JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 7, September - October 2005

Statically Qualified Types in Timor

J. Leslie Keedy, Klaus Espenlaub, Christian Heinlein and Gisela Menger, University of Ulm, Germany
Mark Evered, University of New England, Australia

Abstract

In earlier papers qualifying types were presented as a technique for dynamically qualifying objects in a general way. This paper considers how such types can be composed statically into the definitions of other types.

1 INTRODUCTION

Qualifying types in Timor[1] represent a new kind of software component, instances of which can qualify the behaviour of other objects in a general way by means of special methods known as bracket methods. A bracket method "catches" invocations of methods of other objects ("targets") and its code can replace that of the target methods. They can use the special `body` statement to invoke the target method and thus provide code which serves as a prelude and/or a postlude for it, e.g. to synchronise, monitor or protect the object to which the target method belongs [6, 7]. In many cases bracket methods can be defined and implemented without having a direct knowledge of the types which they will later qualify. Hence we expect that they will provide a significant contribution to the emergence of a genuine software components industry, as envisaged by McIlroy [14] more than three decades ago.

Instances of qualifying types ("qualifiers") have a role analogous to adjectives in natural language which are used to qualify nouns (the "targets"), e.g. a "synchronised person" or a "protected list". Such adjectival qualification occurs in two forms in natural language. The first form is entirely dynamic. For example the nouns serving both as subject and as object of the following sentence are dynamically qualified by adjectives: "The nice old lady held the pretty little baby." The adjectives appear in noun phrases which are especially created for a particular sentence. This is loosely equivalent to using Timor qualifiers as described in our earlier papers [6, 7], whereby qualifiers can be associated with objects as the latter are instantiated, and even later in the life of the object.

[1] see http://www.timor-programming.org

Cite this column as follows: J. Leslie Keedy, K. Espenlaub, Ch. Heinlein, G. Menger, M. Evered: “Statically Qualified Types in Timor”, in *Journal of Object Technology*, vol. 4, no. 7, September - October 2005, pp. 115-137 http://www.jot.fm/issues/issue_2005_9/article5

But it is also possible to use adjectives in a dictionary-like way, to define new nouns which then statically have the characteristics of the adjectives. Such definitions also take the form of noun phrases, but in this case they have a defining character, e.g. "a student is a studying person". In this way new nouns are defined which can be used statically in other contexts without the inconvenience of having to repeat the adjectives, e.g. "my brother is a student". The purpose of the present paper is to show how qualifiers can be statically incorporated into new types whose instances automatically have the appropriate qualification(s).

There are many reasons for supporting static qualification alongside dynamic qualification. The first is convenience. For example if all the objects of a particular type in a particular program are to be synchronised, then it is convenient to define this once.

A second reason is security, since the definer of a type into which qualifiers are statically composed can be sure that the qualification is always applied on all objects of the type.

Another reason for providing a static qualification mechanism is efficiency. Static definition gives an optimising compiler the freedom, for example, to eliminate dynamic bracket scheduling from the run-time code.

A further important reason is that static qualification allows separate aspects of a problem to be tackled independently and then brought together into a single result. In this sense the technique offers an alternative approach to that found in aspect oriented programming (AOP) [10], as represented in languages such as AspectJ [11] and Aspect C++ [16].

Supporting static qualification raises a number of issues which do not arise in the case of dynamic qualification, in particular how static types can be composed from existing qualifiers and substantival types, how such types can be implemented, to what extent qualifiers have the right to invoke methods of their target, how the target should be defined in cases involving multiple qualification, etc.

In section 2 some examples of qualifying types are introduced to illustrate issues raised in following sections. Section 3 describes and illustrates type definitions which statically incorporate qualifying types. In section 4 it is shown how such types can be implemented. Section 5 discusses how the technique can be used to allow distinct aspects of a problem to be separately programmed. Section 6 provides a conclusion.

A knowledge both of qualifying types (and their special feature, bracket methods) [6, 7] and of the key features for defining and implementing types [3-5, 8, 9] in Timor is assumed in the rest of the paper.

