RC23326 (W0403-130) March 22, 2004
Computer Science

IBM Research Report

Semi-Automatic J2EE Transaction Configuration

Stephen J. Fink, Julian Dolby
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Logan Colby
IBM Rochester
Rochester, MN 55901

Semi-Automatic J2EE Transaction Configuration

Stephen Fink Julian Dolby
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598
{sjfink,dolby}@us.ibm.com

Logan Colby
IBM Rochester
Rochester, MN 55901
lcolby@us.ibm.com

ABSTRACT
This paper describes tool support to help an EJB^{TM} developer configure transactions for container-managed persistence. We formulate aspects the transaction configuration task as a dataflow problem, and apply inter-procedural program analysis to suggest configuration settings. We also discuss issues in the design and implementation of program analysis for EJB applications.

Experimental results show that the analysis effectively identifies transactions and optimized configuration settings, validating the proposed approach. The results suggest that a transaction-scoped configuration model allows superior configuration settings than the more common model of method-based configuration. The experiments additionally evaluate cost/precision trade-offs among various call graph construction algorithms as applied to this task.

1. INTRODUCTION
The Java 2 Enterprise Edition (J2EE^{TM}) programming model provides a high level of abstraction for transaction-oriented business applications. Enterprise Java Beans [21] offers one of the most powerful J2EE features: *container-managed persistence (CMP)*. With CMP, the developer specifies a mapping between Java objects (entity beans) and a persistent store, and the runtime system (EJB container) manages persistence automatically. CMP substantially reduces coding effort, as compared to hand-coding persistence with embedded SQL[17] commands such those provided by the Java Database Connectivity (JDBC^{TM}) API.

To preserve atomicity, consistency, isolation and durability (ACID) properties, EJB programs depend on transaction support. To integrate transactions with CMP, the EJB specification includes support for declarative *transaction attributes*; the deployment descriptor can specify a correspondence between an EJB method and a transaction scope. Based on these transaction attributes, the EJB container establishes transactional contexts at appropriate times during program execution.

When managing CMP, the EJB container faces a variety of implementation choices concerning database interaction. The EJB specification remains silent on many database policy choices that can make or break performance and scalability. For example, high-performance database management systems allow the programmer to specify connection isolation levels, caching protocols, and locking protocol choices in order to maximize concurrency, avoid deadlock, and still preserve desired ACID properties. Additionally, hand-coded programs might exploit program invariants to avoid unnecessary database communication or to optimize certain SQL queries. The EJB container cannot generally exploit these invariants since it cannot infer use cases for the persistent beans.

To address these limitations, J2EE application server vendors such as IBM [16] and BEA [4] have introduced vendor-specific transaction configuration options. With these options, the developer can fine-tune the container's interaction with the database and improve performance and scalability.

In order to set J2EE transaction configurations correctly, one must carefully consider the EJB business logic, the semantic impact of declarative deployment meta-data, and the application's ACID requirements. Even for small-scale applications, transaction configuration often proves an error-prone process. For large-scale production applications, the difficulties compound. Furthermore, EJB programming model "roles" exacerbate the problem. Often, the "Assembler" role player finds herself responsible for transaction configuration across multiple large software components, written independently by various "Bean Providers", and sometimes provided without source code. Should the Assembler misconfigure transactions, the deployed application may suffer excessive serialization, deadlocks, unintended rollbacks, or even incorrect output.

Further compounding the problem, developers have observed that the EJB specification's method-based declarative model often precludes optimal transaction configuration. The EJB specification associates transaction meta-data with a static method declaration, and the meta-data governs behavior each time the associated method executes. However, in practice, an application often uses a particular method and associated entities in different ways in different transactions. Method-scoped transaction configuration cannot express different policies for a single method or entity in different contexts.

To address this limitation, IBM's WebSphere^{TM} Integration Server provides a more powerful transaction configuration facility based on "tasks", or programmer-specified units of work. With this functionality (called *Application Profiles* in product documentation), the EJB Assembler can declaratively identify scoped units of work corresponding to transactions, and configure transactions according to the task scope. This functionality gives more freedom to optimize transactions; however, with more freedom comes more complexity, and a greater burden on the hapless EJB Assembler.

To address the difficulty, we have developed a tool to semi-automatically configure transactions for J2EE applications using CMP. The tool analyzes a J2EE application, reports salient properties of its transactions and container-managed entities, and suggests configuration settings based on the analysis results. The tool requires human oversight to audit the results, to account for potentially overly conservative and/or unsound analysis results.

The main contributions of this paper are

- a formulation of aspects of the transaction configuration problem as a dataflow framework, amenable to well-understood inter-procedural analysis (IPA),
- a discussion of architectural and engineering issues for building a production-quality, extensible analyzer that incorporates J2EE semantics,
- an evaluation of a range of call graph construction algorithms for this task, and
- an evaluation of the task-based configuration model, as compared to the simpler and more familiar method-based alternative.

Experimental results show that the analysis effectively identifies transaction boundaries and suggests significantly optimized configurations based on read/write behavior for container-managed fields. The results demonstrate that a transaction-scoped configuration model, supported by WebSphere's Integration Server, allows superior configuration settings than two less fine-grained alternatives. The experiments additionally evaluate cost/precision trade-offs among various call graph construction algorithms as applied to this task. In most cases, RTA [3] and 0-CFA [26] call graph construction algorithms allow the same quality of configuration settings as do several more precise CFA variants.

To our knowledge, this work describes the first IPA implementation which incorporates EJB constructs such as container-managed persistence, transaction demarcation, container-managed relations, and message-driven execution. Additionally, the experiments present the first published evaluation of competing call-graph construction algorithms for J2EE applications.

We believe the results validate the proposed approach as a valuable contribution to a J2EE development process. A version of the configuration tool will ship shortly with several products in IBM's WebSphere [16] family.

The remainder of this paper proceeds as follows. Section 2 reviews relevant background of the EJB programming model and configuration models. Section 3 presents the goals and high-level design of the transaction configuration tool. Section 4 describes the program analysis, and Section 5 reviews a number of issues that arise in implementation. Section 6 presents an experimental evaluation, examining analysis trade-offs and evaluating different configuration models. Section 7 reviews related work, and Section 8 concludes.

2. BACKGROUND

In this section, we review aspects of the J2EE programming model which pertain to this paper, and show an example program fragment to illustrate usage of container-managed persistence (CMP).

CMP provides a high level of abstraction for managing persistent objects in a distributed application. Under CMP, the EJB developer[1] specifies a mapping between distinguished Java objects (*entity beans*) objects and a persistent store such as a relational database. A component of the J2EE application server runtime system called the *EJB container* manages communication between the database and the application space.

Figure 1 shows an example of EJB program fragment that uses CMP to mediate between EJB business logic and a relational database. The remainder of this section discusses various aspects of the EJB programming model as illustrated by this example.

2.1 Entity Beans

Figure 1a shows a simple relational database schema with three tables: *Person*, *Account* and *Person2Account*. The *Person* and *Account* tables are indexed by a primary key called the **id** field, and each row holds two other data fields. Additionally, each table's schema includes a *foreign key* column (**owners** and **accounts**, respectively) which specifies a *relationship* between rows in *Account* and *Person*. Since one person can have more than one account, the relation *Person2Account* maps from keys in the **accounts** field to multiple *Account* **id**s. (A more realistic relationship that allows joint accounts would needlessly complicate our example.) The example table shows two accounts numbered 1 and 2, which belong to person 0.

In the EJB programming model, the developer defines objects called *entity beans* which correspond to data in the persistent store. In the example, Figure 1b shows two entity beans, *PersonEnt* and *AccountEnt*, which correspond to rows in the *Person* and *Account* tables, respectively. For each entity bean, the Bean Developer provides a *Home* interface and a *Remote* interface. The EJB program uses the *Home* interface to find,create,and remove bean instances, and the *Remote* interface to access entity bean state and in-

[1]Unless otherwise noted, we informally use the term "EJB developer" to refer to the Bean Provider, Assembler, and/or Deployer "role" of the EJB specification, according to which role(s) makes sense in context.

field	**id**	**name**	**address**	**accounts**
type	`int`	`String`	`String`	`int`
	0	Jane Doe	1534 Front St	4

Person Relation

field	**id**	**interest**	**balance**	**owner**
type	`int`	`float`	`int`	`int`
	1	.025	100000	0
	2	.040	500000	0

