Persistent Threads

Florian Matthes Joachim W. Schmidt

Universität Hamburg
Vogt-Kölln Straße 30
D-22527 Hamburg, Germany
{matthes,J_Schmidt}@dbis1.informatik.uni-hamburg.de

Abstract

Persistent threads are a database programming concept particularly well-suited for applications that manage long-term, distributed or cooperative activities. We introduce persistent threads as a novel form of bindings from data in persistent object stores to activated code and relate them to existing binding concepts found in database programming. We also describe the integration of persistent threads into a polymorphically-typed database language and its supporting layered system architecture with particular emphasis on abstractly-defined thread representations which support thread analysis, optimization and portability.

1 Introduction

A noticeable trend in database research and database system development is an increased interest in behavioral and procedural aspects of information systems. Data models that describe dynamic processes in addition to static data structures are capable of capturing more of the application semantics as exemplified by object-oriented models like Taxis, ADAPLEX, Galileo or Fibonacci. Similarly, database systems that support procedures, methods, rules or triggers in addition to persistent data values are capable of factoring-out procedural code from individual application programs into shared databases. The positive effects of eliminating integrity checking, database event detection, exception handling, or user interface management code from application programs on the overall system consistency and application programmers' efficiency have been described amply in the literature. Consequently, one can perceive a shift from passive data stores to more active persistent object systems.

In this paper we focus on the intricate binding issues on the borderline between "active code" and "passive data" that arise in persistent object systems as soon as one gives up the classical separation between short-lived transactions expressed in a host language and long-lived data stored in a database. More specifically, we provide a classification scheme for bindings from code to persistent data and vice versa and we investigate bindings from persistent data and code to threads, which exhibit an interesting duality. On the one hand side threads can be viewed as activities that can be created, executed, synchronized, suspended, terminated, etc. Alternatively, they can be viewed as passive data that can be stored persistently, annotated with attributes, associated with other persistent data structures, moved between nodes in a network, and manipulated by computations.

This duality makes persistent threads a key technology for future persistent object systems and for applications that manage long-term, distributed or cooperative activities like computer-aided design, workgroup communication and workflow management. Unfortunately, this duality also invalidates many design assumptions on which today's volatile thread implementations are based. Therefore, we also discuss in some detail implementation aspects of persistent thread bindings.

The specific persistent thread model presented here has been developed within the context of the persistent programming environment Tycoon[1] [MS93, Mat93] where it serves as an abstract model to shield pro-

**Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994**

[1]TYped Communicating Objects in Open eNvironments.

grammers and application-oriented tools from details of the underlying system implementation (object store, database language evaluator, scheduler, recovery subsystem) while being sufficiently low-level to support a variety of (possibly application-defined) scheduling and activity management strategies.

The paper is organized as follows: In section 2 we introduce a terminology for the description of bindings to data, code and threads in persistent systems which we use throughout the paper. We then review briefly the evolution of database languages in terms of their binding patterns between data, code and threads. We argue that the expressiveness and usability of a database system model is related directly to the orthogonality of its binding patterns. In section 4 we propose a next step in the evolution of database languages by introducing the concept of first-class bindings to threads. We argue that an explicit and orthogonal handling of threads naturally leads to the concept of persistent threads, a systems abstraction suitable for novel high-level models for long-term activity management. Finally, in section 5 we report on our implementation of persistent threads in the Tycoon system based on an abstractly-defined store protocol, code representation and thread semantics.

2 Bindings in persistent systems

In this section we introduce a terminology for the description of bindings to data, code and threads in persistent systems which we will use throughout the paper.

A binding is an association between a name and a computational entity from a specific semantic domain [Str67]. We also say that a name is bound to a computational entity. An environment is a (possibly ordered) collection of bindings. Names are used to identify entities in an environment. Different names can be bound to the same entity (sharing, aliasing). The details of this identification process (static scoping, dynamic scoping, user-defined conflict resolution) and mechanisms to manipulate environments (import/export, inheritance, record extension, imperative update) are irrelevant for the purpose of this paper.

Entities can be atomic (like integers or booleans) or structured (like records, objects or functions). Structured entities typically consist of environments. For example, the fields of a record lead to bindings from field names to other entities. Therefore, bindings can be used to model (recursive) relationships between entities.

Entities can be flat (like records) or nested (like functions in Algol-like languages). In a nested entity, names bound in a global outer environment are automatically visible in a local inner environment. As will be seen in sections 3.1 and 3.3, the semantics of bindings from and to (dynamically) nested entities requires particular attention.

Entities can be transient (like local program variables) or persistent (like database tables). Virtually all database systems restrict environments of persistent entities to contain only bindings to other persistent entities since bindings to volatile entities would lead to "dangling references". Such constraint violations are avoided in many systems by a transitive reachability rule: Every entity reachable from a persistent entity becomes persistent, too. In reachability-based systems, there is a so-called "persistent root environment", for example, the set of all globally-defined database names in O_2 [BDK92]. An entity is made persistent by making it reachable through chains of bindings (e.g., database definition, class extent) starting from this persistent root environment.

The following three categories of structured entities are of particular interest in extended (higher-order) database modeling:

Persistent Data (D) describe the persistent state of an information system by a collection of computational entities related through bindings. The structure (types) of the persistent entities and their bindings are described by a database schema.

```
persDB=database
  peter=[age=30, married=true, boss=NULL],
  paul=[age=30, married=true, boss=persDB.peter], ...
  persons={persDB.peter, persDB.paul, ...}
end
```

In this example, the name *persDB* is bound to a database, i.e. a persistent environment that stores bindings for the database variable names *peter, paul* and *persons*. For example, *paul* is a name bound to a record (an environment with three bindings). One of these bindings associates the name *boss* to the record identified by the name *peter* within the environment *persDB*. The set bound to the name *persons* defines an environment of anonymous bindings.