2 SOME EXAMPLES

In order to illustrate some of the issues involved, five examples will be particularly useful: the types `RWsync`, `ACLprotecting`, `Monitoring`, `DuplControlling` and `Studying`. Type definitions and implementations for the first two of these were

presented in [7]. They respectively provide reader-writer synchronisation and protection via an access control list (ACL). In this paper three further examples are added, which are especially relevant for static type definitions. `Monitoring` simply counts in its bracket methods the number of times operations and enquiries of the target are invoked, and has instance methods which can return and optionally reset the counter values:

```
type Monitoring { // the type definition
qualifies any:
 op bracket op(...);  // increments the op counter
 op bracket enq(...); // increments the enq counter
instance:
 enq int opCount(); // returns value of the op counter
 enq int enqCount(); // returns value of the enq counter
 op int opCountWithReset(); // returns and resets op counter
 op int enqCountWithReset();//returns and resets enq counter
}
```

It can be implemented as follows:

```
impl MonitoringImpl of Monitoring { // an implementation
state:
 int opCount = 0;
 int enqCount = 0;
qualifies any:
 op bracket op(...) {
  opCount++; return body(...);
 }
 op bracket enq(...) {
  enqCount++; return body(...);
 }
instance:
 enq int opCount()  {return opCount;}
 enq int enqCount() {return enqCount;}
 op int opCountWithReset() {
  try {return opCount;}
  finally {opCount = 0;}
 }
 op int enqCountWithReset() {
  try {return enqCount;}
  finally {enqCount = 0;}
 }
}
```

The three types considered so far are pure qualifying types, in the sense that their implementations do not need access to the methods of objects which they qualify. However some qualifying types can only function correctly if they can access the public methods of their target. The remaining two examples illustrate how such types can be

defined, by combining the features of both qualifying types and attribute types [9], i.e. the two adjectival types supported by Timor.

The fourth example, `DuplControlling`, provides bracket methods capable of handling duplicates in the Timor Collection Library (TCL), cf. [4]. Here is a type definition:

```
type DuplControlling<:ELEM:> for Collection<:ELEM:> {
qualifies Collection<:ELEM:>:
 enq void insert(ELEM e) throws DuplEx;
 enq void insertAtPos(ELEM e, int pos) throws DuplEx;
}
```

The `for` clause indicates that `DuplControlling` is an attribute type [9]. This means inter alia that its methods can use the pseudo variable `base` to access the public methods of its attribute base (in the present example any subtype of the abstract type `Collection<:ELEM:>`).

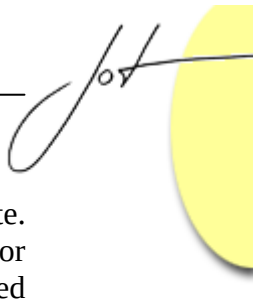
The `qualifies` clause indicates that (subtype) instances of `Collection <:ELEM:>` can be qualified by instances of this type. In particular the insertion methods of a collection object can be bracketed by bracket methods of `DuplControlling`.

Here are two implementations. The first simply ignores duplicates while the second treats duplicates as an error and throws an exception.

```
impl DuplIgnore<:ELEM:> of DuplControlling<:ELEM:> {
qualifies Collection<:ELEM:>:
 enq void insert(ELEM e) throws DuplEx {
  if (!base.contains(e)) body(...);
 }
 enq void insertAtPos(ELEM e, int pos) throws DuplEx {
  if (!base.contains(e)) body(...);
 }
}

impl DuplSignal<:ELEM:> of DuplControlling<:ELEM:> {
qualifies Collection<:ELEM:>:
 enq void insert(ELEM e) throws DuplEx {
  if (base.contains(e)) throw DuplEx.init();
  body(...);
 }
 enq void insertAtPos(ELEM e, int pos) throws DuplEx {
  if (base.contains(e)) throw DuplEx.init();
  body(...);
 }
}
```

The basic idea is that target collections which are designed to accept duplicates can be transformed into collections which are duplicate free, either by ignoring attempts to insert



duplicates or by throwing an exception when an attempt is made to insert a duplicate. Thus the TCL type `Bag` can be transformed either into a `Set` (first implementation) or into a `Table` (second implementation). Similarly the TCL type `List` can be transformed into an `OrderedSet` (first implementation) or an `OrderedTable` (second implementation). Finally the TCL type `SortedList` can be transformed into a `SortedSet` (first implementation) or a `SortedTable` (second implementation).