Account Relation

field	**owner**	**account**
type	`int`	`int`
	4	1
	4	2

Person2Account Relation

a)

```
// Home interface for the Account entity bean
public interface AccountEntHome extends EJBHome {

  // find AccountEnt with a certain primary key
  AccountEnt findByPrimaryKey(int id);
}

// Remote interface for the Account entity bean
public interface AccountEnt extends EJBObject {

  // access the balance field
  int getBalance();
  void setBalance(int b);

  // access the interest field
  float getInterest();
  void setInterest(float i);

  // get my owner from CMR
  PersonEnt getOwner();
}

// Home interface for the Person entity bean
public interface PersonEntHome extends EJBHome {

  // find PersonEnt by id
  PersonEnt findByPrimaryKey(int id);
}

// Remote interface for the Person entity bean
public interface PersonEnt extends EJBObject {

  // access the name field
  String getName();

  // access the address field
  String getAddress();

  // access AccountEnts related
  // to this Person via a
  // container-managed relationship
  Collection getAccounts();
}
```

b)

```
public interface InterestSesHome extends EJBHome {
  InterestSes create();
}

public interface InterestSes extends EJBObject {
  void payInterestAccount(int id);
  void payInterestOwner(int id);
  void getRate(int id);
}

class InterestSesEJB implements SessionBean {

  private AccountEnt getAccount(int id) {
    InitialContext c = new InitialContext();
    AccountEntHome H = (AccountEntHome) c.lookup("ejb:AccountHome");
    return H.findByPrimaryKey(id);
  }

  // transactional attribute: REQUIRED
  private void payInterestInternal(AccountEnt A) {
    A.setBalance((1+A.getInterest())*A.getBalance());

  // transactional attribute: REQUIRED
  void payInterestAccount(int id) {
    payInterestInternal(getAccount(id));
  }

  // transactional attribute: REQUIRES NEW
  void payInterestOwner(int id) {
    InitialContext c = new InitialContext();
    PersonEntHome H = (PersonEntHome) c.lookup("ejb:PersonHome");
    PersonEnt P = H.findByPrimaryKey(id);
    Collection accts = P.getAccounts();
    for(Iterator i = accts.iterator(); i.hasNext())
      payInterestInternal((AccountEnt)i.next());
  }

  // transactional attribute: SUPPORTS
  float getRate(int id) {
    return getAccount(id).getInterest();
  }
}
```

c)

```
void service(ServiceRequest req, ServiceResponse res) {
  InitialContext c = new InitialContext();
  InterestSesHome H = (InterestSesHome) c.lookup("ejb:Interest");
  InterestSes b = H.create();

  b.payInterestAccount(account);
  b.getRate(account);

  UserTransaction T = (UserTransaction) c.lookup("MyTransaction");
  T.begin();

  b.payInterestOwner(owner);
  b.payInterestAccount(account);
  b.getRate(account);

  T.commit();
}
```

d)

Figure 1: EJB code fragments illustrating container-managed persistence. a) RDBMS Tables b) Entity Beans for Account and Person c) Session bean for handling interest d) Service method of client servlet

voke business logic.[2] The Bean Developer would also specify in the deployment descriptor a mapping between fields in entity beans and rows in the relational database. For our example, the mapping is straightforward; however more complex mappings are supported.

More concretely, Figure 1b shows the home interface for the *Account* entity bean: *AccountEntHome*. The *findByPrimaryKey* method returns an instance of *AccountEnt*, set up to represent the corresponding row in the persistent *Account* table. *AccountEnt* instances provide accessor methods to read and update the container-managed fields **interest** and **balance**, which map to the corresponding fields in the persistent row. At transaction boundaries, discussed shortly, the EJB container ensures that reads and updates to *AccountEnt* instances are kept consistent with the backing database row.

The example also illustrates an EJB feature called a *container-managed relationship (CMR)*. In this case, the deployment descriptor declares a *one-to-many* relationship between *Persons* and *Accounts*, which represents our database schema. The entity beans provide access to other beans through CMR accessors; namely *getOwner* and *getAccounts*. The EJB container ensures that relationships between entity beans translate into relationships between database rows expressed by foreign keys.

In addition to entity beans, the J2EE programming model also provides session beans and message-driven bean abstractions. These beans do not use CMP, but can utilize other EJB container services including transaction support.

2.2 Transaction Support

The J2EE programming model offers two ways to express transaction boundaries: container-managed transactions (CMTs) and bean-managed transactions (BMTs).

To define CMT boundaries, the EJB developer inserts declarative transaction attributes in the EJB deployment descriptor. The EJB Specification [21] defines six possible transaction attributes which control CMT demarcation, with the following informal semantics:

REQUIRES NEW Execute the callee in a new CMT. If the caller has an active transaction, suspend it until the call returns.

REQUIRED If the caller has an active transaction, the callee inherits the transaction context. Otherwise, execute the callee in a new CMT.

MANDATORY If the caller has an active transaction, the callee inherits the transaction context. Otherwise, raise an error.

NEVER If the caller does not have an active transaction, the callee executes without a transactional context. Otherwise, raise an error.

[2]For expository purposes we have elided some necessary methods in the example code fragments, and avoid discussion of the parallel *Local* and *LocalHome* interfaces.

SUPPORTS If the caller has an active transaction, the callee inherits the transaction context. Otherwise, the callee executes with no transactional context.

NOT SUPPORTED If the caller has an active transaction, suspend it until the call returns. The callee executes with no transactional context.

In contrast to these declarations, BMTs arise according to how the program manipulates first-class *UserTransaction* objects. Typically, the code acquires a handle to a *UserTransaction* object via a naming service, and initiates a BMT by invoking the *begin()* method on the object. The BMT typically ends with a call to *commit()* or *rollback()*.

Figure 1c shows a session bean called *InterestSesEJB* that manages interest payments to bank accounts. The deployment descriptor associates transaction attributes with methods of the bean, as noted in the figure.

The session bean enlists and interacts with entity beans to perform its business logic functions. Consider the method *payInterestInternal*, which updates an account's balance by paying interest to it. The Bean Developer has declared this method's transactional attribute REQUIRED, enforcing the desired ACID properties for this update. With the REQUIRED attribute, the container will execute *payInterestInternal* as part of a calling transaction. If none exists, the container will create a new transactional context for the scope of the method.

Similarly, the *payInterestAccount* method looks up an account by its primary key, and pays interest. This method also REQUIREs a transaction. Note that this method invokes a helper method *getAccount* which acquires a handle to the desired *AccountEnt* instance, invoking the necessary functions on a naming service and the *Account* home interface. The call to *findByPrimaryKey* is said to **enlist** the *AccountEnt* instance for the duration of a transaction. The EJB container keeps track of enlisted entities, and performs persistence management on enlisted entities when the governing transaction commits.

The *payInterestOwner* method employs the container-managed relationship. This method finds a *Person*, and then pays interest for each account the person owns. The Bean Developer intends for the set of interest payments to appear as a single atomic operation, and so declares a REQUIRES_NEW transactional attribute for *payInterestOwner*. REQUIRES_NEW ensures the desired atomicity, and also forces the container to attempt to commit the operation immediately when this method returns.

In a common J2EE development pattern, other J2EE constructs such as servlets – Java classes invoked by Web servers — act as clients of the EJB application layer. Figure 1d illustrates the *service* method of one servlet. It first pays interest to some account; based on the REQUIRED attribute of *payInterestAccount*, the container will create a transaction for the duration of the this call. Then *service* executes *getRate*, which does not force a transaction to occur.

Next, the servlet decides to manage transactions itself, by initiating a BMT. The servlet acquires a *UserTransaction* object from a transaction manager, and calls *begin* to start a new transactional context. Then, *service* calls *payInterestOwner*; since this method is declared REQUIRES_NEW, the container suspends the BMT and creates a new CMT for the duration of the call.[3] The servlet's next two calls, to *payInterestAccount* and *getRate*, both execute in the context of its BMT.

2.3 Transaction Configuration

Application server vendors such as IBM [16] and BEA [4] have introduced vendor-specific transaction configuration options. These additional options allow the EJB developer to exercise more control over the container's interaction with the database. For example, with IBM's WebSphere Application Server supports configuration options for a transaction's SQL isolation level, concurrency control protocol, and pre-fetching directives ("read-ahead") which enable the container to combine distinct database queries. BEA WebLogic supports similar extensions. These vendor-specific extensions often prove indispensable for high performance on customer applications and some industry benchmarks.

When using CMP, the EJB container *enlists* an entity bean for the duration of a J2EE transaction. The first time during a transaction that the business logic accesses a particular entity bean instance, the container interacts with the database[4] to fetch the entity's current state, possibly acquiring database locks in the process. When the transaction commits, the container negotiates with the database again to ensure that any updates reflect to the persistent store, and to release locks.

Vendors commonly support *method-specific* declarative configuration options. Under this model, meta-data associates configuration options with a particular EJB method. When the container enlists a entity via a call to said method, it obeys the associated transaction configuration. For example, IBM and BEA both support method-specific "read-only" vs. "for-update" configurations. For some concurrency control implementations, the container can exploit the "read-only" configuration to avoid acquiring database write locks and avoid unnecessary update operations when a transaction commits. Thus, "read-only" configuration can increase concurrency and reduce overhead. A transaction that misbehaves by modifying a "read-only" enlistee may suffer a rollback or a costly lock escalation.

As a further optimization, BEA WebLogic supports a "partial reads" configuration for entity enlistments. If a particular transaction will not read particular fields of an entity bean, then the container can avoid fetching those fields from the database, reducing database workload and container overhead. This can substantially improve performance, and BEA exploits this feature in its SPECjAppServer submissions [27].

Our example application can benefit from these optimizations. Consider the enlistment of the Person entity by *findByPrimaryKey* in the *payInterestOwner* method in Figure 1c. This business logic only reads state of this *Person* instance, and does not read the *Person*'s `name` and `address` fields. So, this particular enlistment could be configured for both read-only and a partial-read.