Code (C) is a description of operation sequences that query and update volatile or persistent entities and bindings.

```
procedure changeBoss(pers:Pers) =
  pers.boss:=persDB.paul;
transaction changeAll() =
  for each p in persDB.persons do changeBoss(p);
```

Code involves names to describe bindings to other code (*changeBoss* referenced in the body of *changeAll*), bindings to persistent data (*persDB.persons*, *persDB.-paul*), and bindings to volatile data (*p*, *pers*). In statically-scoped languages, the binding of a name in a code fragment to a matching name in its environment is determined by a textual analysis of the code and of the database schema.

Conceptual Model	Language Model	Implementation Model
entity	variable	data
behavior	function	code
activity	continuation	thread
relationship	name	binding

Figure 1: Corresponding notions at different levels of conceptualization

Approach	Bindings	Description[2]
database programming	C→D	code bound to data
object-oriented databases	D→C	persistent data bound to code
transactional programming	T→C	threads bound to code
activity management	D→T	persistent data bound to threads

Figure 2: Predominant binding patterns (see text)

A Thread (T) is a representation of code *in the process of being executed*. A thread describes a single sequential flow of control in a program. Having multiple threads in a program means that at any instant the program has multiple points of execution, one in each of its threads. Unlike operating system processes, multiple threads can execute within a single (persistent) address space, permitting multiple threads to access shared variables in addition to local variables.

It may be helpful to compare our terminology with corresponding notions in language models and conceptual models as summarized in figure 1. Since in this paper we are interested also in implementation aspects of persistent threads, we are using a rather system-oriented terminology. In our setting (as opposed to, for example, visual programming), the correspondence between high-level conceptual notions like entities, behavior, activities, relationships and their system counterparts (data, code, threads, bindings) is often established indirectly via formal language models expressed in terms of variables, functions, continuations and names bound in scopes. Our system argument that threads should be treated as first-class persistent data could therefore be rephrased in high-level models by requiring activities to be viewed as first-class entities that can participate freely in abstractions like aggregation and classification.

A thread is created by submitting a parameterless code fragment (e.g., the body of the transaction *changeAll*) and (persistent) data (e.g., *persDB*) to an evaluator (**eval** *(changeAll, persDB)*). As described in section 5 the semantics of the evaluator can be defined inductively by rules that map thread states to thread states and that perform side-effects on data. A thread state subsumes bindings to the code fragments currently being executed and a dynamic environment that records the current bindings from names occurring in the code to local and global entities. In most programming and query language implementations thread states are represented as records that reference stacks of so-called "activation records", one for each function or query invocation.

The following thread state describes a snapshot of the execution of the transaction *changeAll* against the database *persDB*. More precisely, it describes the state of the transaction while executing the function *changeBoss* during the first iteration of the **for each** loop, immediately preceding the assignment of the value *persDB.paul* to the field *boss*.

```
thread1 = [result = persDB.paul, dynamicContext = [
  code=(pers.boss:= result),
  localEnv=[pers=persDB.peter],
  globalEnv=[persDB=database...end],
  dynamicContext=[code=(for each p in toVisit
      do changeBoss(p)),
    localEnv=[p=persDB.peter,
      toVisit={persDB.paul,...}],
    globalEnv=[persDB=database...end,
      changeBoss=procedure ...],
    dynamicContext = [] ] ] ]
```

In this example the thread state consists of the result of the current subexpression (the right-hand-side of the assignment, i.e., *persDB.paul*) and a dynamic context (*continuation*) that describes the code still to be executed together with the bindings valid within this code. The next instruction to be executed is the assignment *(pers.boss:= result)*. The binding of the parameter name *pers* can be obtained from the local environment which has been established on entry to the function *changeBoss*. The dynamic context of the function (the state of its "caller", i.e. *changeBoss*) is captured also by a continuation. On function return, evaluation continues with the bindings of this continuation. The local variable *toVisit* is used to control the iteration. Since the dynamic context of the transaction *changeAll* is empty, the thread will terminate on return from this transaction.

Since entities of each of the three categories above –

[2] X bound to Y means that names in entities of category X are bound to entities of category Y, i.e. that the semantics of X depend on the semantics of Y.

data, code and threads – may contain bindings, there are nine possible binding patterns between computational entities in fully orthogonal object systems. In figure 2 we list four of these binding patterns which we regard as "historic" milestones in the development of persistent system models:

- ▷ In database programming languages it is possible to write algorithmically-complete code that establishes C→D bindings to persistent data and that modifies the state of persistent entities and their relationships expressed as D→D bindings. However, code is still separated strictly from persistent data in the sense that reverse bindings are not supported.
- ▷ This restriction is lifted in object-oriented databases where D→C bindings extend the semantics of data entities to also include code fragments (method code) which express behavioral aspects. Similarly, in active databases it is possible to attach code (conditions and actions) as triggers to persistent data (relations, classes).
- ▷ In multi-user (database) systems there are multiple concurrent user sessions accessing shared data via transactional code. Each active transaction corresponds in our terminology to a single thread that is bound (via a T→C binding) to application code which in turn is bound to shared (persistent) data. However, threads are strictly separated from persistent data and code in the sense that a thread cannot access (query, store, update) the set of code or data bindings held by itself or by other threads.
- ▷ In this paper we argue that threads understood as dynamic environments of bindings are highly relevant for novel, activity-oriented applications and, therefore, should gain first-class status in future database models and not be hidden behind a specific built-in binding pattern (transactions). As detailed in section 4.3, mechanisms to establish bindings from code, from data, and from threads to threads are very helpful to manage co-operative and distributed activities. In particular, D→T bindings from names in persistent data to threads naturally lead to the concept of persistent threads in reachability-based persistent systems, which we regard as highly relevant to *long-term activity management*.

The semantics and implementation of pure D→D bindings like the binding of *persDB.paul.boss* to *persDB.peter* in *persDB* and of pure C→C bindings like the binding of the name *changeBoss* in the function *changeAll* are sufficiently well understood that we restrict ourselves in the following discussion to the binding patterns highlighted in figure 2.

3 From data-oriented modeling to object-orientation

In this section we review briefly the evolution of database languages in terms of their binding patterns. We argue that the expressiveness and usability of a database system model is related directly to the orthogonality of its binding patterns and that, in retrospect, many ad-hoc binding restrictions found in database systems are simply based on the choice of an improper implementation technology.