Our final example is a type `Studying`, which has been modified from the description in [9] to test via a bracket method that persons with a `Studying` attribute are at least 17 years old.

```
type Studying for Person {
instance:
 String uni;
 Date matriculationDate;
 enq int ageAtMatriculation();
 enq String toString();
qualifies Person:
 enq Date dob(Date dob) throws SemanticError;
maker:
 init(String uni, Date matriculationDate)
                                  throws SemanticError;
}
impl StudyingImpl of Studying {
state:
 final int minAge = 17;
 String uni;
 Date matriculationDate;
instance:
 enq int ageAtMatriculation() {...}
 enq String toString() {...}
qualifies Person:
 enq Date dob(Date dob) throws SemanticError {
  if (Date.yearDifference(matriculationDate, base.dob)
      < minAge) throw SemanticError.init();
 }
maker:
 init(String uni, Date matriculationDate)
                                  throws SemanticError {
  if (Date.yearDifference(matriculationDate, base.dob)
      < minAge) throw SemanticError.init();
  this.uni = uni;
  this.matriculationDate = matriculationDate;
 }
}
```

This example illustrates how semantic constraints can be checked using bracket methods, when an attribute is added to its base (i.e. when the attribute's maker is invoked) and when methods of the base are called. It also illustrates why it is important in Timor that

"variables" declared in type definitions (here `Date dob` in `Person`, cf. [9]) are actually considered to be method pairs [5, 8]). Note that `SemanticError` is an unchecked exception. New checked exceptions cannot be added in bracket methods (cf. [7]).

In [9] we observed that attribute types are usually behaviourally conform [13] with their attribute bases. Examples such as this (where an attribute applies constraints on the attribute base (or other attributes) are clearly an exception[2], because from a client's viewpoint he does not normally expect a restriction on an item such as the age of a person, since this is not defined as an exception in the definition of the type `Person` (which is why an unchecked exception is used - a situation which often arises with bracket methods). This is a deliberate policy. To insist that constraints associated with all attributes are defined in the attribute base (and possibly in other attributes for the same base) would inhibit the modularity and the flexibility of attributes, since they would then all have to be defined at the latest when the attribute base is defined.

In the following sections we discuss various issues which arise out of these examples, beginning with the way they can be integrated statically into other types and their implementations.

3 STATIC TYPE DEFINITIONS

Like Timor's attribute types, qualifying types have a role in programs similar to that of adjectives in natural language. Just as adjectives extend and modify the meaning of nouns, so attributes extend and qualifiers modify the semantics of objects in a modular way. As the last two examples in the previous section illustrate, a type can serve as both an attribute and a qualifier for the same base object. Hence these must be treated uniformly at the syntax level in definitions of other types (i.e. in clauses which can be regarded as equivalent to noun phrases, corresponding in the following grammar to `inheritedItems`). The basic structure of such "adjectival" type definitions has already been outlined informally for attributes in [9]. Part names can be used, but are not explicitly illustrated here, except where this is necessary to handle repeated inheritance.

More formally, the syntax can be understood in terms of the following EBNF:

```
derivationClause = ( "extends:" | "includes:" ) inheritedItems.
inheritedItems   = { [qualifyingList]  simpleItem ";"
                   | qualifyingList  compositeItem ";" }.
qualifyingList       = "{" inheritedItems "}".
compositeItem    = "(" inheritedItems ")".
```

The basic bracketing rule is that the public methods of `inheritedItems` (whether simple or composite) are qualified by the bracket methods of their `qualifyingList` (if

[2] The programmer of a Timor type has the opportunity to define subtypes which can be used polymorphically (using the keyword `extends`) and other derived types (using `includes`). This choice can also be made when composing qualifying types (such as `Studying`) statically into new types.

any), but public methods added by other items in the same `qualifyingList` are not. When a client invokes a method of a qualified item, the appropriate bracket method (if any) of each item in its `qualifyingList` is scheduled in turn (from left to right) at the point where the predecessor executes a `body` statement.

Examples of Type Definitions

Applying this syntax to qualifying types, a `SynchronisedPerson` can be defined as follows:

```
type SynchronisedPerson {
extends:
 {RWsync;} Person;
 // the simpleItem Person is statically qualified by a
 // qualifyingList containing a single qualifier (the
 // reader-writer qualifier RWsync)
}
```

This shows how the qualifier `RWsync` reader can be statically combined with another type to produce a new type with instances which are always synchronised.