In many cases, a single statement may enlist entities on behalf of different calling contexts which use the enlisted entity in different ways. For example, consider the enlistment of an *Account* by *findByPrimaryKey* in the *getAccount* method. When called from *getRate*, the *Account* instance is read-only and only reads the **interest** field; however, when used from *payInterestAccount*, business logic reads and writes **balance** in addition to reading **interest**. So, we cannot effectively optimize the enlistment by associating method-specific configuration meta-data with the *getAccount* method. Instead, we need a more fine-grained configuration model.

To address this problem, WebSphere Integration Server [16] supports more flexible transaction configuration with a feature called *Application Profiles*. With this feature, the assembler can declaratively specify a *task boundary*, which corresponds to either a declarative or programmatic transaction boundary. When the container enters a new transaction scope, it can dynamically install a new transaction configuration for a set of EJB methods. For the scope of the transaction, the container obeys this "application profile" to control database interactions. The application profile can change with each transaction. So, we can configure different profiles for *getRate* (read-only and partial read of **interest**) and *payInterestAccount* (no optimizations possible).

3. TOOL ARCHITECTURE

We have developed tool support for semi-automatic configuration of transaction attributes for CMP EJBs. The tool enhances a J2EE integrated development environment such as WebSphere Studio Application Developer [16]. Figure 2 shows the tool architecture.

The tool analyzes the J2EE application and produces a report summarizing how the application uses transactions and how each transaction uses entity beans. As shown in the figure, the analysis takes input from four sources:

- **Business Logic**: the developer's Java code,
- **Data Source Information**: configuration data regarding the persistent store (relational database),
- **Deployment Descriptor**: XML files holding declarative specifications, and
- **Application Use Cases**: input parameters specifying other runtime configuration meta-data.

The tool analyzes these artifacts and produces a report summarizing a set of *task identifiers* identified in the application. As discussed in the previous section, we associate a task with each J2EE transaction boundary; a task identifier serves as a static name for a set of tasks that might arise at runtime. In our current implementation, each task identifier corresponds to a program point at which a transaction may begin.

[3]Note that EJBs do *not* support nested transactions.

[4]For this discussion, we assume the container is not configured to exploit aggressive bean caching options.

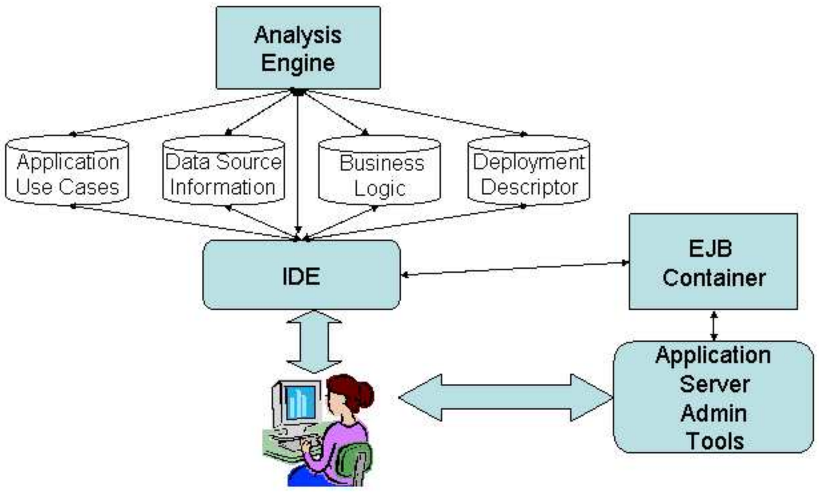

Figure 2: Block architecture of the semi-automatic configuration tool.

The report lists, for each task, the set of entity beans that the task might enlist during its execution. For each entity bean configured with CMP, the tools reports whether the task may create, remove, read, or update the bean. If read or updated, the tool reports which fields may be read or updated.

The IDE presents the report in a GUI format for perusal by the developer. When prompted, the tool will *suggest* transaction configuration attributes based on the report. The particular configuration attributes suggested depend on the configuration model supported by the application server.

In the remainder of this paper, we consider three possible configuration models for CMP configuration. Each model supports transactions configurations which apply to CMP entities under a particular *scope*. We consider the following three definitions of the scope:

Bean scope For a bean B, bean scope $< B >$ denotes all uses of B in the application.

Method scope For a bean B and an EJB method m, we define the method scope $< B, m >$ to be all uses of B which occur when a transaction enlists B by invoking method m.

Transaction scope For a bean B and a task T, we define the transaction scope $< B, T >$ to be all uses of B in task T.

These three scope definitions represent three possible designs for specifying transaction configurations. With bean-scoped, all enlistments of a particular entity share the same transaction configuration. With method-scoped, all enlistments of a particular entity via a particular method share the same configuration. With transaction-scoped, all enlistments of a particular entity in a particular transaction share the same configuration. Thus, the three options represent progressively more fine-grained configuration models. Method-scoped corresponds to the transaction attribute model of the EJB Specification [21], and transaction-scoped corresponds to the WebSphere Application Profile feature [16]. We include the Bean scope as a third, even simpler alternative.

For the remainder of this paper, we focus on two aspects of transaction configuration: *read-only* configuration and *partial-read* configuration for enlisted entity beans. The tool will suggest a transaction configuration that will maximize performance while conservatively respecting potential program behavior.

More specifically, for read-only configuration: Given a set of scopes S, the tool aims to determine the maximal subset $S' \in S$ such that each $s \in S'$ does not update any persistent bean state.

For partial-read configuration: for each $s \in S$ we specify a set of container-managed fields F_s such that s contains no reads of any $f \in F_s$. For each $s \in S$, the goal is to maximize the set F_s, which will in turn minimize the communication with the database.

The current implementation provides **semi-automatic** configuration support; the tool suggests transaction configurations and presents an analysis report to justify its recommendations; however, a human may choose to accept, reject, or modify the configuration. With the current analysis technology, we believe human oversight is required to safeguard against misconfiguration due to analysis limitations.

For example, program analysis results may be overly *conservative*; the analysis may not incorporate enough precision to recommend the optimal configuration.

Alternatively, the analysis may not be *sound*; it may rely on unsafe assumptions about program behavior. While sound analysis is preferable where possible, J2EE codes use many dynamic language features that cannot be completely analyzed statically. Instead, the implementation sometimes makes unsafe assumptions in order to produce useful reports despite analysis limitations. In Section 5, we discuss some unsafe assumptions regarding analysis of reflection and related services. The current implementation also relies on other unsafe assumptions such as having the whole program available, adherence to contracts in the J2EE specification, and no use of user-defined native code. The tool reports unsafe assumptions made to the user, for human consideration when evaluating configuration suggestions.

4. ANALYSIS

4.1 Overview

We define a program analysis to suggest the transactional configuration. The algorithm has three steps:

1. Construct a call graph,
2. Identify transactions in the program,
3. Traverse the program representation to determine how each transaction uses each entity.

The literature presents many algorithms for call graph construction in object-oriented languages. See [11] for an in-depth review. Section 5 discusses some issues particular to analyzing EJB applications.

The following sections describe our transaction analysis, which takes the call graph as input. Each node in the call graph represents some method in program, analyzed in some context. Thus, the graph may have many nodes representing a particular method, if the graph supports many contexts per method. Edges in the call graph represent feasible calls between nodes.

4.2 Transaction Analysis—Definition

We model each J2EE transaction as a side effect of a method dispatch, and name each distinct transaction as an edge e in the call graph.

First consider CMTs. Each CMT begins when the program calls a method with a REQUIRED or REQUIRES NEW transaction attribute. We name a container-managed transaction as an edge $e = (m, n)$, where m and n are nodes in the call graph, and n represents a method marked with a REQUIRED or REQUIRES NEW attribute.

Each bean-managed transaction begins during a call to *UserTransaction.begin()*. We model *begin()* as a method with an empty body, whose invocation begins a transaction. So, we name a bean-managed transaction as $e = (m, n)$, where n is a node corresponding to *UserTransaction.begin()*.

The goal of transaction analysis is twofold:

1. determine the set of transactions; i.e., determine the set of edges in the call graph whose invocation gives rise to a transaction, and
2. label each basic block in the (inter-procedural) control-flow graph with the set of transactions which may be in scope when the block executes.

The problem maps directly to an instance of a monotone, distributive dataflow framework. Let CG be the call graph, and let G be the inter-procedural flow graph derived from CG, in the style of Sharir and Pneuli [25]. That is, G consists of the union of the individual control flow graphs, one for each node in CG. Additionally, for each edge in the call graph, G contains a *call* edge and a *return* edge. Figure 4 shows part of the inter-procedural flow graph corresponding to the sample code in Figure 3.

