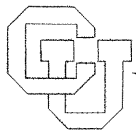
Incorporating Active and Multi-database-state Services into An OSA-Compliant Interoperability Toolkit *

O. Boucelma
J. Dalrymple
M. Dougherty
J-C Franchitti
R. Hull
R. King
G. Zhou

CU-CS-769-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research was supported in part by NSF grants IRI-9107055 and IRI-931832, and ARPA grants BAA-92-1092, F30294C0253, and 33825-RT-AAS.

Incorporating Active and Multi-database-state Services into an OSA-Compliant Interoperability Toolkit *

O. Boucelma†, J. Dalrymple, M. Doherty, J-C. Franchitti, R. Hull‡, R. King, G. Zhou§

Computer Science Department
University of Colorado
Boulder, CO 80309-0430

Abstract: This paper describes how Amalgame, a CORBA-compliant toolkit that supports software and database integration, is extended to include two complementary services. These are (1) "Activeness": In broad terms this is the ability to specify rulebases and execution models for them, possibly using a local persistent store; and (2) Multi-state Management: This provides the ability to easily manipulate and access multiple, simultaneous states of a database (or part of a database) and the deltas between them. A database programming language called Heraclitus[OO] (abbreviated H2O) is currently being implemented to support these services. Two example applications of these services are described: mediators for data integration, and enhancements to Amalgame to support sophisticated reasoning when combining and re-using interoperating software components.

1 Introduction

Supporting interoperation of software and database components is one of the most pressing Computer Science problems today. Current technologies focus on the development of standard architectures (such as OSAs [Bla94] using CORBA [OMG90]) which include an extensible set of services (e.g., persistence in the Texas Instruments Open OODB [WBT92]) that support the interoperation of entities manipulated by service specific languages. As an example, the ODMG'93 [Cat93] industry standard Object Definition Language (ODL), an extension of the OMG Interface Definition Language (IDL), allows the definition of location transparent persistent entities. Manipulation of these entities is made possible via programming language bindings to the ODMG'93 database language. Standard architectures are not currently addressing higher semantic levels of interoperation, such as managing replication, restructuring, and merging of data; and intelligent configuration management such as the selection of reusable software components. In this paper we show how OSAs can be populated to support these higher levels of interoperation.

*This research was supported in part by NSF grants IRI-9107055 and IRI-931832, and ARPA grants BAA-92-1092, F30294C0253, and 33825-RT-AAS.

†On leave from Université de Provence, France

‡On leave from the University of Southern California

§A student at the University of Southern California

As a starting point, we use the CORBA-compliant Amalgame toolkit [FK93b, FK93a], which provides facilities to support the integration of new or pre-existing software applications and components. To extend the scope of these facilities, we are now developing an object-oriented DBPL called Heraclitus[OO], abbreviated H2O, in order to support two fundamental H2O-based services: (1) "Activeness": In broad terms this is the ability to specify rulebases and execution models for them, possibly using a local persistent store. (2) Multi-state Management: This provides the ability to easily manipulate and access multiple, simultaneous states of a database (or part of a database) and the deltas between them. We are using Amalgame to build a variety of integration tools relying on these H2O-based services, to support functionalities such as data integration, incremental update of replicated data, and the intelligent integration of diverse software components. The multi-state management service is being used to develop version management tools in the context of software and groupware processes; it also forms an integral part of the activeness service.

In Section 2 we describe how the H2O services are incorporated into the Amalgame toolkit. Section 3 describes the multi-state management service, and indicates how it can be used to support highly flexible versioning in the context of configuration management. Section 4 describes how to use activeness in support of semantic aspects of database interoperation. Section 5 discusses how the Amalgame toolkit itself is being enhanced by using both activeness and multi-state management.

This work was influenced by a wide variety of research projects, including database programming languages (DBPLs), interoperability tools, active database systems, and software environments. In connection with DBPLs, the Heraclitus project [GHJ+93, GHJ94] has developed a stand-alone relational DBPL that incorporates deltas as first-class citizens. The H2O DBPL generalizes this by (i) providing an object-oriented DBPL that supports the ODMG'93 standard Object Query Language and is built on a persistent object-oriented platform; and (ii) providing much richer multi-state management capabilities, that includes the notion of "alternatives". Furthermore, (iii) H2O-based services are integrated within an OSA-compliant framework.

Most active database systems [MD89, CW91, SJGP90, GJ91] support one or a handful of pre-defined execution models for rule application. Because Heraclitus and H2O support deltas as "first-class citizens", they permit the specification of a wide variety of execution models. In conjunction with Amalgame, H2O provides the basis for constructing *active modules*. These are application objects (in the terminology of CORBA, see Section 2) that incorporate a rulebase, an execution model, and possibly a local persistent store. Thus, unlike active database systems, our framework can support active capabilities independently of any database system. Active modules also appear to be richer than "active objects" in the sense of [Buc93], because the execution models in active modules can access multiple virtual states, and because they can act in the context of distributed, heterogeneous environments.

Activeness capabilities have also been incorporated directly into software engineering environments, such as Marvel [HKBBS92] and GOODSTEP [CHCA94]. As with most active database systems, these environments support fixed execution models. We feel that the H2O-based multi-state management service

will be useful in these environments, both to help in managing versions of configurations, and for the purpose of exploring alternatives during the process of selecting tools for creating software artifacts.

2 Incorporating H2O Features into an OSA-Compliant Framework

In this section, we describe how we use H2O features in combination with current interoperability standards.

2.1 H2O with Respect to Existing Interoperability Standards

The goal of the H2O project is to provide multi-state support and activeness capabilities that can be easily integrated into existing programming languages. Multi-state support and activeness, along with distribution, persistence, etc., are part of a collection of (orthogonal) paradigms which play an important role in modern (database) software systems. However, building such systems on top of these various paradigms constitutes a serious interoperability challenge.

To address this challenge, current industry and research trends are converging on a bottom-up standardization of interoperability. In particular, Object Service Architectures (OSAs) are a major emerging trend in the systems software industry. An OSA [Bla94] consists of of a collection of independent software services which interoperate via a software backplane or message passing bus. To adhere to this standardization effort, we are implementing the H2O capabilities on top of the Amalgame framework [FK93a, FKB94]. Amalgame, which is being implemented at the University of Colorado, Boulder, is an application integration framework built on top of an OMG-compliant OSA. OMG's OSA is also referred to as the "Common Object Request Broker Architecture" (CORBA).

OMG has identified three broad categories of components which interoperate via an Object Request Broker (ORB) through object interfaces specified using the CORBA IDL. The components identified by OMG are: *Object Services*, *Common Facilities*, and *Application Objects*. Object Services provide basic functions for realizing and maintaining objects (e.g., lifecycle, naming, persistence, etc.). Common facilities provide general purpose capabilities, such as browsing and versioning, which are useful in many applications. Finally, application objects are specific to particular end-user applications. OMG recently adopted the ODMG'93 database programming language industry standard as a basis for its persistence service.

To allow the definition of distributed objects which are location-transparent, the Amalgame framework implements a parser for the OMG's IDL with various bindings to underlying RPC mechanisms (e.g., Q, ONC RPC, Sun RPC, etc.). The Amalgame framework also provides IDL bindings to several programming languages including ANSI C, C++, Modula-3, Common Lisp, and Ada. The Amalgame IDL parser is currently being extended to accommodate the IDL-extended syntax of the ODMG'93 database language. These extensions will allow Amalgame to implement applications running on top of several commercial ODMG semi compliant OODBMSs such as O2 [Deu90]. Based on these extensions, Amalgame will also

provide a C++ binding to ODMG'93 for the TI Open OODB, an extensible component-based database system.