3.1 Binding names in code to persistent data

All database management systems with a programming language interface support C→D bindings. In a third-generation language, a C→D binding to a database is established at program run-time using an explicit operation similar to the SQL *connect* statement. C→D bindings to individual elements in a database collection are established using explicit cursor manipulation operations, typically embedded into program loops.

In fourth-generation languages like Ingres/Windows 4GL or PL/SQL and in database programming languages like DBPL [SM94] or E [RCS93], the outermost program environment already contains bindings to persistent entities which are therefore directly accessible in statements and expressions. Moreover, these languages provide bulk data types [MS91] with operations that work on collections of (anonymous) bindings at once. As a consequence, programming with persistent bulk data in these languages is as effortless as programming with volatile data in 3GLs.

3.2 Binding names in persistent data to flat code

An object in an object-oriented database can be modeled as an environment of bindings and a (hidden) object identifier.

```
peterObj:PersonObject=object age=30, married=true,
   boss=NULL,
   changeBoss=method(newBoss:PersonObject)
      self.boss:=newBoss
end
```

Attributes like *age*, *married* or *boss* lead to standard D→D bindings to persistent state variables while a method definition is a D→C binding from a method name (*changeBoss*) to a code fragment (**self**.*boss*:= *newBoss*). In most object-oriented models, a method is bound in an environment attached to an object class; however, some systems (e.g., O_2) also support so-called "exceptional objects" where methods can be overridden by bindings established on a per-object basis. Message names in code, like *changeBoss* in the dot expression *peterObj.changeBoss(...)*, are bound dynamically to matching method code identified by D→C bindings attached to the object itself (*peterObj*), its class (*PersonObject*) or its transitive superclasses.

D→C bindings are also supported by active database systems where it is possible to bind a trigger consisting of a condition (a code fragment that returns a boolean value) and an action (a code fragment that performs a side-effect) to a persistent data structure (typically a global collection variable). Views as attribute values in Postgres and "viewers" as proposed in [SMR+93] are a third form of D→C bindings that attach code fragments (returning bulk data values) to individual persistent data items.

D→C bindings add a new dimension to data modeling since it becomes possible to attach behavior to shared and persistent data and to adopt a data-centristic execution model. In this view, the application logic is no longer hard-wired statically in "structured" application code that drives the passive database system via read/write instructions. Instead of this, the application logic can be divided into semantically rich and loosely coupled conceptual classes attached to persistent data structures, and the application system is "driven" by messages dispatched dynamically by the DBMS.

In all systems mentioned so far, the code participating in a D→C binding has to come from a *flat* environment. For example, the object-oriented programming languages Eiffel, C++, Modula-3 and Trellis as well as the object-oriented database systems ObjectStore and O_2 have syntax and scope rules that make it impossible to bind a function that is nested within another function as a method to a database object. As a consequence, the only bindings available inside method code are static global D→D or D→C bindings, and dynamic bindings established via explicit method arguments (*newBoss*), and the dynamic binding of the distinguished identifier **self** to the receiver of the message. Analogous restrictions hold for stored database procedures written in fourth-generation languages and triggers in active databases.

The rationale behind these restrictions is to simplify the implementation of D→C bindings.

A more elegant implementation of D→C bindings to flat code is achieved in object-oriented database systems like O_2 that manage executable code in the object store itself. In these systems D→C and D→D bindings are implemented uniformly as intra-object-store bindings exploiting the concept of persistent object identity.

3.3 Binding names in persistent data to nested code

The increased modeling power of orthogonal D→C bindings that also handle *nested* code correctly has been demonstrated by higher-order programming languages like Lisp, Scheme, Standard ML, Dylan and Haskell (higher-order functions) but also by Smalltalk (first-class blocks) and CLOS. As a consequence there is a clear evolution in the family of higher-order database languages from PS-algol [AM85], Napier88 [DCBM89], P-Quest [MMS92], Fibonacci [AGO91] to Tycoon [MS92] allowing programmers to treat functions, procedures and transactions as first-class computational entities that can be passed as arguments, returned as results, and embedded into persistent data structures.

Here, we focus on the semantics of D→C bindings to nested code and do not discuss the relative advantages of full higher-order models over plain object-oriented models (see [SM93]). The following simple example shows a parameterized transaction *disallowBoss* that overrides an existing method binding defined for the message *changeBoss* of a person object *thisPerson*. The new D→C binding relates *changeBoss* to a nested method code fragment that depends on the parameter value *thisBoss* of its enclosing transaction.

```
transaction disallowBoss(
          thisPerson,thisBoss:PersonObject)=begin
 let oldMethod=thisPerson.changeBoss
 thisPerson.changeBoss:=method(newBoss:PersonObject)
     if newBoss!=thisBoss then oldMethod(newBoss)
     else raise illegalBossException end
end
```

This transaction can be called with person objects as arguments, for example, to raise an exception if *paulObj* is to become the boss of *peterObj* or if an attempt is made to delete the boss of *paulObj*:

```
disallowBoss(peterObj, paulObj)
disallowBoss(paulObj, NULL)
```

A correct representation of the D→C binding for the name *changeBoss* has to consist not only of a binding to the nested method code but it has also to record the environment of global C→D bindings valid for the nested code (bindings for the parameter *thisBoss* and the variable *oldMethod* of the enclosing transaction). Such a [code, environment]-pair is called a (function) *closure*. The two transaction calls above yield the following closures:

```
peterObj.changeBoss=[
  code=(if newBoss!=thisBoss then ... end),
  globalEnv=[thisBoss=paulObj,oldMethod=...]]
paulObj.changeBoss=[
  code=(if newBoss!=thisBoss then ... end),
  globalEnv=[thisBoss=NULL,oldMethod=...]]
```

The code bound to *peterObj.changeBoss.code* and *paulObj.changeBoss.code* is shared, however, since the global environments differ the execution of the method code in these different environments has different semantics.