Following standard practice, we define the dataflow instance by associating a flow function with each edge in the inter-procedural flow graph. Figure 5 shows the flow function definitions that describe how transactions arise. At each program point, we define C to be the set of CMTs in scope and B to be the set of BMTs in scope. We define a distinguished BMT called E to represent the *empty* transaction context. The meet operator is set union: for a basic block X, the transactions in scope at the entry to X, or $< C_X, B_X >$, is the union of all transactions that flow into X. In more detail, the flow functions along edges work as follows:

call edges The flow function for *call edges* follows directly from the J2EE definition of transactions. Both CMTs and BMTs flow from callers to callees, modulo CMT declaration.

- For NEVER and NOT SUPPORTED, the method is defined to execute without any transaction, so the flow function removes all CMTs and BMTs.
- The entry to main has the empty transaction by definition.
- REQUIRES NEW always executes in a new CMT, and so all other CMTs and all BMTs are suspended.
- REQUIRED methods, which must always execute in *some* transaction, have two cases: if it must always be called from within a transaction, the call changes nothing. If it may be called with the empty BMT, then the call removes the empty BMT and adds a new CMT.
- The UserTransaction.begin call starts a new BMT, suspending all CMTs and all other BMTs.
- Other calls change nothing.

```
void main() {
  requiresNew();
  UserTransaction t = foo();
  t.commit();
}

// transactional attribute: REQUIRES NEW
void requiresNew() {
  required();
}

// transactional attribute: REQUIRED
void required() {
  notSupported();
}

// transactional attribute: NOT SUPPORTED
void notSupported() {
  ...
}

UserTransaction foo() {
  UserTransaction t = lookup("myTransaction");
  t.begin();
  return t;
}
```

Figure 3: Example pseudo-code for transaction analysis.

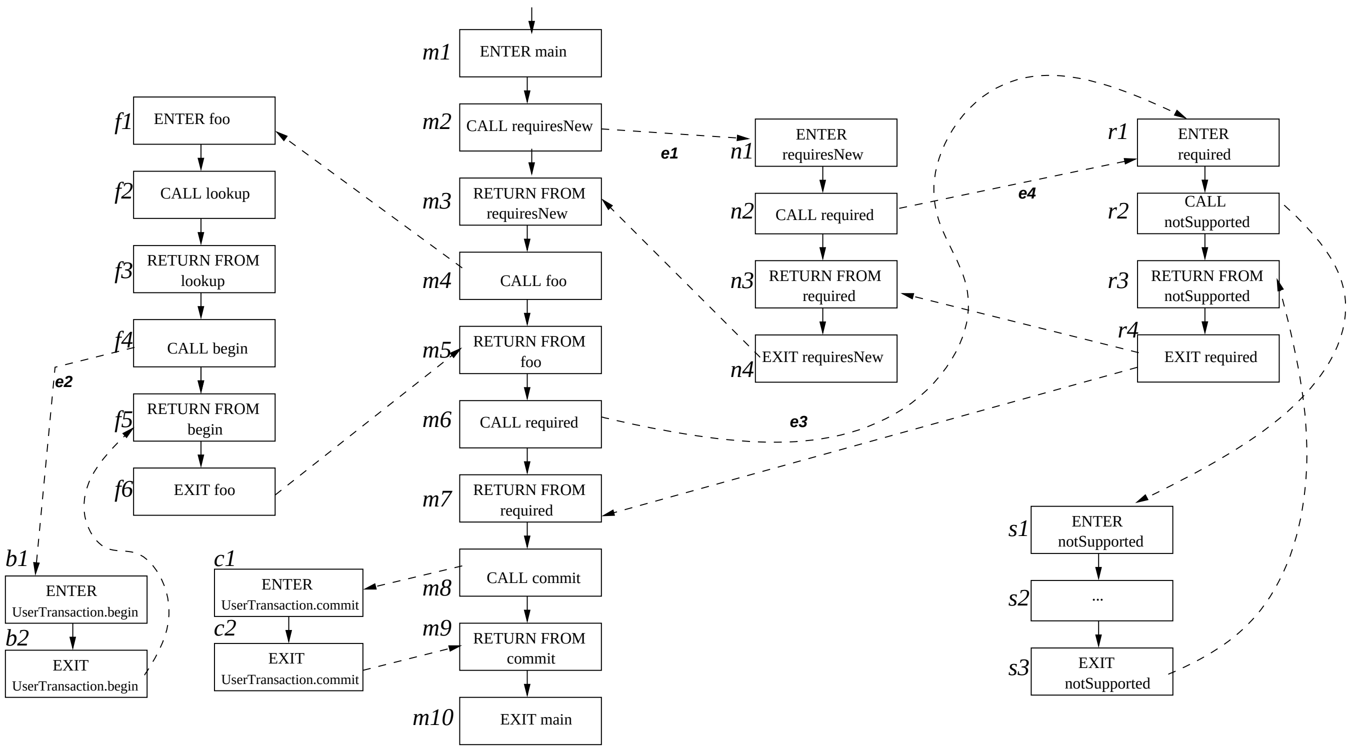

Figure 4: Inter-procedural flow graph corresponding to Figure 3.

For a *call* edge e to a node N,

$$f(< C, B >) = \begin{cases} < \epsilon, \{E\} > & \text{if } N \text{ is NOT SUPPORTED} \\ < \epsilon, \{E\} > & \text{if } N \text{ is NEVER} \\ < \epsilon, \{E\} > & \text{if } N \text{ is } \textit{entry} \text{ to main} \\ < \{e\}, \epsilon > & \text{if } N \text{ is REQUIRES NEW} \\ < C \cup \{e\}, B - \{E\} > & \text{if N is REQUIRED and } E \in B \\ < C, B > & \text{if } N \text{ is REQUIRED and } E \notin B \\ < \epsilon, \{e\} > & \text{if } N \text{ is } \texttt{UserTransaction.begin} \\ < C, B > & \text{otherwise} \end{cases} \tag{1}$$

For a *return* edge e from a node N,

$$f(< C, B >) = \begin{cases} < \epsilon, \{E\} > & \text{if } N \text{ is } \texttt{UserTransaction.commit} \\ < \epsilon, \{E\} > & \text{if } N \text{ is } \texttt{UserTransaction.rollback} \\ < \epsilon, B > & \text{otherwise} \end{cases} \tag{2}$$

For an intra-procedural *call-to-return* edge e, spanning a call to node N:

$$f(< C, B >) = \begin{cases} < C, B > & \text{if } N \text{ has a container-managed transaction declaration} \\ < C, \epsilon > & \text{otherwise} \end{cases} \tag{3}$$

For all other *intra-procedural* edges,

$$f(< C, B >) =< C, B > \tag{4}$$

Figure 5: Flow functions for transaction analysis.

Basic Block	Dataflow Function at Entry to Basic Block	Final Solution
$m1$	$< \epsilon, \{E\} >$	$< \epsilon, \{E\} >$
$m2$	$< C_{m1}, B_{m1} >$	$< \epsilon, \{E\} >$
$m3$	$< C_{m2}, B_{m2} > \cup < \epsilon, B_{n4} >$	$< \epsilon, \{E\} >$
$m4$	$< C_{m3}, B_{m3} >$	$< \epsilon, \{E\} >$
$m5$	$< C_{m4}, \epsilon > \cup < \epsilon, B_{f6} >$	$< \epsilon, \{e2\} >$
$m6$	$< C_{m5}, B_{m5} >$	$< \epsilon, \{e2\} >$
$m7$	$< C_{m6}, B_{m6} > \cup < \epsilon, B_{r4} >$	$< \epsilon, \{e2\} >$
$m8$	$< C_{m7}, B_{m7} >$	$< \epsilon, \{e2\} >$
$m9$	$< C_{m8}, \epsilon > \cup < \epsilon, \{E\} >$	$< \epsilon, \{E\} >$
$m10$	$< C_{m9}, B_{m9} >$	$< \epsilon, \{E\} >$
$n1$	$< \{e1\}, \epsilon >$	$< \{e1\}, \epsilon >$
$n2$	$< C_{n1}, B_{n1} >$	$< \{e1\}, \epsilon >$
$n3$	$< C_{n2}, B_{n2} > \cup < \epsilon, B_{r4} >$	$< \{e1\}, \epsilon >$
$n4$	$< C_{n3}, B_{n3} >$	$< \{e1\}, \epsilon >$
$r1$	$< C_{n2} \cup \{e4 \text{ or } \epsilon\}, B_{n2} - \{E\} > \cup < C_{m6} U \{e3 \text{ or } \epsilon\}, B_{m6} - \{E\} >$	$< \{e1\}, \{e2\} >$
$r2$	$< C_{r1}, B_{r1} >$	$< \{e1\}, \{e2\} >$
$r3$	$< C_{r2}, B_{r2} > \cup < \epsilon, B_{s3} >$	$< \{e1\}, \{e2\} >$
$r4$	$< C_{r3}, B_{r3} >$	$< \{e1\}, \{e2\} >$
$s1$	$< \epsilon, \{E\} >$	$< \epsilon, \{E\} >$
$s2$	$< C_{s1}, B_{s1} >$	$< \epsilon, \{E\} >$
$s3$	$< C_{s2}, B_{s2} >$	$< \epsilon, \{E\} >$
$f1$	$< C_{m4}, B_{m4} >$	$< \epsilon, \{E\} >$
$f2$	$< C_{f1}, B_{f1} >$	$< \epsilon, \{E\} >$
$f3$	$< C_{f2}, B_{f2} >$	$< \epsilon, \{E\} >$
$f4$	$< C_{f3}, B_{f3} >$	$< \epsilon, \{E\} >$
$f5$	$< C_{f4}, \epsilon > \cup < \epsilon, B_{b2} >$	$< \epsilon, \{e2\} >$
$f6$	$< C_{f5}, B_{f5} >$	$< \epsilon, \{e2\} >$
$b1$	$< \epsilon, \{e2\} >$	$< \epsilon, \{e2\} >$
$b2$	$< C_{b1}, B_{b1} >$	$< \epsilon, \{e2\} >$
$c1$	$< C_{m8}, B_{m8} >$	$< \epsilon, \{e2\} >$
$c2$	$< C_{c1}, B_{c1} >$	$< \epsilon, \{e2\} >$

