Chapter 2.1.4
Tycoon: A Scalable and Interoperable Persistent System Environment

Florian Matthes, Gerald Schröder, and Joachim W. Schmidt

Technical University Hamburg-Harburg
Harburger Schloßstraße 20
D-21071 Hamburg, Germany

Summary We describe the Tycoon persistent system architecture that achieves a high degree of scalability and interoperability through a full integration of persistent data, programs and threads while maintaining a strict separation of storage, manipulation, modelling and representation tasks into well-defined system layers. For several of these layers alternative implementations with distinct operational support are provided that can be configured dynamically to match best the requirements of a given application. Moreover, we present Tycoon's architectural contributions to enable interoperability with existing generic services like databases, user-interface toolkits and C++ program libraries.

1. Introduction and Motivation

Modern information systems provide their users with a flexible and problem-oriented access to large repositories of complex persistent objects. The efficient and cost-effective implementation of such information systems has to be based on *generic services* provided by commercially available systems like relational or object-oriented databases, graphical user-interface toolkits, program libraries or standardized communication services.

The goal of the Tycoon[1] project carried out by our group since 1992 in the context of the ESPRIT Basic Research Project FIDE is to improve substantially the programmer productivity and modelling flexibility in such an *open* heterogeneous system environment through contributions at two levels. First, Tycoon provides a persistent polymorphic programming language with an elaborate higher-order type system to uniformly describe existing generic services and to enable the consistent integration of these services via high-level application code. This language aspect of the Tycoon project is described in more detail in Chapter 1.1.1, [12, 17] (synopses in Chapters 1.4.2 and 3.3.3) and [18, 14].

In this paper we focus on the second contribution of the Tycoon project, the development of a *persistent system architecture* to support the Tycoon language and a seamless generic server integration. Several innovative components of the Tycoon system have been described already in [15, 11, 5] (synopses in Chapters 1.5.2, 2.2.4 and 2.3.5) and [8, 9]. Here we present and explain Tycoon's overall system architecture which excels through its scalability and interoperability.

Tycoon's system *scalability* makes it possible to cover in a single linguistic and architectural framework system implementations that range from a stand-alone main memory Tycoon implementation on a personal computer to a large, networked, multi-user, optimizing persistent Tycoon implementation. Contrary to

[1] Tycoon: Typed communicating objects in open environments.

monolithic information system architectures such as SAP/R3 or Oracle, Tycoon applications that require only a limited system functionality can work with an efficient and lean system version that nevertheless can be scaled easily to cope with increasing operational demands over the lifetime of the application.

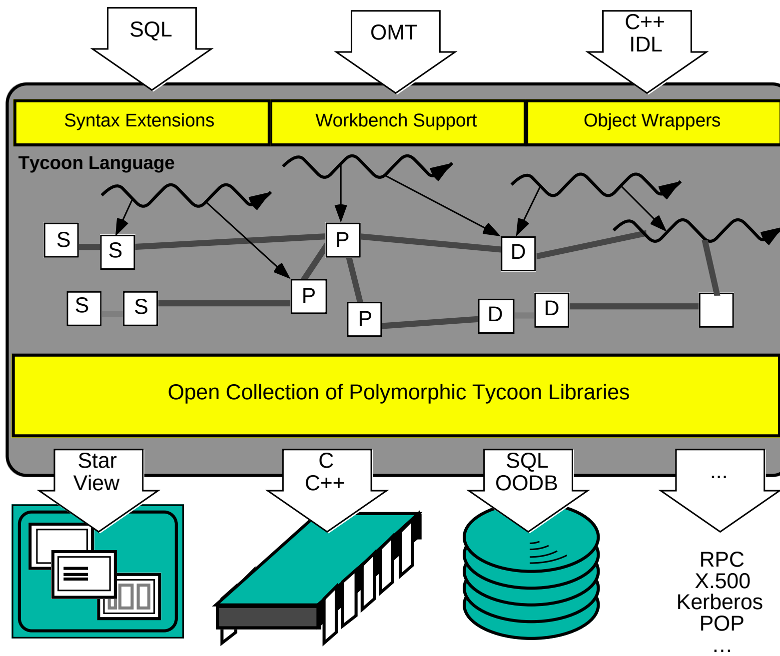

Fig. 1.1. Interoperability in the Tycoon environment

Figure 1.1 describes Tycoon's contributions to *interoperability*:

- Existing and evolving generic services are integrated in a *uniform manner* as polymorphic Tycoon libraries that provide type-safe access to external data and code. Using this approach, several generic services such as window systems (OpenWindows/NeWS), user-interface management systems (StarView), database systems (Ingres, Oracle), remote procedure call libraries (SunRPC), authentication services (Kerberos) and information retrieval engines (Inquery) have been integrated successfully in an add-on fashion into the Tycoon system. Within the Tycoon language, uniform naming, typing, binding and lifetime rules apply to screen, program and data objects (S, P, D in figure 1.1). In contrast to closed systems like fourth generation languages, Tycoon enables programmers to integrate also new generic services into their applications, that is, interoperability is not limited to a fixed set of object types like tables or forms. This extensibility is indicated by ellipses in the server icon list of figure 1.1.
- Integrated services are indistinguishable from services implemented in Tycoon itself. In particular, *type safety* and *persistence abstraction* can be obtained systematically also for external services in the context of the Tycoon system. As a

result, many programming errors can be found by static type checking at compile-time. This should be seen in contrast, for example, to untyped or string-oriented bindings as found in user-interface toolkits or SQL databases. Furthermore, Tycoon programmers are relieved from storage management issues like garbage collection, data transfer from and to disk, and recovery of persistent data.
- It is not only possible to integrate existing services into the Tycoon programming environment, but Tycoon is designed to be itself easily *integrated as a subsystem* into larger systems. For example, parts of the Tycoon compile-time and run-time environment can be reused as servers for persistent data storage, for portable code representation and evaluation, or for dynamic type checking. Similarly, Tycoon application code can be called from main programs written in C, C++, FORTRAN, COBOL etc. This interoperability aspect is indicated by the arrows at the top of figure 1.1.