By incorporating H2O-based services in the Amalgame framework, we can leverage off of the existing distribution and persistence capabilities. H2O-based services initially include "multi-state" support, and "activeness". The "activeness" service subsumes the "event" service specified by the OMG. We feel that one of the contributions of the H2O project is to investigate candidate services for extending the sparsely populated list of services specified by the OMG. Figure 1 illustrates the Amalgame OSA-compliant architecture extended with H2O-based services and common facilities. This figure also gives examples of specific application objects based on H2O and Amalgame capabilities.

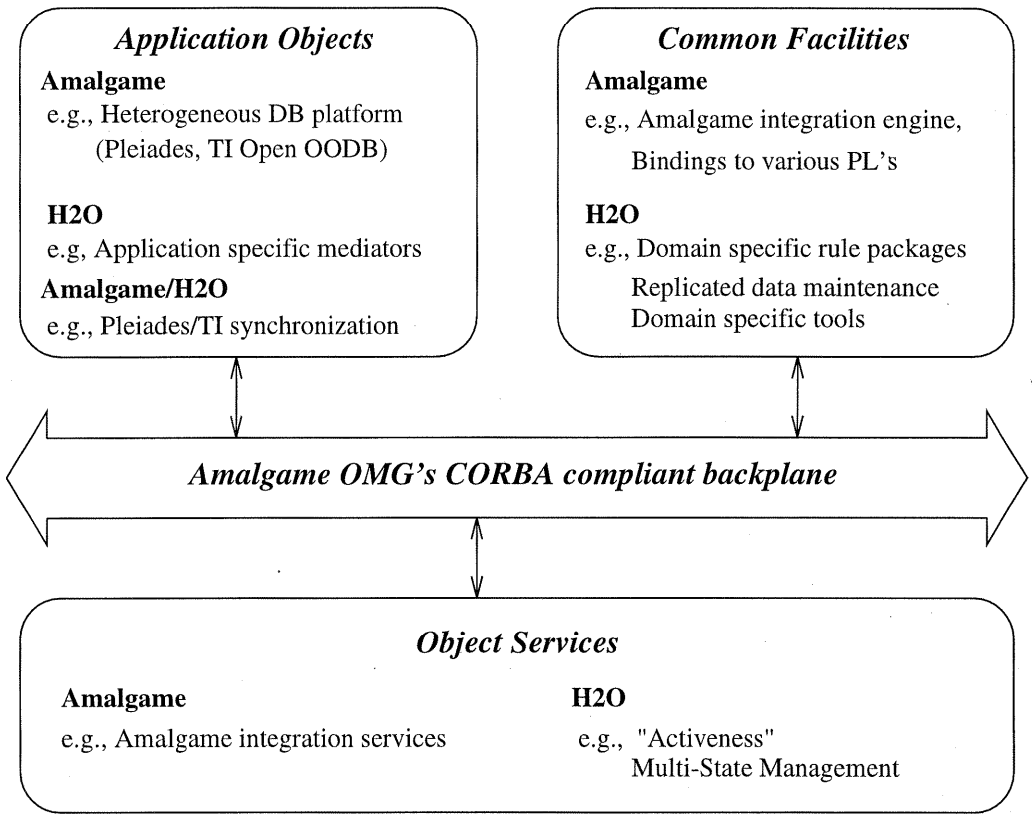


Figure 1: The Amalgame/H2O OSA-compliant framework

2.2 Populating the Amalgame Framework with H2O-based services

H2O's multi-state capabilities are provided via extensions to the ODMG'93 C++ binding. These extensions, embodied in the H2O DBPL enable the definition and manipulation of database subschemas which are subsets of classes in a database schema (see Section 3). Multiple, possibly hypothetical, "alternative" states of these subschemas and deltas between them can be created and manipulated at run-time. The multi-state language extensions provide operators for specifying the nature of these alternatives and characterize the differences between them in terms of deltas.

The H2O activeness service will provide extensions to the ODMG'93 C++ binding to configure and program various rule execution models. These language extensions have not yet been finalized; however, we have been experimenting with "activeness" by providing common facilities which can be used to build "active modules" that drive specific applications (see Section 4). Since H2O-based services are built as components of the Amalgame OSA, and since Amalgame provides support for distribution and persistence, active modules can therefore involve distributed and persistent components which all communicate via the Amalgame ORB.

Although, we are currently focusing on multi-state and activeness services, we will extend the set of H2O-based services and/or common facilities in the future. For example, we are currently providing an H2O-based multi-state manager as a common facility in the Amalgame framework. This facility includes domain specific rule packages, capabilities for selecting and defining rule execution models and hypothetical alternatives managers, and other domain specific tools (such as query rewriting tools). Using the H2O-based common facilities and services, it becomes possible to develop application specific rulebases and various types of "active" and "multi-state" modules. The resulting H2O applications are classified as new Application Objects in our extended Amalgame framework.

3 H2O Multi-State Management: Deltas and Alternatives

In this section we describe the multi-state management object service built using the H2O DBPL and give a brief description of its implementation. The H2O DPBL supports alternatives, which are essentially different versions of a database state, and deltas, which correspond to the differences between database states. As discussed below, alternatives and deltas may be defined relative to the full database schema or to subschemas.

One important application of alternatives is to provide flexible version management, with easy and natural mechanisms for context-switching between versions. To illustrate this, we will describe how alternatives can be used to represent and access in a transparent manner different configurations of a software process. Another important application of alternatives is to provide hypothetical database access.

In addition to alternatives, H2O supports *deltas*, which correspond to the differences between alternatives. Deltas were originally introduced in the relational DBPL Heraclitus [HJ91, GHJ94]. Deltas can be thought of as proposed updates that have not been applied to the database. Deltas are useful if the difference between two alternatives must be analyzed, e.g., to determine how different the two alternatives are, or to characterize some specific properties of that difference. Indeed, in most active database models rule conditions can access the delta between the initial state of a transaction and the "current" state that incorporates a user-requested update and the results of rules fired up to some point. Another important application of deltas, which will be illustrated below, concerns merging two or more proposed updates. Finally, in the current prototype of the H2O DBPL, deltas are used internally to support alternatives.

The H2O language constructs are general enough to be applicable to any C++-like host language. We

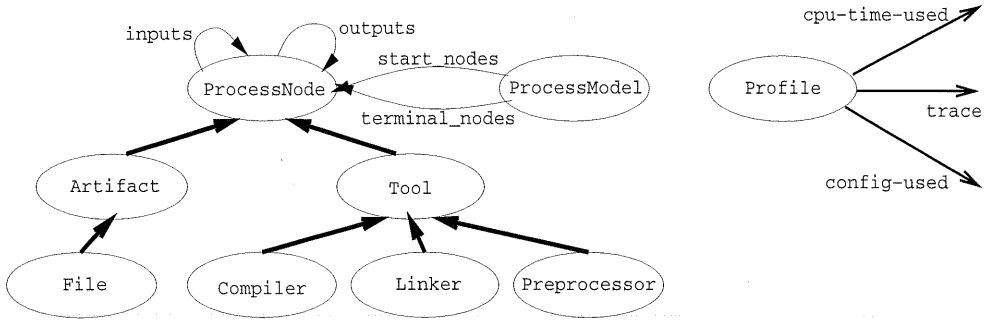


Figure 2: Class hierarchy for example schema

have chosen to use the ODMG'93 C++ binding as the host language for the prototype implementation. The ODMG standard defines the following sublanguages:

- ODL – Object Definition Language
- OML – Object Manipulation Language
- OQL – Object Query Language

We use the notation H2O[*xxx*] to refer to a particular host language *xxx* extended by H2O constructs. For example H2O[ODL] refers to the H2O extension of the object definition language.