It would be possible to define types for the duplicate-free collections in the TCL along the same lines, using the qualifying type `DuplControlling`, e.g.

```
type Set<:ELEM:> {
includes: // indicates that Set is not a subtype of Bag
 {DuplControlling<:ELEM:>;} Bag<:ELEM:>;
}
```

However, we consider it inappropriate to define a `Set` in terms of a `Bag`, inter alia because it would result in considerable restrictions from the viewpoint of polymorphism when compared to the definition given in [4]. As that paper illustrates, `Set` is better defined as part of a more comprehensive collection hierarchy based on the combining of orthogonal properties, as this creates more opportunities to take advantage of subtyping. However, because of the separation of types and implementations in Timor it is possible for a type definition which does not rely on qualifying types to be implemented using implementations of qualifying types. In a later section we illustrate how the first implementation of `DuplControlling` can be used to implement `Set`.

Multiple Qualification

Types bracketed by more than one qualifier are expressed as follows with the above syntax:

```
type SynchronisedMonitoredPerson {
extends:
 {RWsync; Monitoring;} Person;
}
```

In this example the type `Person` is statically qualified by both `RWsync` and `Monitoring` such that a client method invocation of a `Person` method results in the appropriate `RWsync` bracket being applied. When this executes a `body(...)` statement the appropriate `Monitoring` bracket is applied. When this in turn executes a `body(...)` statement the target method originally invoked by the client is called.

Mixing Attributes and Qualifiers

The syntax can be used to mix attributes and qualifiers, e.g.

```
type StudyingSynchronisedPerson {
extends:
   {RWsync; Studying;} Person;
}
```

In this example the bracket methods are applied to methods of the target type (i.e. `RWsync` qualifies `Person`) but not to the methods of its attributes (i.e. in this example not to the instance methods of `Studying`, which are also instance methods of the new type [9]). Since `Studying` is itself a specialised qualifier, when a client invokes the method for setting the `dob` value of `Person`, the `op` bracket of `RWsync` is first scheduled and when this invokes `body(...)`, the specialised bracket of `Studying` is run to check whether the date of birth being set is appropriate.

Adverbial Qualification

Adverbial qualification (i.e. qualification of qualifiers) is possible, because items in a qualifying list can themselves have a qualifying list, e.g.

```
type SynchronisedlyStudyingPerson {
extends:
   {{RWsync;} Studying;} Person;
}
```

Here the methods of `Studying` are synchronised, but not those of `Person`. This applies both to the instance methods of `Studying` and to its bracket methods. Thus if a client invokes the method for setting the `dob` value of `Person`, the `op` bracket of `RWsync` is scheduled to synchronise the specialised bracket of `Studying` before the `dob` method of `Person` is invoked.

Adverbial qualification which involves multiple levels of bracketing is non-trivial, because, for example, the synchronisation has to be released before the `Person` method

(which is not synchronised in this example) is actually called, and then it has to be reclaimed for the `Studying` bracket postlude. A full discussion of adverbial bracket scheduling is beyond the scope of this paper and will be the subject of a separate paper.

Composite Targets

Assuming that `Employed` is another `for Person` attribute type, the methods of `Studying`, `Employed` and `Person`, are all synchronised by `RWsync` in this example:

```
type SynchronisedStudyingEmployedPerson {
extends:
   {RWsync;} ({Studying; Employed;} Person;);
}
```

Here `Studying`, `Employed` and `Person` are grouped to define a `compositeItem` qualified by `RWsync`.

To synchronise the methods of `Studying` and `Employed` (but not those of `Person`) with the same qualifier, i.e. using the same synchronising variables, the following definition can be used:

```
type SynchronisedlyStudyingEmployedPerson {
extends:
   {{RWsync;}(Studying; Employed;);} Person;
}
```

while in the next example each attribute (but not `Person`) is synchronised, but using separate synchronisation variables:

```
type SyncStudyingSyncEmployedPerson {
extends:
 {{RWsync rws1;} Studying; {RWsync rws2;} Employed;} Person;
 // the part names are necessary for the RWsync items,
 // as described in [9].
}
```

Methods of Qualifiers

So far qualifiers have been considered from the viewpoint of their bracket methods, but types such as `RWsync` are exceptions in not having their own instance methods. For example the type `Monitoring` has instance methods which provide the monitoring agent with the monitoring details gathered by the bracket methods and which reset the counts to zero.