4 Activity-oriented programming with persistent threads

Based on the discussion of the previous section we propose a next step in the evolution of database languages by introducing the concept of first-class bindings to threads. The concept of threads (continuations, sessions, running transactions) is either non-existent or only implicitly available in today's database systems. We argue that an explicit and orthogonal handling of threads naturally leads to the concept of persistent threads, a systems abstraction suitable for novel high-level models for long-term activity management.

4.1 Threads in persistent systems

As mentioned in section 2 (see also figure 2), threads and T$\rightarrow$C bindings already exist implicitly in transactional multi-user DBMSs. For example, running transactions in a DBMS correspond to isolated threads (bound to transaction code) that are activated, suspended or aborted under the control of a centralized scheduling "master thread" that gains control whenever these threads access shared database entities. On an implementation-level, a transaction descriptor in the scheduling subsystem is a record that aggregates bindings to a (suspended or running) thread with additional information relevant for synchronization purposes, like the shared (read-only) bindings and the exclusive (updated) bindings held by the thread, bindings to other threads waiting for resources of this thread, or the cost of the operations executed by the thread so far. Database systems (like Ingres) that support named checkpoints inside transactions provide an additional mechanism to store multiple "frozen" thread states that can be reactivated at the user's discretion.

Novel transaction models (see, e.g., [BK91, GR93]) propose to give "power users" the ability to extend the semantics of the "master thread" by introducing additional scheduling concepts like lock modes and by triggering the execution of user-defined code fragments whenever two threads access a shared persistent object concurrently. This code can use an algorithmically-complete language to decide when to abort, suspend or notify the conflicting threads.

While threads managed by a DBMS scheduler are volatile (they are limited to the lifetime of their corresponding operating system processes), a limited form of persistent threads can be found in the (single-user) persistent higher-order systems Napier88 [DCBM89] and PQuest [MMS92] that have an atomic *stabilize* operation. This operation can be called anywhere inside a program to define a consistent persistent system state. This state not only consists of the global database state variables but also includes the state of the program evaluator. If a system crash occurs during program execution, the execution can be resumed in the state valid at the last *stabilize* operation.

```
procedure p(x:Int) =
 begin stabilize(), print("leave p, x=", x) end
procedure main() =
 begin print("call p"), p(3), print("; end") end
```

For example, assuming that during the execution of *main* the system crashes after returning from *p*. On system restart, program execution would resume with the first statement after the *stabilize* operation, and the output would be *leave p, x=3; end.*

To summarize, these systems already have a hidden persistent thread functionality that is, however, severely limited to a single top level thread.

4.2 Threads as first-class persistent entities

In this section we illustrate how threads fit as first-class computational entities into persistent object systems. Our presentation is based on the thread abstraction available in the Tycoon system, a polymorphic persistent programming environment developed in the FIDE project by our group at Hamburg University [MS93, Mat93].

In Tycoon, computational entities (data, code, threads) are either bound in the scope of individual programs or in the scope of persistent modules. The execution of a Tycoon program code *c* invoked from an operating-system shell leads to the creation and execution of a Tycoon thread bound to *c* in an initial environment that contains C$\rightarrow$D bindings from all module names imported by *c* to corresponding linked persistent module values.

Users or applications at the operating-system level can create independent Tycoon threads running against a shared set of persistent modules. Conceptually, the computational entities of all Tycoon threads and modules reside in a common persistent object store. This seamless integration of volatile and persistent store simplifies the access to databases represented as persistent modules. Furthermore, it facilitates the exchange of data, code and threads between threads via shared variables. These variables are bound in the scope of persistent modules and are typically protected by synchronization mechanisms like transactional locks, semaphores, monitors, or message queues.

The following simplified excerpt of the Tycoon system library interface *Thread* defines the basic functionality of a corresponding module *thread* which exports a parameterized abstract data type *thread.T* and operations to inspect, create and execute multiple threads from within Tycoon programs.

```
interface Thread export
 T(R <:Ok) <:Ok
  (* T(R) is the type of threads that on termination
     return values of type R. *)
 Let State = Tuple case suspended, running,
     terminated, aborted, blocked end
  (* An enumeration of the possible thread states. *)
 new(I,R <:Ok  code :Fun(:I):R  input :I) :T(R)
  (* Return a new, suspended thread to execute
     code(input). *)
 self(R <:Ok) :T(R)
  (* Return the current thread. *)
 copy(R <:Ok  thread :T(R)) :T(R)
  (* Return a shallow copy of thread. The execution
     of thread does not affect copy(thread).
     However, entities bound in the code executed by
     thread and copy(thread) are shared. *)
 run(R <:Ok  thread :T(R)) :Ok
  (* If thread is suspended then resume execution
     until execution terminates, aborts with an exception,
     is suspended or blocked. thread and self() execute
     concurrently. *)
 state(R <:Ok  thread :T(R)) :State
  (* Return the state of thread that may change
     dynamically if running or suspended. *)
 join(R <:Ok thread :T(R)) :R
  (* Block until thread terminates or aborts. Return
     the result or propagate its exception. *)
end
```

The module *thread* encapsulates the representation of threads, the semantics of the Tycoon evaluator (*thread.run*, see section 5) and the details of the mapping from threads to physical processing units. Since some versions of the Tycoon system are based on persistent object stores which allow multiple workstations to access a Tycoon object store concurrently, multiple physical processing units (workstations) may be involved in thread execution.[3]

The type *thread.T* and all functions exported from the interface above are polymorphic, i.e. they have an explicit type parameter *R* that has to be instantiated with a subtype of the trivial "top type" **Ok** [MS92]. Threads are polymorphic data structures since they can describe the execution of code that returns values of an arbitrary result type *R*. This type *R* has to match the return type of the *code* function passed as an argument to the *thread.new* function.

As a minimal example of (volatile) thread programming, the following Tycoon program creates a thread *t* to evaluate the function *code* that returns a value of type *Int*. This result is computed by adding the statically-bound *data* value and the dynamically-bound *parameter* value.

```
import thread :Thread
let data :Int = 3
let code(parameter :Int) :Int = data + parameter
let t :thread.T(Int) = thread.new(:Int :Int code 4)
thread.run(:Int t)
let result :Int = thread.join(t)   (* ⇒  7 *)
```

Remember that the thread bound to *t* is a first-class entity in Tycoon – it can be passed as a function argument, returned from a function, bound to a name in the scope of a persistent module, exported to a portable data file, or sent across a communication channel.