We introduce the H2O-based multi-state management service and its implementation in four stages. First, we describe the notion of *subschemas* relative to which alternatives and deltas are defined. Next, we introduce and illustrate the notion of alternatives and show how they correspond to differing states of a particular subschema. We then introduce deltas, which capture the differences between alternatives or database states. We close with a brief description of how these constructs are implemented in the prototype currently under development.

As an example we use a simple process model for testing software artifacts. The model represents tools such as compilers and linkers, configurations of these tools, and profiles obtained by testing artifacts that are processed using these configurations. The classes used in this example are shown schematically in Figure 2. To keep the number of classes in this example small we have followed the style of object-oriented programming, and used inheritance to incorporate graph properties into artifact and tool objects. In a more database-oriented approach, software objects would not be subclasses of `ProcessNode`, and each `ProcessNode` object would use an attribute to indicate which artifact or tool is associated with it.

3.1 Schemas and Subschemas

In many applications it is useful to build alternative states and deltas for some restricted part of the database schema (class hierarchy), rather than of the whole schema. In H2O, both alternatives and deltas may range over the full schema, or over parts of it. We define a *subschema* to be a subset of the classes which make up the schema, with the restriction that all subclasses of any class in a subschema are also

included in that subschema. To illustrate, in our example it is useful to model different configurations by using alternatives that range exclusively over a subschema which includes the `ProcessNode` class and its subclasses, and also the `ProcessModel` class (see Figure 2). In this subsection we describe how H2O schemas and subschemas are specified.

An H2O schema is defined by writing the interfaces to the classes and specifying subschemas in H2O[ODL]. A partial declaration of our example schema follows.

```
interface ProcessNode {
        attribute Set<ProcessNode> inputs;
        attribute Set<ProcessNode> outputs;
};

interface ProcessModel {
        attribute Set<ProcessNode> start_nodes;
        attribute Set<ProcessNode> terminal_nodes;
};

interface Profile {
        attribute integer cpu-time-used;
        attribute List<integer> trace;
        attribute alternative<ProcessSchema> config-used;
};
```

There are no H2O extensions to the ODL class interface declarations. (The use of the class type `alternative` in an attribute of class `Profile` will be discussed shortly.) For the sake of brevity we have omitted the method declarations for the above classes. Method signatures will be given below as needed. In addition, the interfaces of the other classes given in Figure 2 are omitted.

The H2O[ODL] also extends the ODMG'93 ODL by including a syntax for the specification of subschemas. The following example defines three subschemas. `ProcessSchema` groups the classes which define a process model, namely `ProcessModel` and `ProcessNode`. Subschema `ProfileSchema` consists of the single class `Profile`. Instances of `Profile` capture data about executions of a specific process model. The final subschema `ProcessAndData` is actually the entire schema, defined as the union of the previous two subschemas. When specifying subschemas, it is not necessary to explicitly mention subclasses of already included classes.

```
subschema ProcessSchema { ProcessModel, ProcessNode };
subschema ProfileSchema { Profile };
subschema ProcessAndData { ProcessSchema union ProfileSchema };
```

Alternatives and deltas are generic types parameterized to operate over any declared subschema. Declaration of a subschema results in the implicit instantiation of an `alternative` class and a `delta` class whose instances operate on that subschema. We use the syntax of C++ templates; the subschema over which instances of these classes operate must be supplied when the class is declared.

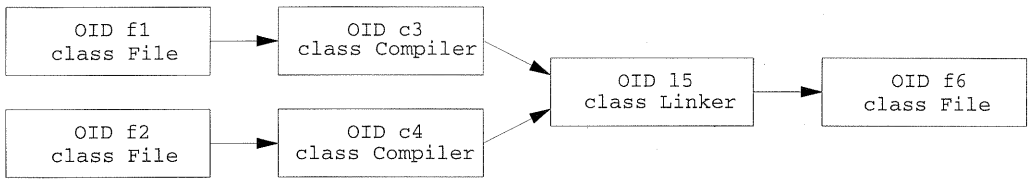


Figure 3: Process model in alternative `config[0]`

The following code defines arrays of alternatives and deltas which operate on the `ProcessSchema` subschema.

```
alternative<ProcessSchema> config[4];
delta<ProcessSchema>       delta[4];
```

Since the `alternative` and `delta` data types are classes, instances of these types can be created and destroyed using the C++ `new` and `delete` operators. They can also be manipulated through pointer values, arrays, and other C++ constructs. As illustrated in the class `Profile` above, they may serve as attribute values.

3.2 Alternatives

We consider now alternatives and the operators provided for them, and indicate some applications of them. There are two basic operators for alternatives: `under`, which permits execution of code in the context of an alternative; and `spawn`, which creates a new alternative from an existing one. The H2O constructs for manipulating deltas and alternatives are part of H2O[OML]. Some constructs such as `under` for alternatives and `when` for deltas can also be used in H2O[OQL]. The examples given below will use the following variable.

```
ProcessModel theModel;
```

We assume for this example that the database state currently holds only one object `m0` of type `ProcessModel`, and it has no start nodes and no terminal nodes.

The examples below will use the following functions:

- `InitProcessModel()` – Initializes the process model `m0` to the state shown in Figure 3.
- `AddPreprocessor()` – Adds a preprocessor node between the node holding OID `f2` and the node holding OID `c4`.
- `ChangeCompiler()` – Replaces the compiler `c4` by the compiler `c8`.

An alternative is created by executing a database update inside a `spawn` command. The following code creates an alternative in which the process configuration appears as in Figure 3. The keyword `root`

indicates the actual state of the subschema in the database; we are assuming for this example that `root` holds only one object, namely `m0`.

```
from root spawn config[0] { InitProcessModel(); }
```

The state of the database when viewed through alternative `config[0]` is given below. We shall use a simple notation for describing states: each object is described by its class, followed by its OID and a list of attributes in parentheses. The attributes are listed in the order specified in the object's interface declaration. The first object given above is a `File` with OID `f1`. Its set of `input` nodes is a null set and its set of output nodes is the set consisting of the `Compiler` object with OID `c3`.

```
database objects:
        File(f1: { }, { c3 })
        File(f2: { }, { c4 })
        Compiler(c3: { f1 }, { l5 })
        Compiler(c4: { f2 }, { l5 })
        Linker(l5: { c3, c4 }, { f6 })
        File(f6: { l5 }, { })
        ProcessModel(m0: { f1, f2 }, { f6 })
```

An alternative state can be accessed by the `under` operator. Suppose that variable `one_node` holds the OID `f2`. The following code accesses the `output` nodes of object `f2` in alternative `config[0]`. Thus, it returns the set containing OID `c4`.

```
under config[0] {
        Set<ProcessNode> fileOutputs;
        fileOutputs = one_node.outputs;
}
```

Importantly, the `under` operator permits a programmer to use (possibly large) existing code fragments in the context of a selected alternative. Furthermore, the specific alternative that a code fragment is run against can itself be determined at runtime. Code executed within an `under` may modify the associated alternative. For example, the expression `under config[0] { AddPreprocessor() }` has the effect of modifying the value of the alternative identified by `config[0]`.