When defined statically in derivation clauses, such methods become additional instance methods of the target. But they are not qualified by their own bracket methods, nor by any other qualifiers defined in the same `qualifyingList`. For example instances of a type `MonitoredPerson`, defined as:

```
type MonitoredPerson {
extends:
 {RWsync; Monitoring;} Person;
}
```

have instance methods inherited from `Person` which are bracketed by the `RWsync` and `Monitoring` brackets, and instance methods of `Monitoring` which are not bracketed. This interpretation can be understood as follows. The instance methods of a qualifier can be regarded as equivalent to the public methods of an *attribute* type $\texttt{Monitoring}_i$ and the bracket methods as those of a *qualifying* type $\texttt{Monitoring}_b$. Then the type `MonitoredPerson` is equivalent to:

```
type MonitoredPerson {
extends:
 {RWsync; Monitoring_i; Monitoring_b} Person;
}
```

External Qualifiers

Not all qualifying types are semantically suitable for direct use in static type definitions along the lines described in the previous section. For example the type `ACLprotecting`, which provides access control list protection to a target object, has instance methods which set up and modify entries in the ACL. Inheriting such instance methods statically alongside the methods to be protected would normally make no sense (except perhaps for testing purposes) because clients of the new type could then manipulate the ACL to provide themselves with access to the supposedly protected methods of the target.

To accommodate such cases in an appropriate way, a `simpleItem` can be defined by reference, e.g.

```
type ProtectedPerson {
extends:
 {ACLprotecting*;} Person;
maker:
 init(ACLprotecting* anACL);
}
```

The reference must be linked via a parameter referring to an external object which is passed to a maker, as the example illustrates. This enables the invoker of the maker (i.e.

the creator of the object) to maintain a separate ACL to which he might have private access, while passing references for the protected object to other clients.

External qualifiers can be used to good effect in other cases. For example if all the `Person` activities in a program are to be synchronised and monitored as a group, the following type definition could be used:

```
type GroupControlledPerson {
extends:
 {RWsync*; Monitoring*;} Person;
maker:
 init(RWsync* rWsync; Monitoring* monitoring);
}
```

This might then be used in a program as follows:

```
RWsync* personSync = new RWsync.init();
Monitoring* personMon = new Monitoring.init();
...
GroupControlledPerson* newPerson =
 new GroupControlledPerson.init(personSync, personMon);
}
```

External qualifiers are independent objects whose methods can be accessed via references independently of the objects which they qualify.

In contrast with qualifying types inherited by value, the instance methods of an object inherited by reference do not become part of the type containing the reference. Hence clients of objects of type `GroupControlledPerson` cannot use a cast statement to gain access to their instance methods (e.g. those of `Monitoring`). But their bracket methods are applied to the methods of the appropriate target, here the `Person` item. However, in an implementation of the type an actual reference passed in (as a parameter to a maker) can be used as a normal reference. This means that an implementor of a type such as `ProtectedPerson` has to be trusted, but not the clients of the type.

If a reference item is itself adverbially qualified by some other qualifier, this has no effect, i.e. its bracket methods are not subject to the bracket methods of the other qualifier(s). However, adverbial qualification of external objects is possible, by qualifying the external object (*not* the reference), as is described in [7].

Adding New Methods in a Type Definition with Static Qualifiers

If new instance methods are added in an `instance` section to a type whose definition includes a `qualifyingList` containing qualifying types, the new instance methods are not bracketed. Technically it would be difficult to bracket these, because a `qualifyingList` qualifies a base type inherited in an `extends` or `includes` clause, not the type currently being defined. Furthermore a `derivationClause` can contain

multiple inherited bases (to enable the definition of multiple inheritance), and these multiple bases can be defined to be bracketed in different ways. Hence it would not be clear how the new methods should be qualified.

This "limitation" accords with the basic philosophy behind the design of Timor, whereby separate aspects should be handled separately. In this respect the definition of types corresponding to noun phrases should be seen as a two stage activity:

a) First, the individual components (attribute types, qualifying types and the components which they are defined to qualify) are defined and implemented.
b) Second, these should be used along the lines discussed in section 3 to compose new types, without adding new instance methods.

Redefining Methods in a Type Definition with Static Qualifiers

Inherited methods can be redefined in the usual way. Because the methods being redefined must be `inheritedItems`, their static bracketing is also defined, and the corresponding definition applies to the redefined method. However, methods which are separately bracketed cannot be merged, since it would not be clear how the merged methods should be qualified.