Table 1: The flow functions of Figure 5 as applied to the flow graph of Figure 4.

return edges Return edges denote how transactions flow from callees to callers: CMTs do not and BMTs normally do. Thus, for most returns, all CMTs are removed and BMTs left. The `commit` and `rollback` operations end the current BMT, so they return the empty BMT.

call-to-return edges Call-to-return edges denote how transactions flow over calls. CMTs cannot be suspended or ended by a callee so they are always passed across call sites. A callee with CMT attributes will restore any current transaction, including BMTs, when it returns, so such callees pass all current transaction across the call. Other callees can potentially end a BMT, so it must flow through the callee and back. Hence, such callees do not pass BMTs across the call.

other edges Other edges are those not involved in calls in any way. Since all transaction operations are modeled as calls, such edges can have no effect.

One potentially confusing aspect is the interaction of the *call edges* and *return edges* on the one hand, and the *call-to-return edges* on the other. There are two normal cases. For methods with CMT attributes, the *call-to-return* edge passes all current transaction across it, and the *call edge* establishes the transaction context for the callee. For other methods, the CMTs are passed across by the *call-to-return edge* and the BMTs flow into the callee and back out to the caller.

Table 1 shows the flow functions for the flow graph of Figure 4. The last column of the table shows the precise solution to the dataflow system.

The set of flow functions is distributive, so the solution procedure of Reps, Horwitz, and Sagiv [23] would give the precise (meet-over-all-valid-paths) solution in polynomial time. Performance is paramount since the tool will perform this analysis on-line in an interactive setting. For large programs, even a compact representation of the entire inter-procedural flow graph and associated flow functions might be prohibitive in time and space costs. So, before building the inter-procedural flow graph, we exploit structure to simplify the problem, as discussed in the next section.

4.3 Transaction Analysis—Optimization

We can optimize the solution process by exploiting some properties of this system.

> **Property 1:** All blocks in a method will have the same container-managed transactions.

This is obvious from inspection of the flow functions of Figure 5. For all intra-procedural edges (including *call-to-return* edges), the container-managed transactions C are invariant. The set of container-managed transactions changes only on method calls.

Using similar reasoning, we notice a more general property:

> **Property 2:** If no call to *UserTransaction.begin*, *commit*, or *rollback* is transitively reachable from a call graph node N, then each basic block in N will have the same solution.

Property 2 expands on Property 1 by including BMTs. Clearly, each normal intra-procedural edge maintains invariance of the set of bean-managed transactions. From equation (3), no call to a method with a container-managed declaration will change the set of bean-managed transactions. Finally, since only calls to *UserTransaction.begin*, *commit*, and *rollback* can change the set of BMTs in scope, each call from N will return with the same set of BMTs as called with.

Using Property 2, we can eagerly collapse the CFG for most call graph nodes to a single node. For collapsible call graph nodes, we need only compute the set of transactions in scope at the entry block; the solution must be the same for all other blocks. Exploiting this fact, we have implemented a solver with two phases.

In phase 1, after building a call graph, we perform a simple reachability analysis over the call graph to identify collapsible nodes according to Property 2. Then, excluding these collapsible nodes, we compute the flow of BMTs through the inter-procedural flow graph. We solve this with demand-driven dataflow; that is, we construct portions of the inter-procedural flow graph as needed.

In phase 2, we consider only the "collapsed" flow graph, with one set of transactions tracked for each call graph node. We transfer the solution from phase 1 onto edges of the collapsed flow graph, and then solve. The solution for phase 2 is significantly cheaper than solving for the entire inter-procedural flow graph, as we have only one set per call graph node, rather than one set per basic block.

At the end of phase 2, we have a solution for each program point. Note that the solution is *precise* for CMTs; CMTs do not propagate along *return* edges, so the solution is identical when solved over all paths as if solved over only inter-procedurally-valid paths. Our current solver for phase 1 is a simple iterative solver that does not consider inter-procedurally-valid paths; thus, it does not compute the precise solution for BMTs. In future work, we plan to implement the algorithm of Reps et al. [23] to remedy this. In practice, we have not encountered pollution in the end results due to the imprecise BMT solution, although it is easy to construct test cases where such pollution occurs.

We further reduce the scope of transaction analysis by considering the overall context; recall that we use the the transaction analysis solution to determine which transactions may enlist which entity beans, in order to guide transaction configuration. So, in the end, only methods that may enlist entity beans are relevant to the end solution. As a pre-pass, we filter out all call graph methods which cannot transitively call any method in the application class loader, since these methods cannot affect transaction boundaries or enlist entities. This pruning eliminates portions of the standard Java libraries that do not affect the solution.

Finally, having (logically) labeled the inter-procedural flow graph with the transactions that may reach each basic block, it is straightforward to determine which transactions may enlist which entity beans. We simply traverse the call graph and record for each statement that may enlist an entity, which transactions may be in scope when the statement executes. As before, we exploit Property 2 to avoid building individual CFGs where it can be avoided.

5. IMPLEMENTATION

5.1 Analysis Infrastructure

We have developed a a general Java bytecode inter-procedural analysis framework, called DOMO. The infrastructure consists of roughly 70,000 lines of Java code.

One design goal was to support a range of cost/precision analysis trade-offs, to support various clients as effectively as possible. To this end, the infrastructure supports a range of object-oriented call graph construction and pointer analysis algorithms, focusing primarily on flow-insensitive algorithms. Following the methodology of Grove and Chambers [12], one composes a call graph construction algorithm by specifying policies to guide selection of context sensitivity at call sites, and disambiguation policies for modeling program variables and heap-allocated objects. We have implemented a range of call graph construction algorithms, and will discuss a few in the next section.

The implementation of the pointer analysis and call graph construction algorithms incorporates many of the optimizations such as those described for SPARK [19]. The system uses a dual-mode bit vector implementation, switching between sparse and dense representations to conserve space. All pointer analysis flow functions filter by declared type on-the-fly, and some keep history of old vs. new information to reduce redundant work. The core dataflow solver uses worklist-driven iteration, with heuristics to periodically recompute pseudo-topological order as the system of constraints grows dynamically. The system does not perform cycle detection. The system uses *field-sensitive* models [19], and tracks flow through locals with flow-sensitive def-use information from a register-based SSA representation [7]. We have also invested considerable effort into efficient bytecode parsing, a compact intermediate representation, and space-efficient set and graph data structures.

In most cases, the analysis safely models the JVM specification, including exceptional control flow. We do not consider potential side effects from concurrent operations on shared data structures; however, since EJBs are forbidden to manipulate threads, that should not affect correctness of the target analyses. The analysis models a typical J2EE deployment with three classloaders; one each for Application code, Extension (i.e., container runtime) code, and code from the Primordial class loader. The analysis will not handle more general use of user-defined classloaders.

5.2 Modeling EJBs

A major challenge was to design the analysis architecture with enough flexibility to effectively analyze higher-level semantics of J2EE.

An immediate design question is whether to analyze the application *before* deployment or *after* deployment. During deployment, EJB applications pass through an extensive source-to-source code generation step, which introduces implementation details of the target EJB container implementation.

We chose to analyze the program *before* deployment. Instead of analyzing the deployed code, we model by hand many aspects of how the program interacts with the EJB container. This choice has three advantages: 1) scalability – it reduces the body of code to analyze, 2) precision – it is likely that a human-generated summary of container semantics is more precise than could be inferred practically from the raw container implementation, and 3) portability – the analysis results do not vary depending on the container implementation. The choice has three disadvantages: 1) aesthetics – the modeling process introduces more human intervention, 2) human error – which invariably follows from human intervention, and 3) future maintenance work — needed to update the analysis as the J2EE specification changes in the future. Although we decided the pros outweigh the cons, dealing with human error in the J2EE models has been a constant challenge. It requires considerable domain expertise to model the interactions completely and correctly.

Modeling simple library methods, such as most native methods in the Java standard libraries, can be done concisely with a simple specification. For simple flow-insensitive models, we use a simple XML language to represent a method's semantics. This approach also suffices for some J2EE methods, when the method's definition is static and the semantics fixed. In many cases, we substitute synthetic models in place of standard J2SE and J2EE methods that we assert will not have side effects that affect the properties of interest (e.g. I/O). These models improve performance by reducing the scope of the analysis, and in many cases increase precision by eliminating opportunities for dataflow pollution.