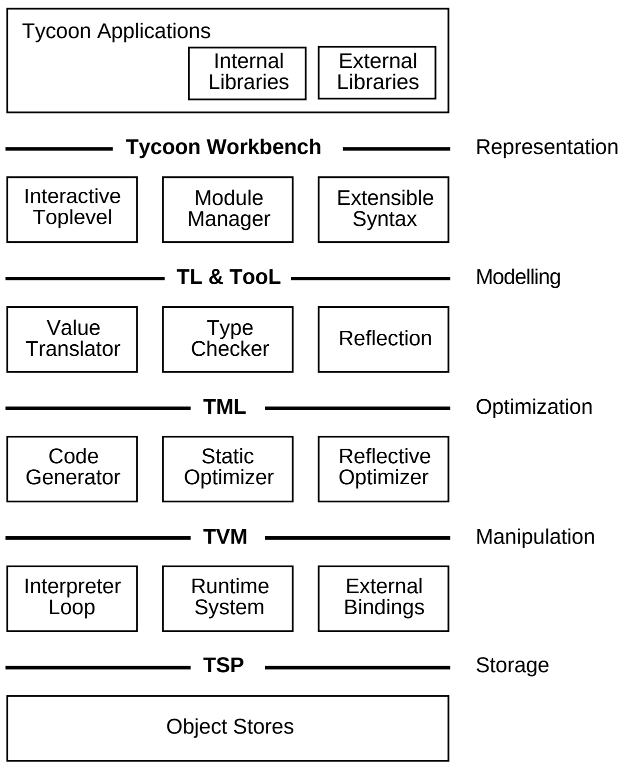

Fig. 1.2. Overview of the Tycoon persistent system layers

Figure 1.2 provides an overview of Tycoon's persistent system architecture that strictly separates storage, manipulation, modelling and representation tasks into well-defined system layers. For several of these layers, alternative implementations with distinct operational support are provided that can be configured dynamically to match best the requirements of a given application.

In this paper we discuss these layers of the Tycoon system environment in a bottom-up fashion: As described in section 2, full persistence abstraction for data and code is achieved already at the lowest layer of the Tycoon system defined by the Tycoon Store Protocol (TSP). The Tycoon Virtual Machine (TVM) presented in section 3 provides a platform-independent higher-order execution model that also supports calls, callbacks and exception handling involving external code written in other languages. In addition to executable TVM code, our persistent system architecture relies heavily on a persistent intermediate code representation (Tycoon Machine Language, TML) presented in section 4 which is used for static and reflective run-time code analysis and optimization. Since TML is an untyped language it is possible to implement multiple (polymorphically-typed) languages using a common TML intermediate representation. In Section 5 we highlight how modern language features like dynamic typing and reflection are supported by Tycoon's persistent system architecture. The wide range of services to be integrated into the Tycoon languages calls for a syntactic flexibility supported by the concept of extensible grammars as described in section 6. In section 7 we sketch how the Tycoon system components are integrated into a self-contained persistent programming environment including mechanisms for separate compilation and global optimization. The paper ends with a short summary of our experience using the Tycoon environment.

2. The Tycoon Store Protocol (TSP)

The Tycoon Store Protocol defines a *uniform* data-model-independent call-level interface to multiple (commercially distributed) persistent object stores. TSP is defined as an ANSI C header file which fully abstracts from implementation details of the underlying persistent stores. TSP is discussed in detail in [11] (see Chapter 2.2.4).

A TSP client can switch freely from one store implementation to another and work with more than one persistent store at once, but references between stores are not supported. Furthermore, TSP defines a portable linear data representation (TXR) to achieve store-level interoperability between all TSP server implementations. TSP concentrates on that subset of store functions that is provided by existing, widely used commercial and public-domain persistent object stores. TSP has a slight bias towards the needs of clients that have to implement persistent and possibly distributed programming languages.

A typical TSP server is implemented by writing a store adaptor that maps TSP data structures and functions to data structures and operations of an existing object store. As of today, this has been carried out for the commercial multi-user store ObjectStore (ObjectDesign) and the research prototype FLASK (Chapter 2.3.1). Moreover, our group has implemented two additional stand-alone TSP servers particularly well-suited for personal computers.

TSP contributes significantly to Tycoon's system scalability since TSP clients can choose between

- different garbage collection strategies;
- different object faulting mechanisms;
- optional error recovery, logging and persistent savepoint mechanisms;
- single-user or multi-user access;
- alternative commit protocols (for distributed systems);

– optional security and authentication support.

At present we are investigating the orthogonal combination of these features through a layering of (partial) TSP implementations.

As an experiment in store-level interoperability, an NFS (Network File System) server was written that exports selected TSP store objects as files and directories. In this way, all file-level commands are available on TSP stores, for example, *cd* and *ls* for browsing through an object graph.