The generalization of first-class threads from volatile to persistent has the following semantic implications:

- ▷ The definition of persistence has to be revised. In addition to all persistent modules also all active threads act as "roots of persistence". Moreover, the transitive reachability rule introduced in section 2 has to be extended to also include T→C, T→D and T→T bindings.
- ▷ The semantics of the shallow and deep copy operation has to be extended properly to values of type *thread.T*.
- ▷ In activity-intensive applications, it is desirable to be able to attach additional information to thread values (user id, transaction group id, access rights, authentication key, parent thread, ...). This extensibility is achieved in Tycoon by adding a second type parameter *D* to the thread signature. A value of type *thread.T(D R)* is a thread that computes a value of type *R* and that has a descriptor attribute of type *D*. Descriptors are exploited heavily by higher-level scheduling and activity-management algorithms but can also be made visible to application-level code.

4.3 Programming with persistent threads

Having threads as computational entities in a database language, programmers can benefit from the potential of multi-threaded programming [Nel91], like

- ▷ a better exploitation of existing processing resources (e.g., workstation clusters or multiprocessor workstations);
- ▷ a better support for multiple activities of human users within a single application (e.g., a database query tool can process multiple independent queries);
- ▷ a reduced latency of operations by deferring reorganization tasks (e.g., an insertion into an index structure returns control to the caller immediately and spawns a separate thread to perform the index reorganization if necessary);

[3]Currently, a call *thread.run(:R t)* executes *t* and *self()* on the same processing unit.

▷ a better responsiveness of servers in distributed systems by allocating multiple server threads to handle client requests.

By making a thread reachable from persistent data (persistent modules) and by checkpointing the state of the persistent store it becomes possible to protect long-running activities from system failures. After a crash, the thread can be restarted explicitly in the state that was valid at the last checkpoint.

More importantly, persistent threads support directly an activity-oriented style of information system modeling as promoted by scripts in Taxis [BMS93], by process-centered specifications in Estelle, Lotos or SDL [Tur93], or by (visual) process languages of work flow management tools like Regatta [Swe93].

As a highly simplified example, a paper submitted to a workshop can be represented by the following data type that contains a (persistent) thread attribute:

```
Let Paper=Tuple title,authors,abstract,text :String
  reviewer:Person  rating:Rating
  refereeActivity:thread.T(Ok)
end
```

From the viewpoint of the PC chair, each paper has to be reviewed individually, and the set of submitted papers has to undergo the following reviewing activity.

```
for each p in db.submissions do
   p.refereeActivity:= thread.new(:Paper :Ok
      reviewPaper p)
   thread.run(:Ok p.refereeActivity)
end
joinAll(select p.refereeActivity from p in db.submissions)
```

Reviewing the set of all submitted papers is modeled by creating and executing one *p.refereeActivity* per paper *p* which can then run concurrently without interference. Standard query language notations can be used to perform bulk operations on sets of threads, for example, to wait until all threads attached to *db.submissions* have terminated. The user-defined function *joinAll* takes a sets of threads and blocks the current thread until all of the threads have terminated.

```
let joinAll(threads :set.T(thread.T(Ok))) :Ok =
  for each t in threads do thread.join(:Ok t)
end
```

The activity of individual reviewer assignment and review recording is modeled by the following code:

```
let reviewPaper(p :Paper) :Ok = begin
 repeat p.reviewer:=chooseReviewer(availableReviewers)
 until acceptedByReviewer(p.reviewer p)
 sendPaperToReviewer(p.reviewer p)
 try p.rating:= waitForReview(p.reviewer)
 when reviewerNotAvailableExc then reviewPaper(p)
 end
end
```

Contrary to current database practice, the progress of the reviewing process is not captured by a passive relational table that stores a *state* attribute for each paper which is then updated by separate transactions to values like *unassigned*, *assigned*, *sentOut*, *returned*, ... Instead of this, an activity- and goal-oriented script modeled by database language code describes directly the possible states and state transitions. This example makes use of several control structures (loop, recursion, exception handling) for sequential activities and uses threads for long-term concurrent activites. Note that the thread above depends crucially on global bindings (to parameter values, global data, and global code).

In this example, thread synchronization has to be employed to coordinate parallel activities (the assignment of reviewers to individual papers that implies access to the shared variable *availableReviewers*) and to wait for the termination of subactivities.

Persistent threads do not lead necessarily to an imperative, deterministic style of activity management. Instead of this, higher-level activity models can be supported directly by factoring-out synchronization tasks (*parallelDo*, *tryOneOf*, *atomicDo*, *compensatingDo*) from applications into higher-order thread libraries.

5 On thread implementation and formalization

Several important requirements on persistent thread implementations differ substantially from volatile thread implementations, for example:

1. Thread state representations have to be made relocatable and portable in the sense that states need to be abstractly interpretable without reference to a specific machine architecture (e.g., SPARC register files).
2. It is desirable to have automatic garbage collection that reclaims the storage of terminated or orphaned threads.
3. It is necessary to formalize in sufficient detail the effects of thread execution on shared entities in the persistent store, in particular, if these entities have a complex structure. Only then reliable support can be provided for concurrency, recovery, or garbage collection.
4. Store access in distributed persistent memory has different performance characteristics than store access in centralized shared-memory architectures.
5. In data-intensive applications, the thread state can vary dramatically in size, for example, to accommodate the numerous temporary bindings that arise during query evaluation. Clearly, simple thread implementations based on fixed-sized stacks are not acceptable.
6. Built-in schedulers have to be able to work with a number of persistent threads that may exceed

the number of volatile threads by two to three numbers of magnitude.

In this section, we first report on our implementation of persistent threads focusing on the issues (1) through (4) since the remaining investigations are beyond the scope of this paper. We then sketch thread formalization with a clear emphasis on abstract representations of machine code (Tycoon Machine Language, TML), machine states and on explicit modeling of machine-store interactions (Tycoon Store Protocol, TSP).