Modeling EJBs presents more engineering challenges, since the set of methods and their behavior is determined by the application's deployment descriptor. For these cases, we have implemented a simple "EJB compiler" that takes as input the application code and the deployment descriptor, and produces analyzable artifacts that represent method behavior.

For example, suppose the analysis encounters a call to a method *PersonEnt.getAccounts()* from Figure 1b. Before attempting to resolve this call with standard Java semantics, the analysis checks with a listener design pattern [10] to see if any listener understands special semantics for this call. One "EJB Listener", consulting the deployment descriptor, notices that this method represents the container-managed relationship of *PersonEnt* to *AccountEnt*s. So, this listener will create an analyzable artifact representing the semantics of a call to *getAccounts()*, and notify the analysis driver to analyze the call as dispatching to the artifact.

To generate the artifact for this CMR access, the "EJB Compiler" consults the deployment descriptor to deduce bean AccountEnt's primary key type, remote interface, and home

```
java.util.Collection getAccounts() {
  AccountEntHome h = ContainerModel.getPooledAccountHomeInstance();
  AccountEnt B = H.findByPrimaryKey( 0 );
  HashSet S = new HashSet();
  S.add(B);
  if (condition) {
    throw new RemoteException();
  }
  if (condition) {
    throw new EJBException();
  }
  return S;
}
```

Figure 6: Pseudo-code showing the analyzable artifact generated for *PersonEnt.getAccounts()*

interface. Based on these types, it generates an artifact similar to that shown in Figure 6. The simple semantics there suffice to construct a correct call graph incorporating the call to *getAccounts()*. Note in particular the call to a class called *ContainerModel*. The *ContainerModel* is a distinguished analyzable artifact which simulates pooling of bean instances, along with other global container artifacts. Note also that this model for *getAccounts()* will not suffice for all possible client analyses. For example, the returned collection is modeled as always containing one element. In reality, it may have zero or many. We would have to further refine the generated model in order to support a client analysis that was sensitive this distinction.

Using similar logic, we have generated models for many aspects of the J2EE specification, including many functions for Servlets and JSPs, most CMP-related methods, much of JDBC, some SOAP functions, some Apache Struts functions [9], and some WebSphere-specific constructs such as dynacache support [6]. We have not yet modeled other features, such as EJB-QL, two-phase transactions, and those dark corners of Enterprise Java which still frighten us.

In our experience, the extensible architecture for high-level semantic models proved to be the single most important design feature in the entire implementation. The architecture allowed rapid prototyping of J2EE semantics as we encountered them in test applications. This rapid prototyping was crucial for timely completion of the implementation.

5.3 Dealing with reflection

Reflection and introspective services arise often in J2EE applications. In addition to “core” reflective instantiation with *newInstance*, J2EE applications will often create objects via invocations to services such as JNDI lookup, Java Bean instantiation, RMI *narrow* services, serialization, return values from objects such as *java.sql.ResultSet*, various flavors of servlet and JSP contexts and sessions, and message arguments to message-driven beans. For this discussion, we will call such objects *opaque* objects. Figure 1 shows several examples of opaque objects returned from a JNDI *lookup* method; this idiom appears frequently in J2EE applications.

It is impractical to expect a tool user to specify the behavior of calls to each of these services, or the type of each opaque object. While it may be possible to statically divine the behavior of some opaque services from configuration data, in other cases, we must fall back to conservative static estimates.

Due to the large number of opaque objects which arise from various services, we decided to handle all such cases with a single general mechanism. In all these cases, we have decided to unsafely assume that every *checkcast* on an opaque object succeeds, and use this assumption to infer types. Notice that this technique will handle the common usage of naming services as illustrated in Figure 1, where the obtained object is immediately downcast into a usable type.

We actually implement two separate schemes for deducing the type information, depending on the precision of the call graph/pointer analysis employed.

When we use a pointer analysis at least as precise as 0-CFA, we track the flow of objects to checkcast statements on-the-fly. We introduce a distinguished type called *Malleable* into the type system. We model each method that returns an opaque object as returning a new instance of type *Malleable*, and configure the pointer analysis to analyze each call to the method with one level of calling context. When the analysis evaluates the constraint corresponding to a *checkcast*, it records the relevant type T for the opaque object’s allocation site. The system then adds a constraint to indicate that the original allocation returns an object of type T, in addition to the *Malleable*. In this way, when the iterative constraint solver terminates, it will have added and solved constraints corresponding to all possible inferences from downstream *checkcasts*.

Less precise call-graph construction algorithms such as RTA [3] don’t track the flow of individual objects to individual statements. We have found in practice that a straightforward application of the *Malleable* inference for these algorithms is too imprecise; in RTA, each object flows to each checkcast, which winds up sucking in vast regions of the class hierarchy which are otherwise unreachable.

To handle reflection for these imprecise algorithms, we have implemented an iterative driver that separates call graph construction from type inference for opaque objects. After building the initial call graph, we construct def-use chains to track each opaque object inter-procedurally through the program, based on the initial call graph. We record each case where an object flows to a checkcast. Then we expand the original call graph based on this information. The process repeats until it discovers no new opaque objects.

Our current implementation does not track opaque objects through the heap; it only tracks the use of each opaque object through local variables and call/returns. Although this limitation is unsafe, in all cases, we have found at least one checkcast for each opaque object. Based on this fact and the observation that the original assumption is technically unsafe anyway, we have deemed the current implementation adequate for practical use.

We do not currently model method invocation via reflection (*java.lang.reflect.Method.invoke*); for the results reported here, ignoring these calls does not change the transaction configuration results. A more complete analysis would have to track *Method* object def-use chains to approximate these reflective invocations.

6. EXPERIMENTAL RESULTS

This section presents an empirical evaluation which characterizes the tool's output, compares call graph precision choices for this domain, and evaluates whether transaction-scoped configuration is substantially more useful than the simpler bean- and method-scoped models.

6.1 Benchmarks

For our evaluation, we collected a set of publicly-available benchmark programs and one real-world IBM customer code.

Table 2 lists the benchmarks used for this study. The *SPECjAppServer* codes are industry benchmarks that emulate a manufacturing, supply chain management, and order/inventory system [27]. The three versions are similar, all derived from the same original code base, but exercise different versions and features of the J2EE specification. *Trade3* is an IBM WebSphere sample application modeling an on-line stock brokerage applications [15]. *PetStore* is a sample application from Sun that models an on-line store [?]. Anonymous code *X* is a demanding, framework-intensive application from an IBM customer.

The table shows the number of classes and methods which are eligible for analysis from bytecode source. These numbers exclude those classes which are modeled artificially, and so represent a subset of the J2EE and J2SE libraries.

6.2 Call Graph Construction Algorithms

We evaluate the following call graph construction algorithms:

- **CHA**: class hierarchy analysis [8],
- **RTA**: rapid type analysis [3],
- **0-CFA**: context-insensitive control-flow analysis [26], disambiguating between heap objects according to concrete type,
- **0-1-CFA**: context-insensitive control-flow analysis, disambiguating between heap objects according to allocation site [12] (as in Anderson's analysis [2]) and
- **0-1-C-CFA**: 0-1-CFA enhanced with extra context for "container" objects, as described below.

0-1-C-CFA is motivated by the observation that container or collection objects often lead to large pollution in type estimates. So, 0-1-C-CFA adds an extra level of context-sensitivity for container objects, which we define as any object that *extends java.util.Collection*. For each method defined on a container object, we analyze each invocation to the method in a context defined by the receiver object (the container). Furthermore, we disambiguate objects allocated inside such methods by naming the instance according to a (container object, allocation site) pair. This extra context gives 0-1-C-CFA the power to disambiguate between objects stored in standard collections, which would otherwise flow to all uses of the container type.

6.3 Methodology

For each benchmark and each application, we analyzed a program that consisted of the application code itself and all third-party libraries that came bundled with it. To this we added the J2SE 1.4.2 libraries from Sun, with some internal implementation packages (e.g. most *com.sun* and *com.ibm* packages) excluded *a priori* from analysis. And finally, we included the J2EE libraries from IBM WebSphere Integration Server 5.1, again excluding most internal runtime packages.

All experiments reported were run on a single-processor IBM NetVista workstation with a 1.8Ghz Intel Pentium IV, 1.5GB of RAM. All runs use the Sun 1.4.2 JRE and `-Xmx800M`. For performance runs, each experiment was run in a separate JVM instance.

Since we use multiple call graph construction algorithms, the size of the analyzed code varies depending upon call-graph precision. The size statistics from Table 2 are a loose upper bound, since many methods are unreachable and never considered for analysis. Table 3 shows a more accurate metric: the size of the program actually analyzed for each experiment. The **Methods** count represents the number of distinct source methods that appear in the call graph. As discussed earlier, the implementation models reflective factory methods with a level of calling context, so even context-insensitive call graphs have more nodes than distinct methods. The table shows that overall, RTA produces much more precise call graphs than CHA, and 0-CFA produces significantly more precise call graphs than RTA. In terms of distinct methods discovered, the more precise CFA variants improve precision slightly over 0-CFA.