A primary design goal behind TSP has been to provide efficient data storage as independently as possible from the data and language model supported by TSP store clients. In particular, TSP should be capable of supporting polymorphically typed models where the components of store objects contain values of multiple types that cannot be fixed in a separate schema definition phase. TSP therefore uses an *untyped* low-level store model, i.e., there is no separate dictionary of type or schema information maintained by the store. Instead of this, store values are made self-descriptive by imposing a regular object layout and a uniform tagging scheme. As a result, TSP achieves data model independence without sacrificing the advantages of self-descriptive statically-typed databases.

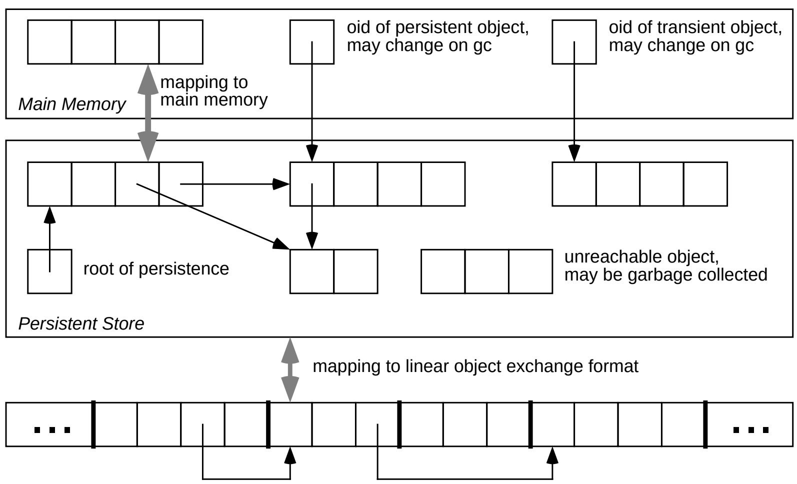

Fig. 2.1. Accessing a persistent store via the Tycoon Store Protocol (TSP)

The access to persistent store objects via the TSP is depicted in figure 2.1. Store objects are identified by unique persistent object identifiers (OIDs) that are also used to establish bindings between persistent objects. Each store has a single root of persistence (a location that holds the OID of a distinguished root object). Every object reachable from the root of persistence is stored persistently. Store objects can be created through TSP operations but are never explicitly destroyed by clients. The space of store objects that are not reachable from the persistent root object or from a (transient) OID held outside the persistent store during a session is reclaimed by a garbage collector. In general, all object access is performed using TSP function calls. TSP also supports the mapping of store objects into main memory to reduce the overhead incurred by repeated TSP calls.

Concurrency control and recovery on TSP stores is based on a flat transaction model. The TSP operations issued between two consecutive calls of the TSP *commit* operation are regarded as a single atomic transaction. Transactions of concurrently executing TSP clients are serialized. Furthermore, a TSP client can trigger an explicit TSP *rollback* operation to undo all side-effects on a TSP store since the last *commit* operation.

TSP provides customization facilities to support a tight integration of TSP servers into client systems. A TSP client can install callback functions that are called whenever certain state transitions occur in the backend.

- If a failure occurs, a failure handler is called with an error code. Depending on this code, the client can try to recover from the failure. This is especially useful in handling deadlocks or commit failures.
- If garbage collection is to be performed, a garbage collection handler is called. The client can decide to perform some additional operations (e.g. in an interactive user mode) before the internal garbage collection is triggered.
- The garbage collector of a TSP backend calls a client-provided enumerator function that has to enumerate all store object identifiers (OIDs) which are held outside the object store. Objects transitively reachable through these OIDs are not garbage collected. Since some persistent stores change OIDs during garbage collection, the client-provided enumerator function is also used to propagate OID changes to TSP client data structures.
- To transfer arbitrarily complex object store subgraphs between multiple TSP servers via sequential streams, a handler is needed that encapsulates the byte stream access. In this way, the TSP implementation remains independent of file systems, network protocols or other input/output facilities.
- TSP provides hooks to enable a portable transfer of client-specific data types like unique symbols, floating point numbers, matrices, etc. not supported by the predefined TSP data formats. This is accomplished by so-called *extern/intern* handlers that are called on data export and on data import via sequential byte streams.

3. The Tycoon Virtual Machine (TVM)

The Tycoon Store Protocol defines a passive data repository that is independent of a particular evaluation model. Executable code is stored as persistent data by the store manager. The store manager is thus independent of code representations that can be tailored to specific application purposes.

The Tycoon Virtual Machine (TVM) is an abstract call interface above the TSP layer that defines a bytecoded instruction set based on a higher-order, functional execution model. TVM bytecode is either interpreted by a virtual machine or is compiled on the fly into target machine code. The TVM interpreter and its associated run-time system are written in ANSI-C and have been ported to SunOS 4.1.x, Solaris 2.x, Linux, Windows-NT and MacOS.

The platform-independence of the TVM model makes it possible to dynamically transfer portable bytecode between heterogeneous nodes in a distributed programming environment without recompilation. Utilizing TSP's linear external data representation (TXR), it is also possible to migrate a persistent thread (code in the process of being executed) across system boundaries [8, 9, 15]. A thread is represented as an object graph describing bindings between code and data objects.