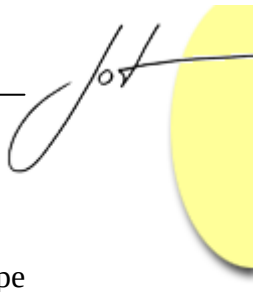
Adding New Methods and Redefining Methods in Derived Types

In accordance with the philosophy for handling new instance methods, if a type which includes static bracketing is itself extended, any new methods which are added in the derived type are not automatically bracketed with the brackets of the base type. However, any methods of the new type which redefine methods of the base type are bracketed in the same way that the original method is bracketed.

4 IMPLEMENTATIONS

In contrast with the standard OO paradigm an implementation in Timor does not automatically inherit code. It can optionally include *re-use variables* of any appropriate type. These may, but need not, be defined in terms of the type in question or in terms of a specific implementation of that type. Such re-use variables are characterised by the hat symbol (`^`) preceding the type or implementation name associated with the variable. The compiler compares the public methods of re-use variables with those methods of the type being implemented which do not appear in the `instance` section, matching them in the order of their appearance. When a match is found, the corresponding method of the re-use variable is treated as the implementation of a public method of the type being implemented.

This technique implies, for example, that a type defined in terms of qualifiers can, but need not, be implemented using implementations of these (or other) qualifiers. Conversely it implies that a type not defined in terms of qualifiers can (but need not) be implemented using qualifier implementations.



An Implementation of a Statically Qualified Type without using Qualifiers

The type `SynchronisedPerson`, defined earlier in terms of the qualifying type `RWsync`, can be implemented without re-use variables as follows:

```
impl SynchronisedPersonImpl1 of SynchronisedPerson {
state:
 Person p;
 Semaphore mutex = Semaphore.init(1);
 Semaphore readers = Semaphore.init(1);
 int readcount = 0;
instance:
 op String name(String name) {
  mutex.p();    // prelude for writer synchronisation
  p.name(name); // invoke the overridden name op of Person p
  mutex.v();    // postlude for writer synchronisation
 }
 enq String name() {
  readers.p();  // prelude for reader synchronisation
  readcount++;
  if (readcount == 1) mutex.p();
  readers.v();
  try {return p.name();} // invoke the name enq of Person p
  finally {
   readers.p();  // postlude for reader synchronisation
   readcount--;
   if (readcount == 0) mutex.v();
   readers.v();
  }
 }
 ... // similar coding for each instance method of Person
}
```

This implementation is tedious, because every method of the type `Person` has to be overridden to add the appropriate reader or writer synchronisation protocol. It is loosely equivalent to defining a subclass of `Person` in the standard OO paradigm in which each method is overridden and calls "super". We have illustrated this to show that it is possible to implement a type defined in terms of qualifying types without using qualifiers in the implementation. It is clear that a more efficient approach is desirable.

Automatic Implementations with Qualifiers

As was already described in [9], a type which consists only of derivation clauses can be transformed automatically into an implementation using the following basic rules:

a) Change each `extends` or `includes` clause into a `state` clause.

b) For each type which does not already have a part name in the type definition, add a part name (the same as the type name, but beginning with a small letter) to form a variable declaration.
c) For each type which already has a part name in the type definition, use that part name to form a variable declaration.
d) Prefix the hat symbol to each type name of a variable declaration which provides public methods.
e) Add a parameterless maker or makers which conform to the requirements for producing automatic makers (see below).

Here is an example of an automatic implementation for the type `SyncStudyingSyncEmployedPerson`:

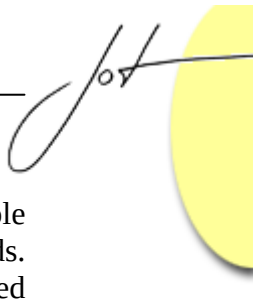
```
impl SyncStudyingSyncEmployedPersonImpl
                          of SyncStudyingSyncEmployedPerson {
state:
 {{RWsync rws1;} ^Studying studying;
  {RWsync rws2;} ^Employed employed;
 }
 ^Person person;
}
```

If explicit makers are provided in the type definition and their parameters correspond in name and type to those of the variables automatically produced, then a maker is automatically added which assigns the values of the parameters to the corresponding state variables (cf. [9]). This applies also to reference parameters. Thus the automatic implementation of `GroupControlledPerson` is:

```
impl GroupControlledPersonImpl of GroupControlledPerson {
state:
 {RWsync* rWsync;
  Monitoring* monitoring;}
 ^Person person;
maker:
 init(RWsync* rWsync; Monitoring* monitoring) {
  this.rWsync = rWsync;
  this.monitoring = monitoring;
 }
}
```

Explicitly Programmed Implementations with Qualifiers

The pattern for automatic implementations illustrates how explicit implementations of types which use derivation clauses for bracketing can be defined. However, the programmer can choose his own variable names (except where re-use variables must have the correct part name, e.g. for repeated inheritance).