Alternatives can be spawned from any pre-existing alternative. The following code creates a new alternative by inserting a preprocessor node in front of the compiler with OID `c4` in the process configuration which exists under alternative `config[0]`.

```
from config[0] spawn config[1] { AddPreprocessor(); }
```

The database state described by alternative `config[1]` is:

```
database objects:
        File(f1: { }, { c3 })
        File(f2: { }, { p7 })
        Compiler(c3: { f1 }, { l5 })
        Compiler(c4: { p7 }, { l5 })
        Linker(l5: { c3, c4 }, { f6 })
        File(f6: { l5 }, { })
        Preprocessor(p7: { f2 }, { c4 })
        ProcessModel(m0: { f1, f2 }, { f6 })
```

One of the important benefits of alternatives is the ability to easily switch between different hypothetical states of the database. The following code creates four alternative configurations of the process model and then executes each configuration and records a profile of its execution. (We are assuming here that the class `ProcessMode` includes a method `Execute` for creating the trace of a configuration and a method `time_used`) for determining the cpu time used.)

```
from root spawn config[0]      { InitProcessModel(); }
from config[0] spawn config[1] { AddPreprocessor(); }
from config[1] spawn config[2] { ChangeCompiler(); }
from config[0] spawn config[3] { ChangeCompiler(); }

Profile run_results[4]
for (i=0; i<4; i++)
        under config[i] { run_results[i].trace = theModel->Execute();
                          run_results[i].config_used = config[i];
                          run_results[i].cpu_time = theModel->time_used(); }
```

A much richer application of alternatives is to use them to store different versions of a process program (that might be represented using Petri-nets or some other formalism). A table could be maintained that indicates, for each execution of the process program, which alternative should be used when moving that execution through different steps of the program. An execution might use a fixed alternative for its lifetime, or might use newer alternatives as they become available, if they are compatible with the history of the execution so far. Because multiple executions will be occurring concurrently, the multi-state manager must switch contexts between different alternatives.

3.3 Deltas

Deltas are useful in contexts where differences between states or alternatives need to be analyzed. Two primary applications are in active modules and databases, where rule conditions examine deltas, and in long transactions, where (partial) updates need to be compared for conflict and possibly merged. Also, in the current prototype implementation of the H2O DBPL, deltas are implemented as a primitive, and alternatives are implemented using them.

There are five basic operators involving deltas: (a) *deltification* and (b) *inverse deltification*, which provide a straight-forward mechanism for creating delta values. (c) `apply`, which applies a delta to a given

state or alternative. (c) `when`, that allows hypothetical access to deltas (the expression E `when` δ yields the value that side-effect free expression E would take in the state or alternative that would arise, if delta δ were applied to the current state; this may be `root` or an alternative specified by the context. Finally, (e) `smash`, a binary operator on deltas that yields a new delta corresponding to the effect of applying the first delta followed by applying the second one. Operators for merging deltas are also of interest, and are typically application dependent.

In the following we use the same example as for alternatives, primarily for the sake of brevity. It should be stressed, however, that deltas can be used quite independently of alternatives.

A delta may be defined by using the *deltification* operator. Any operation enclosed in `[< >]` brackets is hypothetically executed and the delta value required to reach that hypothetical state is returned. The following code creates a delta which corresponds to the changes required to reach alternative `config[1]` from alternative `config[0]`.

```
under config[0] { delta[1] = [< AddPreprocessor(); >]; }
```

Inverse deltification, denoted by `[> ... <]`, is also supported. For example, the code

```
delta<ProcessSchema> inv_delta;
under config[0] { inv_delta = [> AddPreprocessor(); <]; }
```

will update the alternative referred to by `config[0]` according to `AddPreprocessor`, and record in `inv_delta` a delta value that will map this new alternative back to the original one.

In H2O, delta values are sets of atomic inserts, deletes and modifies. The value of `inv_delta` is:

```
inv_delta = {
        mod File(f2: { }, { c4 }),
        mod Compiler(c4: { f2 }, { l5 }),
        del Preprocessor(p7)
        }
```

Intuitively, a delta value can be viewed as a partial evaluation of a code fragment. This is useful if a delta is to be accessed repeatedly, as might arise in a series of hypothetical inquiries against the delta, or as might arise in the context of rule application in an active database [GHJ94]. In such cases, the code fragment is partially evaluated only once, and then repeated access to the delta value can be optimized.

The changes specified by a delta value can be realized by the `apply` operator. Thus alternative `config[1]` could have been created by the following code.

```
from config[0] spawn config[1] { apply delta[1]; }
```

The changes required to reach the four hypothetical states that were defined above using alternatives above can be captured using deltas as follows:

```
delta[0] = [< InitProcessModel(); >]
delta[1] = [< AddPreprocessor() when delta[0]; >];
delta[2] = [< ChangeCompiler() when (delta[0] smash delta[1]); >];
delta[3] = [< ChangeCompiler() when delta[0]; >];
```

As an example, `config[2]` can be obtained from `root` by first applying `delta[0]` and then applying `delta[1]`. Note the use of the `smash` operator in `delta[3]`; this creates a delta corresponding to the updates of `delta[0]` followed by the updates of `delta[1]`. (Another operator, `compose`, is also needed in some contexts; we omit discussion of this here due to space limitations. See [GHJ94].)

The `when` operator provides hypothetical access to the state that would result if a delta were applied. The following code is equivalent to accessing the outputs of the node `one_node` under alternative `config[1]`.

```
under config[0] {
        Set<ProcessNode> fileOutputs;
        fileOutputs = one_node.outputs when delta[1];
}
```

A different value for `fileOutputs` would be obtained if `delta[3]` were used in place of `delta[1]`.

We now illustrate how deltas can be used in connection with combining two or more proposed updates to a state or alternative. Suppose that two different engineers are updating `config[1]`, one by applying `AddPreprocessor()` and the other by applying `ChangeCompiler()`. In general it is difficult or impossible to determine whether two proposed updates conflict with each other, when they are specified as procedures. In contrast, suppose that the two delta values corresponding to the application of these procedures, i.e., `delta[1]` and `delta[3]`, are computed. Using a binary *merge* operator that takes into account the semantics of the subschema `ProcessSchema` (space limitations prohibit a formal definition of this operator here), these can be merged to form a new delta that corresponds to the union of these changes. The result of such a merge is given as `merge_1_3`. (The value of `delta[1]` was given above).

```
delta[3] = {
        mod Linker(l5: {c3, c8 }, { f6 }),
        mod File(f2: { }, { c8 }),
        ins Compiler(c8: { f2 }, { l5 }),
        del Compiler(c4)
        })
merge_1_3 = {
        mod File(f2: { }, { p7 }),
        ins Preprocessor(p7: { f2 }, { c8 }),
        mod Linker(l5: { c3, c8 }, { f6 }),
        ins Compiler(c8: { p7 }, { l5 }),
        del Compiler(c4)
        })
```

Importantly, the two alternatives `config[1]` and `config[3]` alone cannot be used to compute the net change to `config[0]` that corresponds to the union of `delta[1]` and `delta[3]`. This kind of delta merging is especially relevant in the context of long transactions, where it is inappropriate for any one user to lock substantial portions of the database for long intervals.