In addition to the interpreter vs. compiler alternative, scalability at the TVM level is achieved by switching between single-threaded and multi-threaded TVM implementations or between TVM implementations that support first-class persistent threads or transient evaluation contexts only. The set of TVM instructions can also be extended with new instructions to meet the needs of specific application domains like floating-point-intensive simulation software.

Individual TVM instructions are guaranteed execution atomically w.r.t. their effects on TSP stores. This property can be implemented rather efficiently at the TVM level and simplifies the implementation of higher system layers in a multi-threaded and multi-user persistent system scenario.

To enable interoperability with existing and future libraries written in other programming languages, TVM includes a portable and efficient mechanism to access functions stored in statically or dynamically bound object libraries. This mechanism is also exploited heavily in the Tycoon architecture to support an open set of "base" types like fixed point and floating point numbers of different precisions, date and time values, etc.

In the TVM bytecode there is no distinction between calls to static and dynamic libraries so the TVM code remains independent of the binding mechanism employed by a particular TVM implementation. The name of the external library and the symbol name (the name of the function to be called) are specified as run-time values along with a description of the function argument values and types. The interpreter maintains a list of statically bound libraries and symbols that is searched first. If no matching static library is found, a dynamic binding via the dynamic linker of the underlying operating system is attempted. On platforms that do not support dynamic linking, all required external servers (like window libraries or databases) have to be bound statically to the TVM. This requires only the insertion of the library name and the exported symbol names into a table in the source code of a distinguished TVM module. Then the TVM has to be rebuilt, compiling this module and relinking the executable. Again, these binding mechanisms are motivated by the desire to support a scalable system architecture.

Much care has been devoted in addition to support callbacks from external code to TVM byte code through a binding mechanism that exports an arbitrary TVM function object (closure) as an external function pointer. This is achieved by a generic wrapper function that takes a TVM argument type descriptor list and a function closure and dynamically generates a wrapper function that maps external function arguments to TVM function arguments.

The parameters that can be transferred between the TVM and external languages are limited to unstructured values, i.e. numbers, strings, words (including pointers and OIDs). Structured values have to be handled by wrapper functions written in C or in TVM code that construct and decompose these values.

To ensure a seamless interoperability at higher system layers, the external call and callback interface has to ensure that TVM exceptions are propagated correctly across language boundaries even in multi-threaded TVM applications.

A non-trivial problem arises if persistent TVM programs and threads work with pointers to data structures outside TSP stores. The lifetime of such data structures is limited to the lifetime of a single operating system process. Therefore, TVM offers mechanisms to log the construction and destructtion of these *volatile* values so that these values can be reconstructed automatically if a persistent thread is resumed. The log is essentially a list of TVM constructor and destructor functions stored persistently. As a result, it is possible, for example, to open and close windows in a windowing system, commit a state of the store, and to have the the windows that existed in the committed state automatically recreated on startup of the store. Note that this feature does not assume that the window system or other external libraries have been prepared specifically for usage in a persistent system architecture but that this persistence support is added independently by the TVM layer. This is discussed in more detail in [7].

4. The Tycoon Machine Language (TML)

While TVM code is designed for efficient executability on multiple platforms, our persistent system architecture relies heavily on an additional persistent *intermediate* code representation (Tycoon Machine Language, TML) to unify static compile-time and reflective run-time code analysis and optimization. TML is an untyped language that is used for the implementation of multiple (polymorphically-typed) languages using a common TML intermediate representation. A detailed description of the syntax and semantics of TML and of the TML optimization algorithms can be found in [5] (a synopsis appears in Chapter 2.3.5).

A TVM bytecode sequence is generated for a given TML term by a compiler backend written in TL, the Tycoon Language. TML is a Continuation Passing Style (CPS) program representation that is well-suited for standard code optimizations and for algebraic query optimizations.

TML is a powerful yet simple program representation technique. The advantage of TML lies in the dramatic reduction of the number of program constructs that have to be handled by an optimizer, namely variable access, function abstraction, function application and primitive function application only. TML is particularly well-suited for control and data flow analysis by making the flow of control explicit through the uniform use of one language construct: the function application. Since TML does not have implicit function returns this language construct can be viewed as a generalized *goto* with parameter passing.

TML has a simple and clean semantics based on the λ-calculus. It is effectively a call-by-value λ-calculus with store semantics. By representing programs in TML, many well-known optimization techniques become special cases of a few simple and general λ-calculus transformations. Due to certain syntactical restrictions on TML trees, these transformations can be applied freely even in the presence of side-effecting calls to primitive functions.

Primitive functions (such as: conditional operations, integer arithmetic, object store access) encapsulate most of the semantics of a specific application programming language and operate on an implicit, hidden store. This separation between generic CPS functionality and specific application language functionality is also reflected by the implementation of the TML optimizers where all information about primitive procedures (meta evaluation function, cost attributes, target code generation function, etc.) is factored-out from the generic TML analysis and rewrite algorithms.

The TML optimizers and the TML to TVM translator do not perform checks on TML trees to test their well-formedness, e.g. whether a function receives more or different parameters than expected. This is the responsibility of the clients of these tools, i.e. of a frontend for a specific application programming language.

5. The Tycoon High-Level Languages

As depicted in figure 1.2, the task of the Tycoon high-level languages (like TL and TooL) is to support flexible high-level modelling of data, programs and threads independent of execution and storage details.