In our experience, a formalization on an appropriate level of abstraction and with an intensive flavor of "constructivity" is absolutely essential for any good implementation of a conceptually rich system abstraction, such as persistent threads.

5.1 The Tycoon thread implementation

The Tycoon thread implementation is divided into subtasks solved by three distinct layers of the Tycoon system architecture:

1. The Tycoon compiler front end performs the type checking and code generation of application programs. Due to the polymorphic nature of Tycoon's type system, no extensions of the Tycoon compiler front end are required to support user-defined type constructors like *thread.T(R)* described in section 4.2.
   Tycoon uses a uniform (tagged) polymorphic data representation. Therefore, no modifications to the Tycoon code generator are required to support operations on first-class persistent threads. The binding of the operations *thread.new*, *thread.run*, etc. to processor-specific compiled C code implementing thread creation, thread execution etc. is accomplished by standard Tycoon language mechanisms.
2. The Tycoon compiler generates abstract machine code (TML). For every hardware architecture there is a separate Tycoon evaluator, implemented as an interpreter or a pair of a target object code generator and a runtime library that dynamically loads the target code into the process address space. The execution of Tycoon threads is performed by TML evaluators that read code held in the Tycoon object store and that are able to store their evaluation state in a portable format in the object store. Thereby, it is possible to exchange suspended evaluator states between different hardware architectures and to represent thread bindings by standard object store identifiers with the usual sharing semantics.
3. The Tycoon object store allows evaluators to abstract from the lifetime and storage details of all

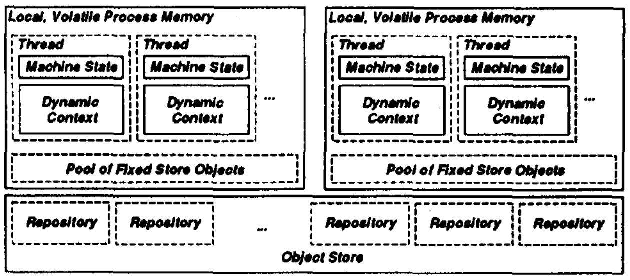

Figure 3: Interaction between TML threads

computational entities (data, code, threads). The store encapsulates buffer management, garbage collection, cache coherence management and recovery management. Tycoon evaluators access the Tycoon store via an abstract store protocol (TSP) implemented as a collection of standardized C functions.

Figure 3 shows the interaction between multiple Tycoon threads accessing a shared persistent object store (possibly partitioned into disjoint repositories). It shows two operating-system processes, each executing a TML evaluator that manages a set of Tycoon threads. Thread states consist of a machine state (a register set) and a dynamic context of variable size. Active threads are cached in local process memory. Furthermore, the object store permits TML evaluators to fix (pin) persistent objects in local process memory. It is the object store's responsibility to ensure the coherence between thread states and other persistent objects cached in multiple process address spaces.

5.2 On thread formalization

A thread formalization has to define invariants maintained by the scheduling operations exported by the interface *Thread* (see section 4.3) and it has to specify the semantics of individual threads by an inductive definition of the bindings and store side-effects performed for each instruction executed by a thread. The first issue has been treated already in the literature, for example, chapter 5 of [Nel91] gives a complete Larch specification of the Modula-3 thread package that is very similar to Tycoon's *Thread* interface. Here we concentrate on the second issue. The TML/TSP specification sketched in the remainder of this section not only affects the granularity level on which thread operations (e.g. *run*, *join*) are performed, but also the degree to which further requirements on threads as enumerated at the beginning of section 5 can be supported.

Tycoon's RISC-style TML code representation is shown in figure 4. This instruction set suffices to implement the full Tycoon language as defined in [MS92] (and similar higher-order programming languages like Fibonacci or Napier88). We are currently moving to an

$c ::= \mathbf{nop}$	no operation
$\mid \mathbf{imm}(b)$	base value access
$\mid \mathbf{lit}_i$	literal value access
$\mid \mathbf{loc}_i$	local variable access
$\mid \mathbf{par}_i$	parameter value access
$\mid \mathbf{glb}_i$	function closure value access
$\mid c_1[c_2]$	object store variable access
$\mid \mathbf{loc}_i \leftarrow c_1$	local variable assignment
$\mid \mathbf{glb}_i c_1 \leftarrow c_2$	function closure initialization
$\mid c_1[c_2] \leftarrow c_3$	object store assignment
$\mid \nabla_n\, c_1\, c_2\, c_3$	function closure allocation
$\mid \lambda\, n\, c_1$	local variable allocation
$\mid c_0(c_1 \ldots c_n)$	function application
$\mid c_1\, ;\, c_2$	sequential evaluation
$\mid \mathbf{loop}\, c_1$	repeated evaluation
$\mid \mathbf{exit}\, c_1$	loop termination
$\mid \mathbf{trap}\, c_1\, \mathbf{with}\, l_i\, \mathbf{do}\, c_2$	exception handling
$\mid \mathbf{raise}\, c_1$	exception generation
$\mid \mathbf{builtin}(op)(c_1 \ldots c_n)$	builtin function application
$\mid \mathbf{alt}\, c_0\, \mathbf{of}$	
$b_1 \rightarrow i_1 \ldots b_n \rightarrow i_n\, \mathbf{do}$ $c_1 \ldots, c_n\, \mathbf{else}\, c_{n+1}$	multi-way case analysis

Figure 4: Abstract TML syntax

even more reduced, continuation-passing style (CPS) [App92] code representation that simplifies the static and dynamic program analysis and optimization tasks that are performed by the compiler, run-time query optimizer and thread scheduler (side-effect analysis, sharing analysis, inlining, dead code elimination, etc.) [GBM94]. However, CPS code needs to be normalized (closure converted, exception converted) prior to execution to achieve good executions on standard hardware architectures.