6.4 Results

We present four categories of results. First, we report results that characterize the reports generated by the tool, regarding distinct transactions and entity enlistments identified. Secondly, we report how effectively the analysis identifies opportunities for read-only configuration options. Next, we evaluate opportunities for partial-read configuration. For both read-only and partial-read configuration results, we evaluate each of the three configuration models discussed in Section 3: *bean-scoped* configuration, *method-scoped* configuration, and *transaction-scoped* configuration. (Recall that

Analysis Scope	Classes	Methods
Primordial Loader (J2SE Libraries)	5978	55535
Extension Loader (J2EE Libraries)	995	6360
`SPECjAppServer2001`	505	4055
`SPECjAppServer2002`	871	8574
`SPECjAppServer2003`	1085	9938
`Trade3`	117	1339
`PetStore 1.3.2`	853	5908
`X`	2782	34158

Table 2: Benchmarks used in this study, and the size of the analysis scope for each.

Benchmark	CHA		RTA		0-CFA		0-1-CFA		0-1-C-CFA	
	Methods		Methods		Methods		Methods		Methods	
	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges	Nodes	Edges
`SPECjAppServer2001`	19088		2031		1806		1801		1800	
	19358	198692	2210	7999	1985	5692	1980	5683	2500	6799
`SPECjAppServer2002`	21327		2121		1866		1860		1860	
	21633	223972	2266	8555	2011	5768	2005	5757	2475	6769
`SPECjAppServer2003`	18810		1864		1590		1590		1584	
	18998	192558	1981	7112	1707	4494	1707	4494	2018	5097
`Trade3`	19175		2464		1618		1590		1587	
	19342	191638	2544	14965	1697	4697	1669	4592	2047	5607
`PetStore`	20526		11700		9336		9297		9267	
	20663	218223	11777	91193	9393	44960	9354	44416	11744	50644
`X`	26644		9596		8106		7915		7903	
	26751	326021	9628	89234	8131	27619	7940	26650	14383	337858

Table 3: For each call graph construction algorithm, the number of distinct methods discovered, and the number of nodes and edges in the call graph.

these three configuration models offer progressively more fine-grained control over transaction configuration.) We also report differences in configuration results depending on the call graph construction algorithm.

Finally, since this tool runs interactively in an IDE, we report analysis runtimes for the different algorithms.

6.4.1 Transactions and Entities

Table 4 shows the number of transactions identified for each code, using each considered call-graph construction algorithm. More precise call graph algorithms can reduce the number of *false* transactions reported, since false transactions arise from false paths in the call graph.

For all codes, the tool reports a relatively small number of transactions, suitable for browsing in a GUI tool. These codes use CMTs heavily and BMTs rarely. For the 3 *SPECjAppServer* codes, the number of transactions does not vary according to call graph precision. Call graph precision does have a slight impact on *Trade3* and *X*, as RTA prunes 5 and 3 transactions respectively from CHA. For *Trade3*, 0-CFA prunes another 3 transactions from RTA. *PetStore* is interesting in that requires at least 0-CFA precision; otherwise, the tool reports at least 45 false transactions.

We do not know the number of false transactions reported by the most precise analysis, although manual inspection suggests that few false transactions remain. One could bound the results by instrumenting the container to record transactions dynamically; this remains a task for future work.

For each transaction, the tool reports the entity bean classes which the transaction may enlist. Table 5 shows the number of enlisted entities reported, depending on analysis precision. Less precise analyses will report more *false* enlisted entities.

The Table shows that for the *SPECjAppServer* codes, increased precision does not reduce the number of reported enlisted entities. Inspecting the results, we believe the *SPECjAppServer* reports include few if any false enlisted entities, even with CHA call graph construction.

However, for the other three codes, call graph precision has a significant effect on the number of reported entities enlisted. Each of these three codes uses general framework libraries to dispatch requests to EJBs. More precise call graph algorithms help refine control flow in these frameworks, reducing the number of false enlisted entities. For each code, RTA significantly improves over CHA, and 0-CFA significantly improves over RTA. We observed no benefit from the more precise CFA variants. These results correlate with the transaction results in that algorithms that improved the number of transactions tended to make the biggest differences on the numbers of enlisted entities as well.

As before, future work will investigate dynamic measurements to bound the number of false reports for the most precise analysis.

6.4.2 Read-only enlistments

In Table 6, we present the *read-only* configuration settings the analysis can infer under each of these three configuration models.

Somewhat surprisingly, a significant number of entity enlistments can be marked as read-only, even using the simple Bean Scope configuration model. In these cases, the analysis concludes that the Bean types are *never* updated after they are created. Secondly, we note that method scope exposes significantly more opportunities for read-only configuration than bean scope, and transaction scope offers even more opportunities. This supports WebSphere's decision to offer transaction-scoped configuration; the finer-grain control pays off in allowing more aggressive configuration.

Note that we cannot directly compare results between call graph algorithms in Table 6, since we saw previously that the less precise algorithms report a significant number of *false* enlisted entities. Table 6 also reports a significant number of *false* enlisted entities.

6.4.3 Partial-read enlistments

Next, we consider *partial-read* configuration settings. Recall that a partial-read configuration specifies only a proper subset of a bean's fields need be read, since the developer asserts that other bean fields will not be read. In effect, this provides a form of dead assignment elimination.

Table 7 shows the percentage of fields that the analysis determines *may be read* under each configuration scope option. If the analysis determines that a field must not be read in a particular scope, then the tool can suggest a partial read configuration to avoid fetching said field from the database. In order to make the numbers more comparable across call-graph construction algorithms, we have filtered them by counting only for those enlistments that were discovered by the most-precise algorithm (i.e. 0-1-C-CFA).

The data indicates that the more fine-grained configuration options expose much greater opportunities for optimization via partial reads. The different call-graph construction algorithms compute essentially identical results for those enlistments discovered by the most precise-algorithm. Using transaction configuration, in most cases the analysis suggests a partial read configuration that would avoid fetching roughly 70% of the persistent fields, assuming each entity enlistment occurred the same number of times. Of course, the actual run-time improvement depends on the relative frequency of individual entity enlistments. It remains a topic for future work to evaluate these configurations with respect to dynamic program behavior.

6.4.4 Tool performance

Finally, we report on the performance of the tool. Figure 7 shows the tool running time for each configuration reported. The figure shows the wall clock time for running the analysis, broken into five categories:

- **Init**: Reading class files and building a class hierarchy structure. Almost all this time is due to formatted I/O and deflating compressed jar files.
- **CallGraph**: Call graph construction.

Benchmark	CHA		RTA		0-CFA		0-1-CFA		0-1-C-CFA	
	CMT	BMT	CMT	BMT	CMT	BMT	CMT	BMT	CMT	BMT
SPECjAppServer2001	41	0	41	0	41	0	41	0	41	0
SPECjAppServer2002	41	0	41	0	41	0	41	0	41	0
SPECjAppServer2003	68	0	68	0	68	0	68	0	68	0
Trade3	115	3	110	2	107	2	107	2	107	2
PetStore	98	2	98	2	53	2	53	2	53	2
X	147	0	144	0	144	0	144	0	144	0

Table 4: Number of distinct container-managed transactions (CMTs) and bean-managed transactions(BMTs) identified by the tool with each call graph construction algorithm.

Benchmark	CHA	RTA	0-CFA	0-1-CFA	0-1-C-CFA
SPECjAppServer2001	81	81	81	81	81
SPECjAppServer2002	80	80	80	80	80
SPECjAppServer2003	115	115	115	115	115
Trade3	300	172	130	130	130
PetStore	432	416	259	259	259
X	1153	798	780	780	780

Table 5: The total number of enlisted entities discovered, using each call graph construction algorithm.

Benchmark	CHA	RTA	0-CFA	0-1-CFA	0-1-C-CFA
Bean Scope					
SPECjAppServer2001	36	36	36	36	36
SPECjAppServer2002	36	36	36	36	36
SPECjAppServer2003	37	37	37	37	37
Trade3	0	0	0	0	0
PetStore	96	90	54	54	54
X	171	50	50	50	50
Method Scope					
SPECjAppServer2001	44	44	44	44	44
SPECjAppServer2002	40	40	40	40	40
SPECjAppServer2003	46	46	46	46	46
Trade3	20	35	55	55	55
PetStore	104	98	62	62	62
X	612	494	494	494	494
Transaction Scope					
SPECjAppServer2001	51	51	51	51	51
SPECjAppServer2002	51	51	51	51	51
SPECjAppServer2003	67	67	67	67	67
Trade3	187	147	105	105	105
PetStore	146	140	90	90	90
X	1089	734	716	716	716

Table 6: Number of entity bean enlistments that the tool infers are safe to configure with a read-only attribute.