We now present one form of interplay between alternatives and deltas. In some applications involving many alternatives, it is useful to provide an operator `diff` on pairs of alternatives, where `diff(a1,a2)` returns the delta value that would map alternative `a1` to alternative `a2`. In the general case, computing `diff` may be prohibitively expensive. However, if the alternatives are built up from a common `root` using the `spawn` operator, then a reasonably efficient implementation of `diff` can be supported. In particular, a tree can be maintained that corresponds to the history of how the alternatives where `spawn`-ed. For each edge of the tree, two deltas are maintained, one being the deltification of the code that created the new alternative, and the other being the inverse-deltification of that code. Now the `diff` between two alternatives can be computed by `smash`-ing the sequence of deltas and inverse deltas on the path from the one alternative to the other.

3.4 Implementation of the Multi-State Management Service

To further indicate the semantics of deltas and alternatives, we give a brief sketch of the current prototype implementation of the H2O multi-state management service. The implementation includes a library defining a context manager and a preprocessor that modifies the implementation of all classes that are included in any declared subschemas. The preprocessor also translates H2O[*xxx*] code into the DBPL of the underlying database system. Figure 4 shows some of the objects that our prototype would create in connection with the preceding examples. In this figure we have used variable names to denote some of the deltas and alternatives, OIDs are actually used to identify all of these.

The H2O library defines a `ContextManager` class that maintains global information on the status of delta and alternative creation and application. In our prototype, alternatives are implemented using a tree, where the path from `<root>` to an alternative indicates the sequence of deltas that should be applied to build the alternative. For example, in Figure 4, the alternative `<config[2]>` is computed in our prototype from system-generated deltas having OIDs `d56`, `d57`, and `d58`. In this example `<root>` is the only alternative that serves as a "base" from which other alternatives are constructed; our prototype implementation permits multiple "base" alternatives.

To prepare objects for versioning as arises in both alternative and delta creation, the H2O preprocessor modifies the implementation of all classes that are part of declared subschemas. For each of these classes a *surrogate* class is defined that maintains the same interface as the original class. Instances of the surrogate classes maintain object identities and operate on behalf of all versions of the object which share that identity. The surrogate object maintains a *delta map* that identifies the version of an object for all program- or system-defined deltas. Figure 4 shows the surrogate and version objects for the `File` object with OID `f2` that result from the preceding examples. When responding to any query or method, the surrogate object

accesses the global context information in the context manager. This context information determines which deltas need to accessed to get the correct value for the object in the context of currently relevant alternatives and/or deltas.

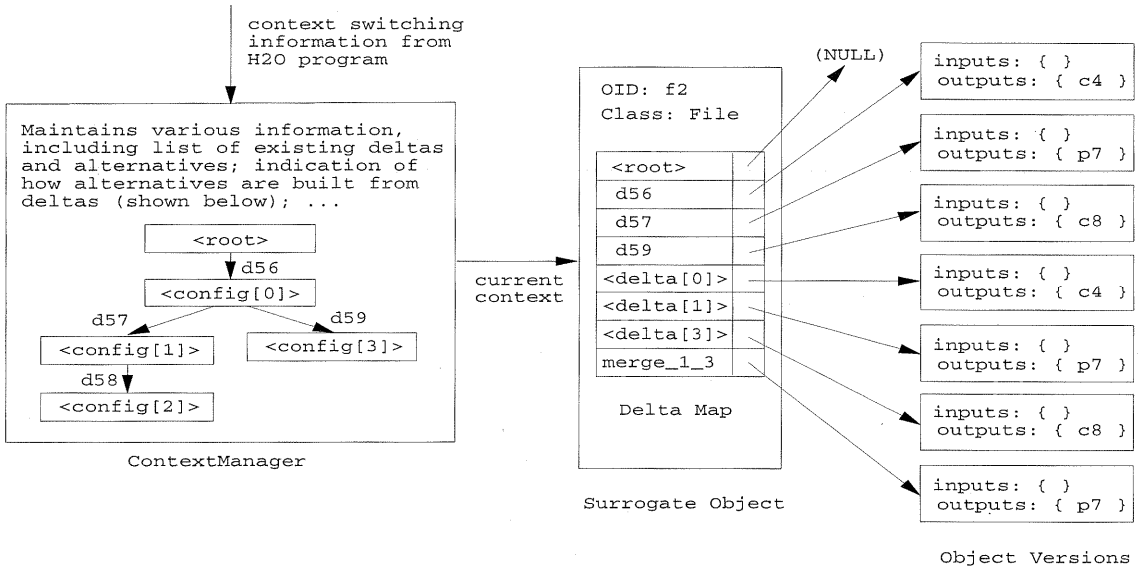


Figure 4: Context manager and object versions

4 Using Active Modules for Database Interoperation

This section briefly illustrates how active modules can be constructed using H2O and the Amalgame toolkit. The examples presented are taken from the domain of database interoperation; we have also developed examples in the domain of software engineering [Dal94].

In terms of an OSA-compliant architecture, active¡ modules can be viewed as application objects that use the activeness service in the context of a particular application. The kernel of an active module typically includes (i) a rule base that specifies most of the desired semantics in a relatively declarative fashion, (ii) an execution model for applying the rules, and possibly (iii) a local persistent store. These can be viewed as independent components that are running on a single system, or on distributed systems. Communication and other necessary interoperation features for active modules are implemented using the Amalgame toolkit.

The different parts of the kernel of an active module can be written using the H2O DBPL. Active modules differ from most active database systems currently described in the literature in two important ways. (a) The first of these is that at a semantic level, the H2O DBPL permits the specification of a wide variety of execution models (e.g., simulating the execution model of STARBURST or POSTGRES). This is useful because different execution models appear to be better suited for different applications [CCCR+90, Dal94]. One of the primary reasons that the H2O DBPL can be used to specify a wide variety

of execution models is because it supports alternatives and deltas (see [GHJ94]). In many cases, specific execution models will be developed and maintained as common facilities, and used by many applications. (b) The second fundamental difference is at the systems level. Active modules are not integrated into a DBMS, but rather can be developed and used as independent components working together with essentially arbitrary database and/or software systems. To be most effective, active modules typically assume that the other systems can automatically generate messages, indicating state changes and other important events that are sent to the active module, i.e., that the other systems have minimal activeness capabilities. When working with legacy systems, this minimal activeness might be implemented by lightweight Amalgame wrappers; alternatively, the active module might periodically poll a legacy system to learn about important events.

We now present an example that illustrates several aspects of active modules. This example is motivated in part by a practical need for supporting interoperation between the Pleiades object manager [TC93] and the TI Open OODB. Pleiades is an Object Management System being developed by the Arcadia team at UMass, and tailored towards supporting Software Engineering Environments. The TI Open OODB is an extensible component-based database system. We focus here on an example with very simple semantics, but it will be clear that much richer semantics, as might arise in a software engineering or real database context, can be supported.