The semantics of TML programs involves syntactic entities that are denoted by indexed (i, j, k, n, m) variable names as follows:

TML instructions (see Fig. 4): $c \in Code$

Base values: $b, lit, g, p, l \in BVal = \mathcal{Z} \cup \{nil\}$

Local environments: $L = [l_0 \ldots l_k] \in Loc$

Dynamic environments:

$$E = [lit\ g_0 \ldots g_n\ p_0 \ldots p_m] \in Env$$

Evaluation results:

$$v \in Val = SVal \cup \{ok, exception(p), exit(p)\}$$

The semantics of a Tycoon object store is defined as a partial mapping from a domain of (tagged) object identifiers *Ref* (disjoint from the set of base values *BVal*) to fixed-sized arrays of state values. A state value is either an object identifier or a base value.

Object identifiers: $r, lit \in Ref$

State values: $sv, l, p, g \in SVal = BVal \cup Ref$

Object stores: $S \in Store = (Ref \times \mathcal{Z} \xrightarrow{fin} SVal)$
$\times (Ref \xrightarrow{fin} Code)$
$\times (Ref \xrightarrow{fin} \mathcal{Z})$

For example, the operation *init* returns the store value $(\{\}, \{\}, \{\})$ while the store operation *new* is de-

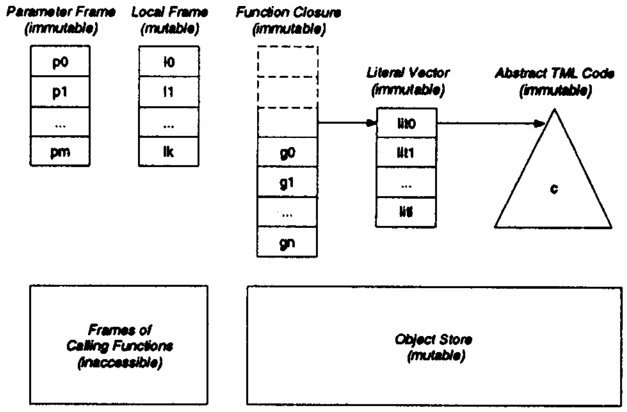
Parameter Frame (immutable)
Local Frame (mutable)
Function Closure (immutable)
Literal Vector (immutable)
Abstract TML Code (immutable)
p0
p1
...
pm
l0
l1
...
lk
g0
g1
...
gn
lit0
lit1
...
lit
c
Frames of Calling Functions (inaccessible)
Object Store (mutable)

Figure 5: The TML machine model

fined by

$$new((StV, StC, StS), n) = (StV', StC, StS')$$

where

$$\begin{aligned} r &\notin Dom(StV) \\ StV' &= StV + \{(r,0) \mapsto nil\} + \{(r,1) \mapsto nil\} + \\ &\quad \ldots + \{(r, n-1) \mapsto nil\} \\ StS' &= StS + \{r \mapsto n\} \end{aligned}$$

It takes a store (StV, StC, StS) and a size n and returns a new store that contains a new, *nil*-initialized store vector of size n that can be identified by its unique OID r. The remaining store operations *get*, *set*, *newclosure*, *fixexecute* are defined analogously.

The state of a TML thread executing a Tycoon function f consists of a quadruple E, S, L, c where E aggregates a reference to an immutable vector *lit* (that holds the string, longreal, ... literals of f), the immutable global bindings $g_0, \ldots, g_n$ of f, and the immutable actual parameter $p_0, \ldots, p_m$ of f. S describes the current state of the object store. L describes those local bindings in f that are inaccessible to other threads, and c describes the instruction of f that is currently being executed. Figure 5 depicts the relationship between these thread components.

The precise semantics of each TML instruction can now be described by its impact on TML thread states and by the operations executed on the persistent object store [Mat93] (structured operational semantics [Plo81]). This semantic definition is "constructive" in the sense that it provides a precise starting point for the implementation of TML interpreters.

The evaluation of a TML instruction is described using the following notation:

$$E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, v \rangle$$

The execution of the (composite) instruction c in a dynamic context E, an object store state S_1 with local state variables L_1 leads to an object store state S_2, local state variables L_2 and an evaluation result v. This definition implies that an instruction cannot modify its dynamic context E.

For example, the deduction rule *[Eval seq]* defines that the execution of the sequential composition $c_1 ; c_2$ in an environment E, against a store S_1, with local state variables L_1 is equivalent to the evaluation of c_1 in this thread state returning a (possibly modified) object store S_2, a (possibly modified) set of local state variables L_2, and a value v, followed by the execution of c_2 in this modified environment, again returning a (modified) object store S_3, (modified) state variables L_3, and a value v'.

[Eval seq]

$$\frac{\begin{array}{c} E, S_1, L_1 \vdash c_1 \Rightarrow \langle S_2, L_2, v\rangle \\ E, S_2, L_2 \vdash c_2 \Rightarrow \langle S_3, L_3, v'\rangle \end{array}}{E, S_1, L_1 \vdash c_1 ; c_2 \Rightarrow \langle S_3, L_3, v'\rangle}$$

Note that the evaluation result v of the first instruction is discarded and that the evaluation result of the instruction sequence is the result v' of the second instruction. This is typical for an imperative programming style, where statements do not compute a result but simply perform side-effects on the store.

The semantics of TML function applications is defined as follows: In a first step, an object store reference r is computed which identifies a function closure in the object store S_2 with code c', literals lit and global variable bindings $g_0 \ldots g_n$. In a next step, the actual parameters $p_0 \ldots p_m$ are evaluated (strict left-to-right evaluation order). The code c' is executed in a newly-allocated dynamic context consisting of $lit, g_0 \ldots g_n$ and $p_0 \ldots p_m$. After c' has been evaluated, the dynamic context of the calling function is restored as it has been left behind by the evaluation of the last argument.

[Eval apply]

$$\frac{\begin{array}{c} E, S_1, L_1 \vdash c \Rightarrow \langle S_2, L_2, r\rangle \\ \mathit{fixexecute}(S_2, r) = (c', lit, g_0, \ldots, g_n) \\ E, S_{i+2}, L_{i+2} \vdash c_i \Rightarrow \langle S_{i+3}, L_{i+3}, p_i\rangle \quad i = 0 \ldots m \\ [lit\ g_0 \ldots g_n\ p_0 \ldots p_m], S_{m+3}, L_{m+3} \vdash c' \Rightarrow \langle S, L_{m+4}, v\rangle \end{array}}{E, S_1, L_1 \vdash c(c_0 \ldots c_m) \Rightarrow \langle S, L_{m+3}, v\rangle}$$

A complete definition ot the Tycoon thread evaluation semantics using the above notation is given in [Mat93].