Benchmark	CHA	RTA	0-CFA	0-1-CFA	0-1-C-CFA
Bean Scope					
SPECjAppServer2001	52	52	52	52	52
SPECjAppServer2002	49	49	49	49	49
SPECjAppServer2003	61	61	61	61	61
Trade3	99+	99+	99+	99+	99+
PetStore	74	74	75	75	75
X	96	96	96	96	96
Method Scope					
SPECjAppServer2001	38	38	38	38	38
SPECjAppServer2002	33	33	33	32	32
SPECjAppServer2003	37	37	37	37	37
Trade3	72	70	66	66	66
PetStore	38	40	43	43	43
X	35	35	35	35	35
Transaction Scope					
SPECjAppServer2001	28	28	28	28	28
SPECjAppServer2002	27	27	26	26	26
SPECjAppServer2003	32	32	32	32	32
Trade3	35	24	20	20	20
PetStore	18	18	23	23	23
X	21	21	21	21	21

Table 7: The percentage of fields that the analysis reports may be read in each configuration scope option. Partial read configuration can avoid fetching any fields that must not be read.

- **Transactions**: Perform dataflow analysis to identify transactions and label the call graph.
- **Tally**: Traverse the call graph and collect usage statistics for each (entity,transaction) pair.
- **Other**: Miscellaneous overhead such as packaging the analysis results for consumption by a GUI client.

The figure shows the expected relative costs of call graph construction algorithms. CHA costs more than RTA since the call graphs are roughly a factor of 10 bigger. We cannot argue that CHA call graph construction should be free, implicitly complete when the class hierarchy is built; we must explicitly traverse the graph to determine which class initializers execute, and to allow the implementation to handle reflection as described earlier.

In all cases, analysis with 0-CFA or RTA completes in less than three minutes, with RTA often around a factor of two faster than 0-CFA. Call graph construction dominates the overall performance; we plan to investigate more sophisticated constraint solver techniques [13, 1] to mitigate this overhead. For smaller codes, initial input can be a significant cost; we could potentially pre-process libraries to cut down this overhead.

7. RELATED WORK

This work follows-on to an earlier IBM WebSphere tool called CmpOPT [14]. CmpOPT performed a whole-program analysis to identify CMP Entity Beans that were accessed read-only, and would generate a report accordingly. CmpOPT did not analyze transaction boundaries or consider scoped units of work. The analysis did not model enough aspects of the J2EE and EJB models to consider whole J2EE applications, and was instead limited to individual EJB Jar files. CmpOPT relied on a fairly precise, context-sensitive call graph construction algorithm described in [18]. The resultant performance was much slower than the most expensive analysis reported here. The CmpOPT analysis employed an extra level of context sensitivity for container objects, which motivated our consideration of 0-1-C-CFA.

The IPA infrastructure of CmpOPT was reused in an IBM tool called SABER [22], which targets validation of J2EE "best practices". The SABER tool does not analyze container-managed persistence or J2EE transactions, as it currently focuses primarily on the Web Application layer of J2EE. One area for future work is to determine whether the relatively less precise but more efficient call graph construction algorithms we have evaluated suffice for SABER clients. Additionally, we plan to leverage our analysis to consider validation of transaction-related properties, such as legality of isolation level directives, and to prevent deadlock or excessive rollbacks.

We are not aware of any other published results of interprocedural analysis of the J2EE framework. Clarke et al.[5] presented a verification tool for checking confinement properties of EJB implementations. Like our work, this tool analyzed combined properties inferred from the deployment descriptor and bytecode business logic. The confinement verification problem was solved with a simple, local pass, but did not require any deep flow analysis.

Our call graph construction implementation resembles other subset-based pointer analysis/call graph construction algorithms for Java, such as [19], [28], [24], and [20]. Our design was heavily influenced by Grove et al.'s [12] approach for parameterizing algorithms for varied context-sensitivity

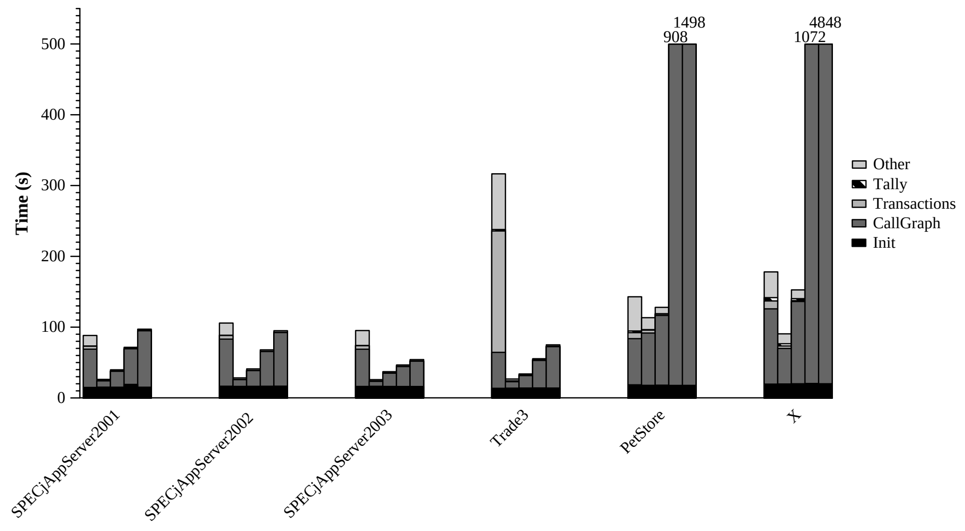

Figure 7: Running time, in wall clock seconds, of the analysis as a standalone tool. For each benchmark, the five bars represent CHA, RTA, 0-CFA, 0-1-CFA, and 0-1-C-CFA, in order from left to right.

policies.

8. CONCLUSION AND FUTURE WORK

This tool represents our first attempt to apply state-of-the-art inter-procedural analysis to reduce complexity in J2EE configuration. The results appear encouraging, as despite its limitations, static analysis provides useful suggestions for transaction configuration. However, much work remains to do, even for the limited configuration tasks this paper addresses.

In future work, we plan to investigate other J2EE transaction configuration knobs. Specifically, we plan to investigate opportunities to suggest pre-fetch and caching attributes, pessimistic vs. optimistic concurrency options, and to prevent deadlock, unnecessary rollbacks, and unintended runtime exceptions from violations of EJB contracts. More generally, we believe that similar techniques may apply for configuration of other high-level programming models embedded in Java, such as XML manipulation frameworks, Web Services, and the "next generation" of enterprise programming models which will undoubtedly arise soon.

9. REFERENCES

[1] Alexander Aiken, Manuel Fahndrich, Jeffery S. Foster, and Zhendong Su. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, May 1998.

[2] L. O. Anderson. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324–??, October 1996.

[4] BEA. BEA Systems. http://www.bea.com.

[5] Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad beans: deployment-time confinement checking. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 374–387. ACM Press, 2003.

[6] Gennaro Cuomo and Catherine Diep. The dynamic caching services: eliminate bottlenecks and improve response time. *WebSphere Developer's Journal*, February 2003.

[7] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[8] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *9th European Conference on Object-Oriented Programming*, pages 77–101, August 1995.

[9] The Apache Software Foundation. The Apache Struts Web Application Framework. http://jakarta.apache.org/.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[11] David Grove. *Effective Interprocedural Optimization of Object-Oriented Languages*. PhD thesis, University of Washington, 1998.

[12] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Transactions*

on Programming Languages and Systems, 23(6):685–746, November 2001.

[13] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of C code in a second. *ACM SIGPLAN Notices*, 36(5):254–263, May 2001.

[14] Matt Hogstrom. Optimizing container-managed persistence EJB entity beans. *IBM DeveloperWorks Newsletter*, November 2001. http://www-106.ibm.com/developerworks/ibm/library/i-optimize.

[15] IBM. Trade3 benchmark. http://gwareview.software.ibm.com/ software/ webservers/ appserv/ benchmark3.html.

[16] IBM. WebSphere Software Platform. http://www.ibm.com/websphere.

[17] American National Standards Intitute. *Database Language SQL*. 1992. ANSI X3.135-1992.

[18] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for Java. *ACM SIGPLAN Notices*, 37(11):359–372, November 2002.

[19] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[20] Donglin Liang, Maikel Pennings, and Mary Jean Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for Java. In ACM, editor, *ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering: June 18–19, 2001, Snowbird, Utah, USA: PASTE'01*, pages 73–79, New York, NY, USA, 2001. ACM Press. Supplement to ACM SIGPLAN Notices.

[21] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.1.* November 2003.

[22] Darrell Reimer, Edith Schonberg, Kavitha Srinivas, Harini Srinivasan, Julian Dolby, Aaron Kershenbaum, and Larry Koved. Validating Structural Properties of Nested Objects. submitted for publication, 2004.

[23] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In ACM, editor, *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: San Francisco, California, January 22–25, 1995*, pages 49–61, New York, NY, USA, 1995. ACM Press. ACM order number: 549950.

[24] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. *ACM SIGPLAN Notices*, 36(11):43–55, November 2001. Proceedings of the 2001 ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA'01).

[25] M. Sharir and A. Pneuli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–223, Englewood Cliffs, NJ, 1981. Prentice-Hall.

[26] O. Shivers. Control flow analysis in scheme. *ACM SIGPLAN Notices*, 23(7):164–174, July 1988.

[27] The Standard Performance Evaluation Corporation. SPECjAppServer. http://www.specbench.org/jAppServer/.

[28] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, September 2002.