Again, scalability is achieved by supporting multiple high-level languages within the Tycoon architecture. The implementation of a particular high-level language is encapsulated by a tailored compiler function that takes typed data structures (an abstract syntax tree of the program to be compiled and a compile-time environment) as its input and returns typed data structures (a TML tree and a possibly extended compile-time environment) as its result. Target code generation and code optimization are performed in a second phase based on the returned TML tree. The TML tree may contain bindings to arbitrary store values like string literals or function closures that are preserved by all rewriting and code generation steps of the backend. The possibility of passing typed persistent bindings from the source code through to the executable code is exploited, for example, in persistent hyperprogramming systems like the one described in [6].

Based on the backend functionality provided by TML, TVM and TSP, and by utilizing the generic frontend functionality provided by extensible grammars (see section 6), a special-purpose database language L can be implemented rather rapidly in the Tycoon persistent system architecture. A tailored compilation function for L can check the static semantics of L (scoping and typing rules) by a recursive traversal of the abstract syntax tree and can realize the dynamic semantics of L by a structure-directed mapping of language terms into TML terms. If necessary, language-specific TML primitives (e.g., *method-lookup*) and run-time support libraries (e.g., implementing operations on bulk data structures like relations) can be utilized to extend the higher-order TML core execution model by semantic primitives required by L.

Our group in Hamburg has implemented two high-level languages based on expressive *polymorphic* type systems. As discussed, for example, in [1] and [12, 17] (synopses in Chapters 1.4.2 and 3.3.3) such languages can serve as meta models to describe and integrate existing relational, functional and object-oriented database models.

- The Tycoon Language (TL, [13, 10]) excels by its very expressive type system based on existential and universal type quantification, recursive types and structural subtyping similar to the type system of the experimental polymorphic programming language Quest [2]. Contrary to Quest which is based on the notion of values, types and kinds (types of types), TL has a full *higher-order* type system where type quantification and subtyping applies uniformly to types, type operators and higher-order type operators, eliminating the need for a separate kind level.

  In addition to very few built-in types like *Int*, *String* or *Bool*, TL defines a small set of data type constructors: Tuples aggregate a fixed number of value and type bindings, arrays aggregate a flexible number of value bindings with homogeneous signatures, variant records define types that consist of a finite union of tuple types distinguished by a discriminator. The orthogonal combination of these type constructors along with recursive type definitions and (recursive) type operators covers virtually all data structures that appear in persistent programming.

  The TL execution model is mainly imperative and includes updatable locations. Since functions and types are first-class language objects, powerful generic higher-order functions can be written. To cope with run-time errors, exceptions and exception handlers are supported. Furthermore, TL includes mechanisms for reflective programming, dynamic typing and external language bindings as described in the remainder of this section.
- The Tycoon object-oriented language (TooL) is organized around (parameterized) classes and message passing. The type system of TooL captures much of the flavor of Smalltalk within a safe static typing discipline. Following the spirit of Smalltalk that provides a highly flexible and extensible programming envi-

ronment based on a small set of expressive language primitives, TooL provides only a few built-in type concepts with rich semantics that achieve power through systematic use and orthogonality. TooL integrates type concepts that are well understood in isolation like object types, subtyping, type matching and type quantification into a practical database programming language.

In the following subsections we point out some innovative language implementation techniques that are utilized in our Tycoon language processors and that are also relevant for other high-level persistent language implementations.

5.1 Dynamic Types

In long-lived and distributed applications there are situations where type checking cannot be performed completely at compile-time. For example, if values are transmitted (via files or communication channels) between independently developed applications, there is no common scope in which a static type check could be performed to guarantee compatibility between data and programs.

For such situations, the Tycoon languages provide dynamically-typed values. A dynamic value is a pair of value v and run-time type representation t that describes the type of v. A type representation is created automatically by the compiler whenever a dynamic value is created. If a value component v is extracted at run-time from a dynamic value, its associated type representation t can be inspected. Most languages that support dynamic types limit this inspection to a simple boolean subtype test: is t subtype of a given supertype T defined at compile-time? Tycoon provides a much richer set of strongly-typed functions for the algorithmic inspection of dynamic values, for example, to iterate over the attribute values and attribute types of a tuple or to construct a tuple from a list of typed bindings.

Another application area for dynamic types is the implementation of generic functions with type-dependent behavior. These functions take a type representation, usually along with a value of this type, analyze the type and exhibit different behavior depending on the type. An example is a generic persistent store browser that is capable of displaying and manipulating values of any type.

In the Tycoon scenario where the compile-time and run-time environment reside in the same persistent store, it is advantageous to use the same type representation at compile-time and at run-time. A dynamic type representation is a compile-time value (a run-time value of the compiler) that is stored persistently until the run-time of the application. More generally, the Tycoon compilers provide a binding mechanism for making arbitrary compile-time values accessible to the generated code. Furthermore, the functions to inspect, check and create type representations are shared (as persistent store objects) between the compile-time type checker and the run-time environment.

The higher-order Tycoon language TL provides a limited form of *dependent* types, i.e. TL type expressions may depend on value identifiers. For example, an abstract data type depends on a specific implementation represented as a tuple value with functional components. Dynamic type representations for such dependent types involve bindings to run-time values. Therefore, the concept of compile-time name equivalence has to be mapped to the concept of run-time value equivalence (e.g., based on OIDs) and type representations may need to be constructed dynamically, depending on the result of value computations.