6 Concluding remarks

This paper gives an abstract view of the evolution of database models and languages in terms of bindings between code, data and threads. We argue that the next logical step in this evolution is an improved support for activity-oriented applications by introducing first-class persistent threads. We also report on our work in formalizing, implementing and using persistent threads in the polymorphic Tycoon database programming environment.

Our distinction between nine patterns of bindings in persistent systems makes it possible to classify database systems based on their support for persistent data, object, and activity management. Moreover, our presentation of the pitfalls encountered in the implementation of C→D and D→C bindings in existing systems is intended as a hint to implementors of D→T bindings not to under-estimate the intrinsic complexity of persistent threads and to realize the relative simplicity of the proposed Tycoon execution model and its implementation architecture.

Finally, it should be noted that persistent threads are an expressive and efficient, but rather low-level concept for the management of cooperative activities. Therefore, we are currently investigating related database synchronization and communication models that have been proposed in the literature [GMS87, Reu89, BDS+93] and how such models can be realized as polymorphic libraries encapsulating Tycoon's persistent threads. This approach to a flexible reconciliation of system and user needs has proved to be highly successful in modern programming and operating environments which offer several higher-level models like monitors, ACID transactions, transactional RPCs and communicating processes on a common, standardized thread abstraction available on multiple system platforms [POS90, OSF93].

Acknowledgements

This research is supported by ESPRIT Basic Research, Project FIDE, #6309 and by a grant from the German Israeli Foundation for Research and Development (*bulk data classification*, I-183 060).

References

[AGO91] A. Albano, G. Ghelli, and R. Orsini. A Relationship Mechanism for a Strongly Typed Object-Oriented Database Programming Language. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 565–575, 1991.

[AM85] M.P. Atkinson and R. Morrison. First class persistent procedures. *ACM Transactions on Programming Languages and Systems*, 7(4), October 1985.

[App92] A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis. *Building an Object-Oriented Database System: The Story of O₂.* Morgan Kaufmann Publishers, 1992.

[BDS+93] Y. Breibart, A. Deacon, H.-J. Schek, A. Sheth, and G. Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. *ACM SIGMOD Record*, 12(3):23–30, September 1993.

[BK91] N.S. Barghouti and G.E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, September 1991.

[BMS93] A. Borgida, J. Mylopoulos, and J. Schmidt. The TaxisDL Software Description Language. In M. Jarke, editor, *Database Application Engineering with DAIDA*, pages 65–84. Springer-Verlag, 1993.

[DCBM89] A. Dearle, R. Connor, F. Brown, and R. Morrison. Napier88 – A Database Programming Language? In *Proceedings of the Second International Workshop on Database Programming Languages, Portland, Oregon*, June 1989.

[GBM94] A. Gawecki, Mathiske. B., and F. Matthes. The Tycoon Machine Language TML: An Optimizable Persistent Program Representation. DBIS Tycoon Report 103-94, Fachbereich Informatik, Universität Hamburg, Germany, March 1994.

[GMS87] H. Garcia-Molina and K. Salem. Sagas. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data, San Francisco, California*, pages 249–259, May 1987.

[GR93] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques.* The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, 1993.

[Mat93] F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

[MMS92] F. Matthes, R. Müller, and J.W. Schmidt. Object Stores as Servers in Persistent Programming Environments – The P-Quest Experience. FIDE Technical Report Series FIDE/92/48, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, July 1992.

[MS91] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Database Programming Languages: Bulk Types and Persistent Data.* Morgan Kaufmann Publishers, September 1991.

[MS92] F. Matthes and J.W. Schmidt. Definition of the Tycoon Language TL – A Preliminary Report. Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

[MS93] F. Matthes and J.W. Schmidt. System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways. In P.P. Spies, editor, *Proceedings of Euro-Arch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

[Nel91] G. Nelson, editor. *Systems programming with Modula-3.* Series in innovative technology. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[OSF93] OSF. *OSF DCE Administration Guide – Core Components.* Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[Plo81] G.D. Plotkin. A structural appraoch to operational semantics. DIAMI FN 19, Computer Science Department, Aarhus University, 1981.

[POS90] Portable Operating System Interface for Computer Environments (POSIX). Federal information processing standards publication NBS-FIPS-PUB-151-1, National Bureau of Standards, 1990.

[RCS93] J.E. Richardson, M. J. Carey, and D.T Schuh. The design of the E Programming Language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[Reu89] A. Reuter. ConTracts: A Means for Extending Control Beyond Transaction Boundaries. In *Third International Workshop on High Performance Transaction Systems*, 1989.

[SM93] J.W. Schmidt and F. Matthes. Lean Languages and Models: Towards an Interoperable Kernel for Persistent Object Systems. In *Proceedings of the IEEE International Workshop on Research Issues in Data Engineering*, pages 2–16, April 1993.

[SM94] J.W. Schmidt and F. Matthes. The DBPL Project: Advances in Modular Database Programming. (to appear in Jounal 'Information Systems'), 1994.

[SMR+93] K. Subieta, F. Matthes, A. Rudloff, J.W. Schmidt, and I. Wetzel. Viewers: A Data-World Analogue of Procedure Calls. In *Proceedings of the Nineteenth International Conference on Very Large Databases, Dublin, Ireland*, August 1993.

[Str67] C. Strachey, editor. *Fundamental concepts in programming languages.* Oxford University Press, Oxford, 1967.

[Swe93] K.D. Swenson. Visual Support for Reengineering Work Processes. In *Proceedings of the Conference on Organizational Computing Systems, COOCS'93.* ACM Press, 1993.

[Tur93] K. Turner, editor. *Using Formal Description Techniques, An Introduction to Estelle, Lotos and SDL.* Wiley series in communication and distributed systems. John Wiley & Sons, 1993.