the actions that take place after a request is issued, denoted by the **postExport** keyword (e.g unpack the returned results).
  - Rules that define the actions that take place once and only once, during the component initialization (denoted by the **initExportRules** keyword).
- Rules for the components that provide operations (denoted by the keyword **importRules**) which are further subdivided into :
  - Rules that extend the interface of the components. In the common case, components' interfaces might inherit from an interface (denoted by the **inheritFrom** keyword), or they might contain an instance of an interface (denoted by the **includes**), provided by a middleware service. In that way, components inherit or include some basic features provided by the services.
  - Rules that define the actions that take place before and after serving a request (denoted by the **preImport**, **postImport** keywords respectively).
  - Rules that define the actions performed once and only once, during the component initialization (denoted by the **initImportRules** keyword).
- Rules that define the transaction model's significant actions (denoted by the **transRules** keyword), which are further subdivided into initiation and termination actions (denoted by **beginTrans**, **endTrans** keywords respectively).

At this point, let us notice that in the case of existing services requiring the use of other services the integration rules can still be applied. However this is feasible only in the ideal case where the source code of the former services is available.

4 Example

In order to exemplify our approach, we describe the synthesis of a simple software system based on services provided by the ARJUNA [7] platform. The system follows the client/server execution model and realizes a simple distributed appointment system. Server components provide operations that can be used to add and remove an appointment, and an operation that returns the current time schedule. In such a system, it is reasonable to require a set of operations to be atomic and to further guarantee the isolated execution of clients that manipulate shared time schedules. Finally, it would be convenient to ensure that changes made to the time schedules will survive subsequent system failures. Hence, the system's requirements are: *atomicity, isolation* and *durability.*

The ARJUNA platform offers three basic services, namely `AtomicAction`, `LockManager` and `StateManager`, which provide the aforementioned properties. The first service provides ways to `Begin`, `Commit`, `Abort` a transaction, and upon transaction termination, it performs a two-phase-commit protocol. In Figure 3(a), we give the integration rules for that service. Basically, the

```
integration AtomicAction ::=
importRules ::=
    xInterface := null;
    preImport ::= null;



    Import ::= null;
    postImport ::= null;
exportRules ::=
    preExport ::= null;
    postExport ::= null;
    Export ::= null;
transRules ::=
    beginTrans begin ::=
      component transaction ofType AtomicAction;
      export Begin to transaction;
    endTrans commit ::= export Commit to transaction;
    endTrans abort ::= export Abort to transaction;
```

(a) Integration Rules for the AtomicAction

```
integration LockManager ::=
importRules ::=
  xInterface := inheritsFrom LockManager;
  preImport ::=
    component lock ofType Lock;
    export new to lock;
    export setlock to this with lock;
    Import ::= null;
  postImport ::= null;
exportRules ::=
  preExport ::= null;
  postExport ::= null;
  Export ::= null;
transRules ::=
  beginTrans begin ::= null;


  endTrans commit ::= null
  endTrans abort ::= null;
```

(b) Integration Rules for the LockManager

```
integration AtomicAction ::=
importRules ::=
        xInterface := inheritsFrom StateManager;
        preImport ::= null;
        postImport ::= null;
exportRules ::=
        preExport ::= null;
        postExport ::= null;
        Export ::= null;
transRules ::=
        beginTrans ::= null;
        endTrans ::= null;
```

(c) Integration Rules for the StateManager

Figure 3: Integration rules for the distributed appointment system

integration rules define the transaction model's initiation and termination actions. For example, to begin a transaction, an interface instance of the component AtomicAction must be declared and the Begin operation must be called on this instance. The second service provides ways to acquire locks, using the setlock method, while it further guarantees that locks are acquired and released in a two-phased manner. The integration rules are given in Figure 3(b). According to them, components that export operations must inherit from the LockManager class. A component instance (denoted by this) must call the setlock operation just before serving an incoming request. Finally, the third service ensures that the server components are recoverable. The rules are quite simple (see Figure 3(c)), since the only guideline is that components that export operations should inherit from the StateManager class. To integrate the system's components with the ARJUNA services, the RPC communication platform should be used. The ARJUNA platform provides ways to pack and unpack RPC requests meaning that the corresponding actions should be included in the integration rules (in the definition of the preExport, postExport and Export actions). However, these rules are omitted due to the lack of space.