The example assumes that a *Student* database is implemented using Pleiades and an *Employee* database is implemented using the TI Open OODB. The schemas of the two databases are shown in Figure 5. We suppose for the example that the *Employee* database wishes to import information about students (say, their `majors`) that happen to be employees. An active module, called here *SE_Mediator*, will monitor the contents of the two databases, and keep the *Employee* database informed about relevant changes to the student database. Subsection 4.1 below discusses how the semantics of *SE_Mediator* are implemented, and Subsection 4.2 discusses how the Amalgame toolkit is used to integrate *SE_Mediator* into a runtime environment with the two databases.

```
interface Student {
    extent Student;
    attribute string     studName;
    attribute integer[9] studID;
    attribute string     major;
    attribute string     local_address;
    attribute string     permanent_address;
};
```

```
interface Employee {
    extent Employee;
    attribute string     empName;
    attribute integer[9] SSN;
    attribute string     dept;
    attribute string     address;
};
```

Figure 5: Schemas of *Student* and *Employee*

4.1 Capturing Semantics in an Active Module

A host of issues are raised when attempting to integrate information from diverse databases in a practical context. We focus on two fundamental semantic issues here:

(a) Ways to identify corresponding pairs of objects from the *Student* and *Employee* databases, and how to maintain this information efficiently.

(b) (Because approaches to item (a) involve having the active module store some data replicated from the two source databases), ways to incrementally maintain replicated data.

Other semantic issues that are easily supported using active modules are constraint maintenance, and notification of the source databases if certain conditions arise.

With regards to (a), in some cases the problem of identifying corresponding pairs of objects from different databases can be straightforward, while in other cases this can be intractable or even impossible. We shall consider a relatively straightforward case here. Specifically, we assume that a student object and an employee object correspond (i.e., refer to the same person in the world) if their names are "close" to each other according to some metric (for instance, if middle names or initials are present or absent), and if the address as employee matches either the local or permanent address as student.

Determining corresponding pairs of objects, such as those in the class `Matches`, can be quite costly. It is often cost-effective to materialize and maintain information about the correspondences once found. On the other hand, the information about students to be imported by *Employee* might be supported in a virtual or a materialized fashion.

In a general design to support the integration of object classes from two databases, the local persistent store of the active module will hold three categories of classes, namely *Export* classes, a *Matches* class, and *Auxiliary Data* classes. For this example, the *Auxiliary Data* classes include: `Stud-Emp` (to be read "student minus employee"), with all attributes relevant to forming a match, namely `studName`, `local_address`, and `permanent_address`. (We assume here that `studName` serves as a key for the class `Student`, otherwise some key information should also be included in `Stud-Emp`.) `Emp-Stud` will have attributes for `empName` and `address`. The single *Export* class is `Stud&Emp`, with attributes `studName`, `empName`, and `major` Class `Matches`'s attributes are the union of the attributes of `Stud-Emp` and `Emp-Stud`. Speaking intuitively, `Matches` will hold one object for each person who is both a student and an employee; `Stud-Emp` will hold one object for each student in *Students* who is not an employee; and analogously for `Emp-Stud`. `Stud&Emp` will hold one object for each object in `Matches`, along with the `major` and `department` attributes. (In practice, `Matches` and `Stud&Emp` might be combined into one class.)

How can we maintain the four classes in the active module *SE_Mediator*? There are two basic issues: (i) importing information from the two source databases and (ii) correctly modifying the contents of the four classes to reflect changes to the source databases. With regards to (i), we assume that *SE_Mediator* has ports assigned to both source databases over which updates are reported. (The details of this are

discussed in Section 4.2 below.) A rule-base can be developed to perform (ii). Some representative rules for doing this, written in a pidgin H2O rule language, are

```
R1:
on message_from_Student_database
if insert Student(x: sn,sid,maj,ladd,padd)
then [insert Stud-Emp(new: sn, ladd, padd);
      pop Student_database_queue];

R3:
on insert into Matches(x: sn,ladd,padd,en,add)
if TRUE
then [insert Stud&Emp(new: sn, en, _)];
```

```
R2:
on insert into Stud-Emp(x: sn, ladd, padd)
if (exists Emp-Stud(y: en, add) &&
      sn is "close" to en &&
      (add=ladd || add = padd)
then [delete Stud-Emp(x);
     delete Emp-Stud(y);
     insert Matches(new: sn,maj,en,dept)];
```

The full rule base would include rules for finding the `major` of matched student employees (assuming these are materialized), for communicating changes to `Stud&Emp` to the *Employee* database, and for handling inserts, deletes and modifies arising from either source database. Although the above rules only consider individual objects, the execution model might apply the rules in a set-at-a-time fashion.

In general, very complex heuristics can be incorporated into the rulebase for performing the matching activity. Some heuristics might be history-dependent (e.g., once a student employee match is made, then the requirement that addresses match can be dropped). Because of its declarative nature, it is relatively easy to modify the rulebase to reflect changes to the heuristics for identifying matches.

We now briefly consider issue (b), that of incrementally maintaining the replicated data in the active module *SE_Mediator*. In our example the replicated data are essentially projections of classes from the source databases, although much more complex restructures can be supported. Maintaining replicated data is closely related to the problem of incremental maintenance of materialized views. Indeed, [Cha94, CW91] provide some solutions to this problem that use active database technology. In both papers, it is assumed that the source database is a full active database system; as noted above, active modules can be used in connection with databases having only very modest active capabilities, or no active capabilities at all. Active modules can maintain replicated data in a variety of ways, ranging from periodic polling of the source databases, to polling the source databases based on simple messages received from them, to receiving explicit bulk deltas from the source databases. This is explored in more detail in [ZHK95].

4.2 Integrating active modules into a runtime environment

We will now show how to integrate the semantics of the active module described above into a runtime environment. We first describe the desired run-time architecture and illustrate the use of Amalgame to generate a corresponding distributed program.

As mentioned earlier, we are interested in generating an active module which acts as a mediator between Pleiades and TI applications sharing a replicated data subset maintained in the local persistent store of *SE_mediator*. Our Pleiades application manipulates a student database for a University, while

the TI Open OODB application manipulates an employee database for the same University. Based on the semantics described in the previous section, our active module will monitor changes to the student database, determine whether these changes also affect the employee database.

Due to space constraints, we will not describe the construction of the Pleiades and TI applications. These applications, as well as the H2O active module, are encapsulated into Amalgame application objects. Both applications can communicate asynchronously with the H2O active module. The Pleiades application performs updates to the student database and notifies *SE_mediator* accordingly; *SE_mediator* propagates relevant updates to the employee database. The TI Open OODB application acts as server which can process update messages forwarded by *SE_mediator*. It is clear that this scenario could be extended to accommodate Pleiades and TI applications performing simultaneous updates.

The applications use the transaction mechanisms provided by the Pleiades and TI systems to ensure data integrity. To simplify our presentation, we assume that messages are always processed successfully so that our application can operate in a non-blocking and fully asynchronous execution mode. Figure 6 illustrates the role of our active module in the replication scenario described above.

As described in the previous section, H2O provides three components to implement the semantics of our active module. These components are: (1) a rulebase, (2) an execution model engine, and a (3) a local database. In our experiment, the rulebase is implemented as a file which contains the various rules specified in the previous section. This proposed implementation facilitates dynamic modifications to the rulebase, however it is clearly unefficient and only appealing in the scope of a prototype.