Otherwise the implementation is like any other implementation. It can for example have additional instance methods and it can implement redefinitions of existing methods. As mentioned earlier, new instance methods are not bracketed by statically defined brackets, and redefined methods are handled by the compiler (from the viewpoint of bracketing) as if they were the method being redefined.

If an explicitly coded instance method invokes methods of state variables (including re-use variables) which are defined to be bracketed, the bracketing does *not* occur, i.e. `inheritedItems` define only how public methods are bracketed when invoked by a client. When invoked in an implementation no bracketing takes place.

Using Qualifiers which do not Occur in the Type Definition

It can be advantageous for an implementor to use qualifier implementations even where the type being implemented was not defined using qualifiers. In this case there is no appropriate pattern in the type definition which can be used in the implementation. Nevertheless implementations can be structured in terms of `inheritedItems` to achieve the same compositional effect as at the type level.

Thus given (any) implementations of `List` and `SortedList` (cf. [4]), the entire duplicate-free types in the TCL can easily be implemented with re-use variables of `DuplControlling`'s implementations, as follows:

```
impl SetImpl<:ELEM:> of Set<:ELEM:> {
state: {DuplIgnore<:ELEM:> di;} ^List<:ELEM:> l;
}
impl OrderedSetImpl<:ELEM:> of OrderedSet<:ELEM:> {
state: {DuplIgnore<:ELEM:> di;} ^List<:ELEM:> l;
}
impl SortedSetImpl<:ELEM:> of SortedSet<:ELEM:> {
state: {DuplIgnore<:ELEM:> di;} ^SortedList<:ELEM:> sl;
}
impl TableImpl<:ELEM:> of Table<:ELEM:> {
state: {DuplSignal<:ELEM:> ds;} ^List<:ELEM:> l;
}
impl OrderedTableImpl<:ELEM:> of OrderedTable<:ELEM:> {
state: {DuplSignal<:ELEM:> ds;} ^List<:ELEM:> l;
}
impl SortedTableImpl<:ELEM:> of SortedTable<:ELEM:> {
state: {DuplSignal<:ELEM:> ds;} ^SortedList<:ELEM:>;
}
```

The qualifiers `DuplIgnore` and `DuplSignal` are not marked as re-use variables (with the `^` symbol) because they have no instance methods which are re-used to match the interface definitions of the types being defined.

In [4] a different, more complicated technique was used to implement the duplicate free types of the TCL. In view of the relative simplicity of the above examples, which

achieve the same result in a more straightforward manner, it has been decided to abandon the more complicated `requires` clause defined there.

5 PROGRAMMING SEPARATE ASPECTS

As was mentioned in the introduction, Timor's adjectival types can be viewed as an alternative approach to AOP for programming distinct aspects of programs separately. The type `DuplControlling` and its two implementations (see section 2) illustrates a simple example of this, whereby the aspect "duplicate-freeness" is separately programmed from other aspects of collections in the TCL.

Consider now another possible aspect, relevant for collections of integers, i.e. where `Collection<:ELEM:>` is instantiated to `Collection<:int:>`. For these it might be desirable to have an additional method `enq int sum()`, which returns the sum of the values of all elements in a particular collection. (The collection might be any of the nine concrete collection types.)

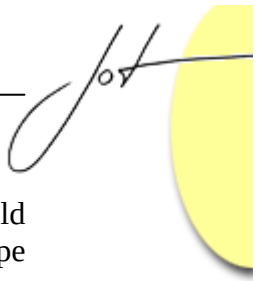
We can envisage two implementation techniques for this. The first would have an integer variable which is continuously updated with each insertion into and removal from the collection, using bracket methods; in this case the enquiry `sum` simply returns the current value of the variable. The second implementation calculates the required value when the client invokes `sum`, by iterating through all elements in the collection, which it accesses via `base`.