5.2 Compile-Time and Run-Time Reflection

The Tycoon languages TL and TooL are implemented using a compiler bootstrapped in the Tycoon language TL. In many Tycoon applications the compiler is used as a *black box* to develop a self-contained stand-alone application program. Advanced Tycoon applications like object-oriented database systems, programmable simulators or compilers for high-level languages can also access the functionality offered by the Tycoon compiler in a structured way. This is called compile-time or run-time reflection depending on the point in time the compiler functionality is used [19] (see Chapter 1.2.1).

Compile-time reflection is achieved by executing user-definable code during the compilation process if so-called *reflective* expressions appear in the source code of a program being compiled. This makes it possible to write code that depends on information that is known statically at compile-time, maybe inferred by the compiler. For example, generic code to traverse recursively defined data structures that depend on statically inferred type information can be written using compile-time reflection.

Run-time reflection is achieved by making typed bindings to compiler subcomponents (parser, type checker, code generator, evaluator, module manager, ...) available to applications at run-time. In this case, code is evaluated or even generated depending on run-time (computed) values that are not known statically at compile-time. Note that this kind of reflection, although very flexible, may lead to unexpected run-time type checking errors during reflective compiler calls.

Full type safety even in the presence of run-time and compile-time reflection is guaranteed by the consistent use of dynamic types in the Tycoon compile-time and run-time environment.

5.3 Typing of External Bindings

The flexible but untyped TVM mechanisms described in section 3 to establish bindings from TVM code to external, dynamically-bound library functions is the basis for high-level data and code interoperability of the Tycoon languages with an open set of external servers.

External functions are inherently type unsafe since neither object files nor linked libraries contain type information. This type information is available in the sources of the generating languages but is lost after compilation. This means that Tycoon cannot automatically check the parameters of function calls to external libraries. However, the correctness of calls to these functions can be enforced by the Tycoon programmer who binds the external libraries into the Tycoon system by assigning appropriate Tycoon signatures to these external functions.

The following example shows the binding of a function *int_add* that takes two integers and returns an integer stored in an external library *math* to the Tycoon identifier *intAdd* with the function type **Fun***(:Int :Int):Int*.

let *intAdd = bind(:***Fun***(:Int :Int):Int "math" "int_add" "iii")*
let *sum = intAdd(12 23)*

The string value *"iii"* specifies the low-level parameter-passing conventions to be used at the TVM level (here: call-by-value with two integers and return by value of an integer). All subsequent applications of *intAdd*, like *intAdd(12 23)*, can be checked for type correctness assuming that the binding has been typed correctly. Note that an external function like *intAdd* has true first-class status in Tycoon, for example, it can be passed as a parameter or can be stored persistently without restrictions.

As a more elaborate example of strongly-typed access to external functions, the following excerpt of the TL interface *SQL* exports polymorphic functions to perform dynamic SQL database access from Tycoon applications:

interface *SQL* **import** ... **export**
error :**Exception with** *sqlError:String* **end**
Table(E <:**Tuple end**) <:**Ok**
...
openTable(**Dyn** *E* <:**Ok** *tableName :String) :Table(E)*
...
insertTuple(E <:**Ok** *table :Table(E) tup:E)* :**Ok**
...
end

In the Tycoon libraries there exist two distinct modules *ingresSQL* and *oracleSQL* that implement this interface with bindings to the dynamic SQL call interfaces of these relational database systems. The unary type operator *Table(E)* exported from the interface describes the type of SQL tables with element type *E*. For example, a value of type *oracleSQL.Table(Person)* is an Oracle table with rows that have attributes as defined by the Tycoon tuple type *Person*.

The polymorphic function *openTable* opens an existing named table for further processing and takes a dynamic type variable *E* as its first argument to ensure that the database table structure matches the Tycoon type information. If a schema mismatch is detected, the Tycoon exception *error* is raised at run-time. All other operations of the SQL interface (queries, table updates) can be checked statically by the Tycoon compiler based on the polymorphic signatures assigned to the SQL functions. For example, the signature of the function *insertTuple* expresses the type constraint that into a table of type *Table(E)* only tuples of a matching type *E* can be inserted.

6. Extensible Grammars

The high-level Tycoon languages described in the previous section aim at minimality and orthogonality of concepts. The definition and implementation of these languages is based on a restricted *abstract syntax* represented by typed data structures in a Tycoon store. The mapping of a concrete syntax (a linear source text or a two-dimensional graphical notation) to an abstract syntax is not part of the language definition proper.

Most programming languages provide a fixed mapping from concrete syntax to abstract syntax. For convenience, several syntactic variants are often provided for the same construct of the abstract syntax. For example, the TL abstract syntax provides only one iteration construct (**loop** ... **exit** ... **end**). Other common loop constructs (**while**, **repeat**, **for**) can be expressed using **loop** in combination with **if**.

For languages like TL and TooL that have to support a wide spectrum of data and execution models, it is desirable to keep the concrete syntax adaptable to the needs of a particular application domain (see also Chapter 3.2.3). For example, declarative bulk data access can be supported by a tailored query language syntax (relational calculus, SQL, comprehensions notation) and parallel programming can be supported by tailored control structures (**par**, **alt**, ...).

The Tycoon technology of *extensible grammars* [3, 4] makes it possible to provide syntactic support for such add-on data and control abstractions.

Extensible grammars are implemented as a library of compiler tools written in TL. They were used to define the core syntax of TL and TooL thereby supporting the syntactic extension of both languages in a uniform way. Extensible grammars are parameterized by the abstract syntax tree constructors of the respective high-level Tycoon language.