The execution model engine and the local database access module are provided as H2O executables programs. The execution model engine is implemented as a procedure which is activated upon receipt of a message from *SE_mediator* the H2O active module. The signature of this procedure is as follows:

```
execution_model_engine(execution_model:String,
                       rulebase:String,
                       local_store: String,
                       message_parameters:Parameter_Types)
```

where `message_parameters` is a place holder for application dependent parameters such as `request_type`, `student`, `employee`, `target_database`, and `source_database`. The execution model engine provides a workspace to hold instances of the variables involved in the various update requests. It also provides an H2O rule language interpreter for evaluating and executing the various rules. The execution model engine controls the execution of rules extracted from the rulebase. The various rules operate directly on top of the local database.

Based on the above H2O components, we can use the Amalgame toolkit to generate an active module that implements the functionality illustrated in Figure 6. Basically, the Amalgame toolkit provides an object service and a set of common facilities to support the integration of new or existing application components. The Amalgame object service supplies a high level specification language, called ASL. It

also provides an ASL interpreter that operates on top of a common facility called the Amalgame engine. An Amalgame engine provides the semantics for representing, storing, and interconnecting heterogeneous application components. Amalgame engines can be implemented in different ways. In this section, we are using the original Amalgame engine. In section 5, we describe an "active"" version of the Amalgame engine which can also reason about valid component interconnections. The Amalgame ASL script for specifying and generating the runtime scenario described above is presented in Figures 7, 8, 9.

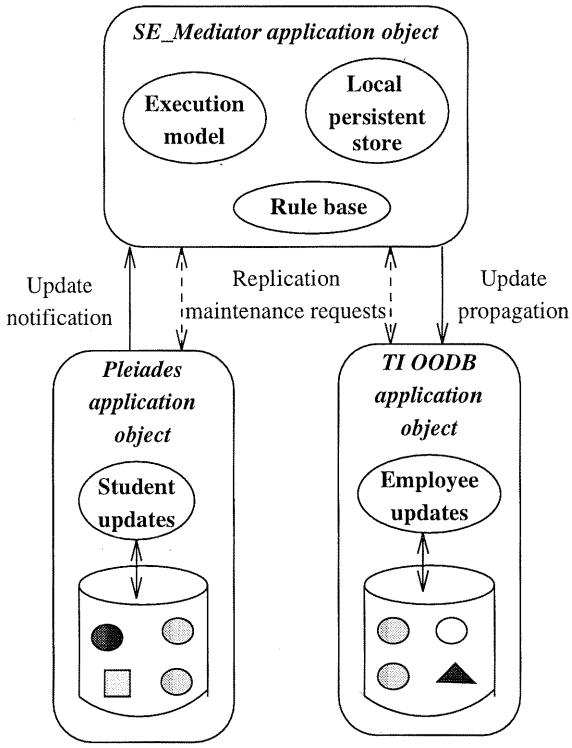


Figure 6: A Run-time scenario for replicating data across heterogeneous databases

Figure 7 illustrates the ASL commands that create an Amalgame environment to contain the class of H2O active modules. Active module subcomponents are composed of ASCII, C, or C++ components. An H2O active module interface is specified using IDL.

Figure 8 illustrates the specification of resulting H2O active component uses TCP sockets, Q, and is written in C++.

Figure 9 shows the specification of the various subcomponents of an H2O active module. Pointers to H2O binaries and files are provided. The C++ implementation for the "request" method of *SE_mediator* is also specified.

Finally, the following ASL command generates an application object instance *SE_mediator*.

```
Generate Runtime    A_H2O_MODULES
As                  SE_mediator
```

```
Attach Environment H2O_MODULES
With               "UNAME:SunOS-demo-4.1.3-2-sun4c",
                   "ISL:IDL",
                   "USL:ASCII,C,C++"
To                 "A_H2O_MODULES"
With Interface     |- >
                   interface Student {
                   (...)
                   }
                   interface Employee {
                   (...)
                   }
                   interface A_H2O_MODULES {
                   request(request_type:Integer,
                           student:Student,
                           employee:Employee,
                           source_database:String,
                           target_database:String
                   ) raises (request_failed);
                   }
                   < -|
```

Figure 7: Creation of an Amalgame environment

5 Using H2O Multi-state and Activeness Capabilities to Support the Integration of Software Systems

In the previous sections, we illustrated H2O capabilities via separate examples. We first demonstrated the use of H2O multi-state capabilities through a simple process model for testing software artifacts. Then, we illustrated the use of H2O active capabilities to support database interoperation. In this section, we describe and illustrate how H2O multi-state and activeness capabilities are being combined to support enhancements to the Amalgame application integration framework.

In its original implementation, Amalgame provides a high level specification language, called ASL, to assemble heterogeneous programs from new or existing pieces of software, which may of may not have been designed to interoperate. For Amalgame to use existing pieces of software, these must first be encapsulated into Amalgame "components" using an ASL command. The encapsulation process consists in specifying a class which describes the component to encapsulate, using the underlying Amalgame data definition language. The corresponding piece of software is then mapped into an instance of that class. As a result, the encapsulation process creates Amalgame application schemas which act as containers for the various software pieces which have been encapsulated. Other ASL commands can then be used to manipulate and assemble components within a single or across multiple schemas.

We are currently rewriting the Amalgame engine using the H2O DBPL. For example, H2O subschemas are used to implement Amalgame subschemas which correspond to subassemblies of components within

```
Attach Extension To     A_H2O_MODULES
Of Type                 "TRANSPORT"
Specified In            "TCPSOCKET"

Attach Extension To     A_H2O_MODULES
Of Type                 "RPC_PROTOCOL"
Specified In            "Q"

Attach Extension To     A_H2O_MODULES
Of Type                 "PROGRAMMING_LANGUAGE"
Specified In            "C++"
```

Figure 8: Specification of the Amalgame operational environment

an Amalgame schema. H2O alternatives provide a mechanism for creating multiple instances of a given Amalgame schema. For example different sort routines can be represented as different alternatives. Finally, the H2O multi-state manager is used to maintain the information required for accessing existing alternatives corresponding to a given schema.

In the original Amalgame implementation, users had to provide hand coded component wrappers, also called "modifiers", to specify context dependent transformations to apply to components so that they could play a different role in various application instances derived from a given schema. In the enhanced version of the Amalgame engine, we are using H2O deltas to represent the Amalgame modifiers. Deltas represent the differences, in terms of Amalgame modifiers, between multiple alternative instances of an application corresponding to a given Amalgame application schema. As an example, let us consider a generic sort routine C encapsulated in an Amalgame schema. To apply C to an array (resp. a list) of integers, we need to provide a variant C1 (resp. C2) that apply to an array (resp. a list). The resulting encapsulated components C1 and C2 are represented using H2O alternatives.

As users specify new application schemas by assembling various subschemas and create corresponding alternatives, it becomes necessary to check the validity of proposed interconnections in the various contexts. An H2O active component is being added to the Amalgame engine to help reason about valid interconnections. Also, H2O's hypothetical states are a useful mechanism to investigate potential valid component interconnections.