5 Conclusion

In conclusion, let us relate our proposal with some considerable early work, in order to point out our contribution. We begin with programming environments like VENARI/ML [10], where the transaction concept is decomposed in a set of properties realized by object classes offered by the environment. The developer specifies application objects to be atomic, durable etc. Objects that adopt all the provided properties are fully transactional. The basic limitations are that the property set is not expandable and that the application objects are statically associated with the properties (i.e. an object of the application cannot be used both as transactional and non-transactional, because its properties are defined at compile time). In the same spirit of decomposing the transaction concept, we meet the RAVEN system[2] and the approach proposed in [12]. The novelty is that properties are dynamically associated with the application objects. However, the property set remains closed and closely related to the underlying system (RAVEN and HERMES/ST respectively). An interesting approach that overcomes the closed property set limitation is presented in [3]. According to that approach, the developer can require a particular transaction model, expressed in an ACTA like language [1]. 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 method we introduced in this paper, aims to overcome the limitations of the aforementioned approaches. Our goal is the synthesis of a transactional middleware platform based on an open service repository associated to an expandable set of properties, that serves as the basic formal proving theory. Our method is based on: *1)* the formal specification of transactional properties, *2)* pure logical reasoning and *3)* stub code generation.

Our work is part of ASTER [1], a framework that deals with the specification and automated composition of software systems, regarding their requirements for quality of service (e.g. security, availability, transactions). The use of such a framework promotes software reuse, makes the verification of the resulted system a straightforward task, and finally eases the way to handle the system's evolution over time. In this work, we contribute by providing the basis for dealing with transactional non-functional properties.

Acknowledgments. We would like to thank Petr Tuma for his remarks and his contribution to the realization of the framework prototype.

References

[1] P.K. Chrysanthis and K. Ramamritham. Synthesis of Extended Transaction Models Using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, September 1994.

[1]ASTER URL: `http://www.irisa.fr/solidor/work/aster.html`

[2] D. Finkelstein, D. Acton, T. Coatta, N. Hutchinson, and G. Neufeld. Object Properties in the RAVEN System. In *Proceedings of the 14th Conference on Distributed Computing Systems*, pages 502–509. IEEE, June 1994.

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

[4] IONA. *Orbix Advanced Programers Manual.* IONA Technologies Ltd, 1 edition, July 1995.

[5] Valérie Issarny, Christophe Bidan, and Titos Saridakis. Achieving Middleware Customization in a Configuration-Based Development Environment: Experience with the ASTER Prototype. In *Proceedings of the 4th International Conference on Configurable Distributed Systems.* IEEE, May 1998.

[6] A. K. Elmagarmid. *Database Transaction Models for Advanced Applications.* Morgan Kaufmann, San Mateo, California, 1992.

[7] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of the ARJUNA Distributed Programming System . *IEEE Software*, pages 66–73, January 1991.

[8] A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. In *Proceedings of the 16th International Conference on Software Engineering*, pages 91–100, 1994.

[9] E. Moss. Nested Transactions and Reliable Distributed Computing. In *Proceedings of the International Conference on Reliability in Distributed Software and Database Systems*, pages 33–39. IEEE, July 1982.

[10] S. Nettles and J.M. Wing. Persistence + Undoability = Transactions. Technical Report CMU-CS-91-173, School of Computer Science, Carnegie Mellon University, August 1991.

[11] OMG. CORBAservices: Common Object Services Specification. Technical report, Object Management Group, November 1995.

[12] R.D. Ranson. Less-Than-Transactional Semantics for TINA. In *6th Telecommunications Information Networking Architecture Conference (TINA 95)*, volume 2, pages 243–257, 1995.

[13] RM-ODP. Reference Model of Open Distributed Processing. Technical Report 10746, ISO/IEC Document, 1994.

[14] Titos Saridakis, Christophe Bidan, Valérie Issarny. Using Nonfunctional Execution Properties to Classify Software for Reuse. In *Franco-Japanese Workshop on Object-Based Parallel and Distributed Computing 1997 (OBPDC'97)*, 1997.

[15] TRANSARC. *Encina Toolkit Server Core Programmer's Reference*, 1997.

[16] Apostolos Zarras and Valérie Issarny. A Framework for Systematic Synthesis of Transactional Middleware. In *Proceedings of Middleware'98.* IFIP, Chapman-Hall, Sept 1998.