The extensible grammar library also defines an abstract syntax for the dynamic definition and extension of grammars. Grammars describe the concrete syntax (EBNF notation) and the translation into abstract syntax trees using typed term constructors. Checks at grammar-definition time ensure that only well-formed abstract syntax trees are generated and that no syntactic ambiguities arise during grammar extension.

In a first step, the TL extensible grammar library is applied reflectively to define an initial concrete syntax for the definition of grammars using term constructors of the grammar library itself. In a second step, this grammar is used for the definition of a core programming language (e.g., TL or TooL abstract syntax). Although this initial syntax would be sufficient to express any well-typed program, it is not user-friendly. Thus, the initial version of the programming language syntax is used in further redefinitions, for example, to provide **while**, **repeat** and **for** loops and higher-level abstractions like **select from where** queries implemented on top of a bulk type library.

7. Integrated Persistent Programming Environment

In this section we sketch how the building blocks of the Tycoon persistent system architecture described in the previous sections are fully integrated into a persistent programming workbench to support the development of large data-intensive applications by teams of cooperating programmers. Since computer-aided software engineering itself is a data-intensive task, it is natural to implement a Tycoon workbench in one of the Tycoon languages exploiting orthogonal persistence, reflection, and dynamic typing (see also figure 1.2).

The interactive top level of the current TL workbench gives access to all tools in the environment via a command-line interpreter. A graphical workbench based on the concept of direct manipulation of persistent data and code is under construction. Separate compilation and incremental linking is currently built into the workbench, but we are implementing a generic separate compilation manager that exploits dynamic types, first-class environments, generic dependency checking and version management.

The development of large applications requires separate compilation. This facilitates teamwork by dividing the problem into manageable parts and eases the re-use of software components. Furthermore, type correctness of software components can be verified based on explicit assumptions about the interfaces of other software components.

TL supports separate compilation through three language constructs, namely, interface, module and library. A library defines a scope for nested module, interface and library declarations. An interface is syntactic sugar for a type- and value-parameterized tuple type definition. A module is syntactic sugar for a type- and value-parameterized link function that computes a tuple value as defined by the type of the module interface. The TL module manager tracks the dependencies between modules, interfaces and libraries and controls automatic recompilation and incremental linking based on these dependencies.

The TL programming environment has to inter-operate with commercial tools for project management and source-code version management. For example, our group is using SunSoft's TeamWare to coordinate the parallel Tycoon development by five full-time employees and approximately 35 students. Therefore, the TL workbench has to maintain the source code of library definitions, modules and interfaces as well as compiled modules and interfaces in the file system. The object store holds linked modules and persistent data only. This makes it necessary to assign version keys, check-sums and time stamps to source code and to compiled modules and interfaces to inform the module manager of changes performed by external tools on individual source or object files.

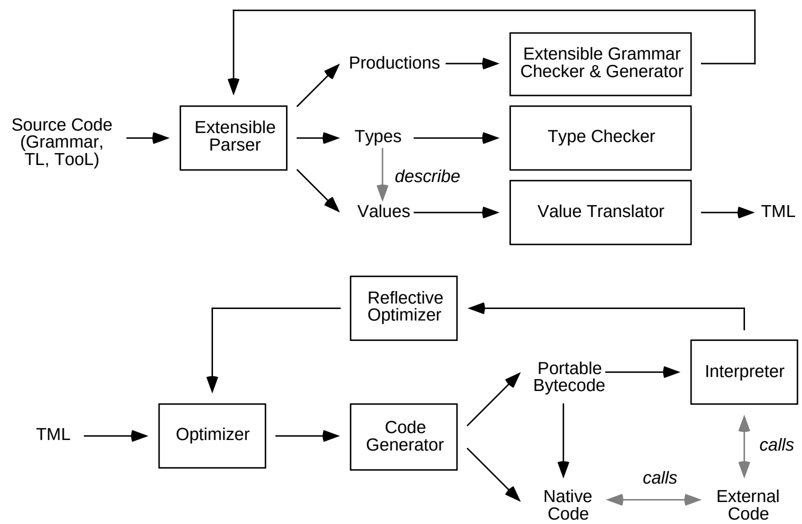

Fig. 7.1. Data flow between the language processors of the Tycoon workbench

The data flow between the components of the various Tycoon system layers described in this paper is shown in figure 7.1. Source files or interactive user input is parsed by the extensible parser (based on the TL or TooL syntax definition). Input phrases that contain grammar definitions are checked for consistency by the grammar checker and then transformed into parse rules for the extensible parser. Type expressions represented as abstract syntax trees are checked by the type checker for well-formedness according to static semantic rules of the respective Tycoon high-level language. Values are translated into (untyped) TML terms. TML terms may be optimized locally before they are submitted to the target code generator for the TVM. The TVM handles transparently the switching between the interpretation of portable bytecode and the direct execution of native machine code (external library code or compiled TVM code).

An interesting detail in figure 7.1 is the run-time reflective code optimizer. On request, the Tycoon target code generator augments the executable TVM code of each function by a (persistent) binding to its abstract intermediate code representation (a TML tree). This makes it possible to inspect function implementations at run-time. This is exploited in Tycoon to perform link-time and run-time optimizations across abstraction barriers.