Another H2O active component is being used to support efficient context switches between alternatives by applying the appropriate deltas to the selected alternatives. Indeed, Amalgame modifiers are implemented H2O rules. Different execution models can be provided to implement efficient complex transformation functions between alternatives. The pidgin H2O rules below illustrates the alternative context-switching situation.

```
on alternative_switch_detect
if current_context = specific_alternative
then switch_to_specific_alternative;
```

```
Encapsulate From    H2O_MODULES
Type Is             "SUB_COMPONENT"
In                  "A_H2O_MODULES"
As                  RULEBASE.rulebase_1:FILE,
                    EXECUTION_MODEL.execution_model_1:BINARY,
                    DATABASE_MODULE.database_module_1:BINARY
From                "\H2O\demos\rulebase,
                    "\H2O\demos\exec_model.o,
                    "\H2O\demos\db_module.o
With Specification  |- >
                    request(request_type, student, employee,
                    source_database, target_database) {
                    execution_model_engine("SEQUENTIAL",
                                           "\H2O\demos\rulebase",
                                           request_type,
                                           student,
                                           employee,
                                           source_database,
                                           target_database);
                    return_abnormally(exception);
                    return();
                    }
                    < -|
```

Figure 9: Specification of subcomponents for the H2O module

```
apply_specific_alternative_routine;
```

Both the H2O multi-state management and activeness capabilities are used at design time by the modified Amalgame engine to create valid applications. Additionally, it is possible for Amalgame generated application to include the multi-state management and activeness module to perform context switches and reason about valid interconnections at runtime. The following example illustrates a particular case of reasoning that can be performed to partially automate the generation of software systems.

To illustrate the use of H2O's activeness in Amalgame let's consider the support of type discovery. When a method m_A (of component A) with signature σ_A is invoked by another component B, the parameters passed to m_A should match the signature of m_A. In the original version of Amalgame, application programmers had to supply a modifier for each component context so that the behavior of the component could be adapted when switching context. For instance, suppose that m_A is the call interface for a sort routine which takes an array as a parameter. It is possible to support the passing of a list as a parameter in a different alternative as long as there exists a modifier to transform the list into an array in the second alternative. An H2O active module can be used as an inference engine to detect type mismatches and automate the type translation based on information stored in an underlying rulebase. The H2O rule below shows how to handle this type of situation.

```
on call_to_method(name, sign(name), params)
if not compliant (sign(name), params))
then translate (params, sign(name))
```

An H2O active module could also be used to select an alternative sort routine to handle lists in the above example. Basically, the H2O activeness capabilities subsume the late binding of object-oriented language while providing added flexibility in terms of modeling of alternatives.

In summary, we have shown briefly how H2O is being used to implement a improved version of the Amalgame engine which is being architected as an active and multi-state module. Although the Amalgame user interface, and overall storage and management of components are unchanged, the H2O multi-state engine replaces the Amalgame transformation driven context switches by database driven context switches. Basically, Amalgame components which used to be passive are now implemented as active database objects. Alternatives can be maintained internally or derived as needed. The overall benefit of using H2O's multi-state management is overall efficiency, the existence of an underlying algebra of composition for subschemas, and the ease of programming provided by the H2O DBPL syntax. Using H2O's activeness features allows to reason about valid component interconnections using an extensible rule base. As a result, using a combination of the H2O multi-state management and activeness services provides powerful support for data and application integration.

6 Conclusion

The system described in this paper is currently under construction, with large portions already complete. In particular, the Amalgame framework is mostly operational and fully OSA-compliant. Also, the kernel of the multi-state manager of H2O DBPL has been implemented. We are currently developing the pre-processor of the H2O DBPL, and designing some specific syntaxes for specifying rules. We have also implemented [Dal94] several representative execution models and active modules using Amalgame and the Heraclitus DBPL [GHJ94], a relational precursor of H2O; we expect these modules to transition to the H2O DBPL in a fairly straightforward fashion.

We are currently focusing on multi-state and activeness services, and application objects built using them. We also plan to develop a number of common facilities that embody the multi-state and activeness services. These might include domain specific rule packages, generic execution models, capabilities for defining and selecting rule execution models and hypothetical alternatives managers, and other domain specific tools (such as query rewriting tools).

References

[Bla94] José Blakeley. Open Object Database Management Systems. In *Proceedings of the International Conference ACM-SIGMOD*, pages 520–520, Minneapolis, May, 24-27 1994.

[Buc93] A. P. Buchmann. Active Objects. Technical report, University of Darmstadt, 1993.

[Cat93] R.G.G. Cattell. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, San Mateo, California, 1993.

[CCCR+90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 225–236, 1990.

[Cha94] T.-P. Chang. *On Incremental Update Propagation Between Object-Based Databases.* PhD thesis, University of Southern California, Los Angeles, CA, 1994.

[CHCA94] C. Collet, P. Habraken, T. Coupaye, and M. Adiba. Active rules for the software engineering platform GOODSTEP. In *Proc. of the 2nd Intl. Workshop on Database and Software Engineering*, Sorrento, Italy, May 1994.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 577–589, 1991.

[Dal94] J. Dalrymple. *Extending Rule Mechanisms for the Construction of Interoperable Systems.* PhD thesis, University of Colorado, Boulder, 1994. in preparation.

[Deu90] O. Deux. The Story of O2. *IEEE Transaction on Knowledge and Data Engineering*, 2(1), March 1990.

[FK93a] J. C. Franchitti and R. King. A Language for Composing Heterogeneous, Persistent Applications. In *Proceedings of the Workshop on Interoperability of Database Systems and Database Applications*, Fribourg, Switzerland, October 13-14 1993. Springer-Verlag, LNCS. .

[FK93b] J. C. Franchitti and R. King. Amalgame: A Tool for Creating Interoperating, Persistent, Heterogeneous Components. In *Advanced Database Systems*, pages 313–336. Springer-Verlag, LNCS #759, 1993. Nabil R. Adam and Bharat K. Bhargava (Eds.).

[FKB94] J. C. Franchitti, R. King, and O. Boucelma. A Toolkit to Support Scalable Persistent Object Base Infrastructure. In *Proceedings of the Sixth International Workshop on Persistent Object System*, Tarascon, France, 5-9 Septembre, 1994. Springer-Verlag, LNCS.

[GHJ+93] S. Ghandeharizadeh, R. Hull, D. Jacobs, et al. On implementing a language for specifying active database execution models. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 441–454, 1993.

[GHJ94] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus[alg,c]: Elevating deltas to be first-class citizens in a database programming language. Technical Report USC-CS-94-581, Computer Science Department, Univ. of Southern California, 1994.

[GJ91] N. H. Gehani and H. V. Jagadish. ODE as an active database: Constraints and triggers. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 327–336, 1991.

[HJ91] R. Hull and D. Jacobs. Language constructs for programming active databases. In *Proc. of Intl. Conf. on Very Large Data Bases*, pages 455–468, 1991.

[HKBBS92] G. T. Heineman, G. E. Kaiser, N. S. Barghouti, and I. Z. Ben-Shaul. Rule chaining in MARVEL: Dynamic binding of parameters. *IEEE Expert*, 7(6):26–32, December 1992.

[MD89] Dennis R. McCarthy and Umeshwar Dayal. The architecture of an active data base management system. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 215–224, 1989.

[OMG90] OMG. The Object Management Architecture Guide. Technical report, OMG, 1990.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. ACM SIGMOD Symp. on the Management of Data*, pages 281–290, 1990.

[TC93] P. L. Tarr and L. A. Clarke. PLEIADES:An Object Management System for Software Development Environment. In *In Proceedings of ACM SIGSOFT '93: Symposium on Foundations of Software Engineering*, December 1993.

[WBT92] D. L. Wells, J. A. Blakeley, and G. W. Thompson. Architecture of an open object-oriented database system. *Computer*, 25(10):74–82, October 1992.

[ZHK95] G. Zhou, R. Hull, and R. King. Using materialization and active modules to support database integration. 1995. In preparation.