To explain this form of optimization, consider the compilation of the expression *stack.empty(s)* in a module *m*. At compile-time, the only information available about the identifier *stack.empty* is the signature of the function. As soon as module *m* is linked, a binding to the module *stack* and its function *empty* is established. Tycoon's reflective optimizer exploits the binding information available after linking or more generally during program execution to compose the TML trees of separate functions into aggregated TML trees (dynamic inlining) that can then again be submitted to a (global) optimizer and code generator. The expansion and optimization of TVM functions in the persistent store are controlled by a tagging scheme and a cost model that avoid repeated optimizations of shared subfunctions and limit the increase in code size resulting from inlining.

8. Concluding Remarks

We have described the layers of the Tycoon persistent system architecture that emphasizes system scalability and interoperability with external servers. Currently, the following set of server gateways is available as TL source code:

- NeWS window system based on display PostScript: 48 modules; 6300 lines of code;
- StarView cross-platform user-interface toolkit: 131 modules; 6700 lines of code generated automatically from C++ header files;
- Oracle RDBMS: 3 modules; 1300 lines of code;
- Ingres RDBMS: 3 modules; 1300 lines of code;
- Kerberos authorization and authentication server [16]: 6 modules; 1055 lines of code;
- Inquery text retrieval engine: 3 modules; 1000 lines of code;
- RPC and socket communication: 13 modules; 1700 line of code.

An additional set of approximately 80 Tycoon modules with 29000 lines of TL code exports reusable library code like standard data types (date, time, string), bulk data types with uniform iteration abstractions, strongly-typed graphical data browsers and generic compiler toolkits (scanner, parser).

Through the bootstrap of the Tycoon language processors (TML, TL and TooL) and the implementation of an integrated object-oriented data modelling workbench (STYLE, see Chapter 3.2.2) we already have some experience with the development of complex Tycoon applications by cooperating teams of programmers.

system component	number of modules	total lines of code
TML	13	7600
TL	36	18200
TooL	22	9200
STYLE	250	41000

Since 1994, a WWW-driven information system implemented in TL has been operational at Hamburg University. This server was developed in the context of the ESPRIT Network of Excellence IDOMENEUS and utilizes Tycoon as a persistent store for data that is supplied by WWW forms and displayed on dynamically-generated WWW pages. Persistent TL threads are used to read, check and transform the data and to emulate persistent sessions between WWW clients and the WWW server. In particular, it is not necessary to install the Tycoon system as a demon, but Tycoon is started dynamically on request since persistent threads lead to very fast system startup sequences.

Acknowledgement This research was supported by ESPRIT Basic Research, Project $FIDE_2$, #6309.

References

1. M.P. Atkinson and P. Bunemann. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2), June 1987.
2. L. Cardelli. Typeful programming. Technical Report 45, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, May 1989.
3. L. Cardelli, F. Matthes, and M. Abadi. Extensible grammars for language specialization. In C. Beeri, A. Ohori, and D.E. Shasha, editors, *Proceedings of the Fourth International Workshop on Database Programming Languages, Manhatten, New York*, Workshops in Computing, pages 11–31. Springer-Verlag, February 1994.
4. L. Cardelli, F. Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Equipment Corporation, Systems Research Center, Palo Alto, California, February 1994.
5. A. Gawecki and F. Matthes. Exploiting persistent intermediate code representations in open database environments. In *Proceedings of the Fifth Conference on Extending Database Technology, EDBT'96*, volume 1057 of *Lecture Notes in Computer Science*, Avignon, France, March 1996. Springer-Verlag.
6. G.N.C. Kirby, R.C.H. Connor, Q.I. Cutts, R. Morrison, A. Dearle, and A.M. Farkas. Persistent hyper-programs. FIDE Technical Report Series FIDE/92/53, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, 1992.
7. M. Kornacker. Persistente Sicherungspunkte für langlebige Aktivitäten in offenen Umgebungen. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, August 1995. (In German.).
8. B. Mathiske, F. Matthes, and J.W. Schmidt. On migrating threads. In *Proceedings of the Second International Workshop on Next Generation Information Technologies and Systems, Naharia, Israel*, June 1995. (Also appeared as TR FIDE/95/136).
9. B. Mathiske, F. Matthes, and J.W. Schmidt. Scaling database languages to higher-order distributed programming. In *Proceedings of the Fifth International Workshop on Database Programming Languages, Gubbio, Italy*. Springer-Verlag, September 1995. (Also appeared as TR FIDE/95/137).
10. F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung*. Springer-Verlag, 1993. (In German).
11. F. Matthes, R. Müller, and J.W. Schmidt. Towards a unified model of untyped object stores: Experience with the Tycoon store protocol. In *Advances in Databases and Information Systems (ADBIS'96), Proceedings of the Third International Workshop of the Moscow ACM SIGMOD Chapter*, 1996.
12. 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.
13. 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.
14. 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.
15. F. Matthes and J.W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, September 1994.

16. A. Rudloff, F. Matthes, and J.W. Schmidt. Security as an add-on quality in persistent object systems. In *Second International East/West Database Workshop, Klagenfurt, Austria*, Workshops in Computing, pages 90–108. Springer-Verlag, 1995. (Also appeared as TR FIDE/95/138).
17. 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.
18. J.W. Schmidt, F. Matthes, and P. Valduriez. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*, pages 270–287. Springer-Verlag, October 1993.
19. D. Stemple, R. Morrison, and Atkinson M. Type-safe linguistic reflection. In *Database Programming Languages: Bulk Types and Persistent Data*, pages 357–362. Morgan Kaufmann Publishers, 1991.