With the first implementation in mind it might be tempting to define the aspect as a qualifying type, as follows:

```
type IntSumming {
qualifies Collection<:int:>:
 enq void insert(int e) throws DuplEx;
 enq void insertAtPos(int e, int pos) throws DuplEx;
 enq void remove(int e) throws NotFoundEx;
 enq void removeAtPos(int pos)throws NotFoundEx;
instance:
 enq int sum();
}
```

But with the second implementation in mind an attribute type definition would be more appropriate, i.e.

```
type IntSumming for Collection<:int:> {
instance:
 enq int sum();
}
```



Type definitions are intended to be on a higher plane than implementations, and should not reveal or favour a particular implementation strategy. Hence two different type definitions should not be needed for what is logically a single type.

The appropriate way to define such types is to regard possible bracket methods as an implementation technique which need not appear at the type level. Hence the second type definition, without the bracket methods, is the more appropriate[3].

Here is an implementation of `IntSumming` which updates an internal variable whenever an integer is added to or removed from the collection:

```
impl IntSummingImpl of IntSumming {
state:
 int theSum = 0;
qualifies Collection<:int:>:
 op void insert(int e) throws DuplEx {
  int n = base.size();
  body(e);
  if (base.size() > n) theSum += e;
 }
 op void insertAtPos(int e, pos) throws DuplEx {
  int n = base.size();
  body(e, pos);
  if (base.size() > n) theSum += e;
 }
 op void remove(int e) throws NotFoundEx {
  int n = base.size();
  body(e);
  if (base.size() < n) theSum -= e;
}
 op void removeAtPos(int pos) throws OutOfBoundsEx {
  int temp = base.getAtPos(pos);
  int n = base.size();
  body(pos);
  if (base.size() < n) theSum -= temp;
}
instance:
 enq int sum() {
  return theSum;
 }
}
```

The following alternative implementation shows how the sum of the integers might be calculated at the time the client invokes the new method:

[3] At a later stage we intend to add a more formal specification technique, which might be used to enhance the type definition.

```
impl IntSummingImpl2 of IntSumming {
state:
 int theSum = 0;
instance:
 enq int sum() {
  Handle iter = base.startLoop();
  while (base.hasMore(iter)) theSum += base.next(iter);
  return theSum;
 }
}
```

In this implementation bracket methods are not used, but access to `base` is essential[4].

While in the `IntSumming` example it is not appropriate to indicate the possibility of bracketing at the type level (as this is merely an implementation alternative), in other examples it can be appropriate to define a type both as an attribute of another type and as qualifying its methods. Thus the type `Studying`, as defined in section 2, needs to indicate at the type level both its dependence (`for Person`) and its qualification of the method for setting the `dob` value. The latter is important because it warns the client that the unchecked exception `SemanticError` can be thrown.

Either of the implementations of `IntSumming` can be used to implement types corresponding to the nine concrete types of the TCL with the additional summing functionality, as the following example for `SetIntSumming` (a `Set<:int:>` which can be summed) shows:

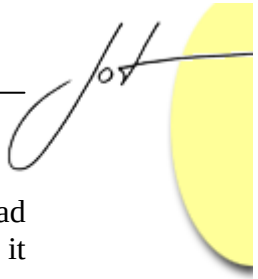
```
type SetIntSumming {
extends: {IntSumming;} Set<:int:>;
}

impl SetIntSummingImpl of SetIntSumming {
state: {^IntSumming is; DuplIgnore<:int:> di;}
                                          ^List<:int:> l;
}
```

The implementations of `IntSumming` were programmed - as separate aspects should be - without consideration for the "duplicate-free" aspect. In this example the result is that the order of the bracketing (from the viewpoint of `body` invocations) is irrelevant, because the throwing of an exception (in the case of the use of `DuplSignal`) or the ignoring of a duplicate (in the case of the use of `DuplIgnore`) in inner brackets is properly handled. In the above implementations the `IntSumming` attribute has been provocatively placed before the duplicate free qualifier to emphasize this.

A simpler version of the qualifier's first implementation, which might simply update the current sum before or after invoking `body` without ensuring (by comparing the

[4] The conventional iterator technique has been modified slightly for reasons which will be discussed in a future paper. Using the conventional technique would also require access to `base`.



number of integers in the set before and after the operation) that the operation had actually taken place, would create a problem for the above implementation, because it would increment the sum with values of duplicates which are rejected by `DuplIgnore`. (Similarly, an attempt to achieve the same effect by checking for exceptions would only work for implementations of types in which `DuplSignal` were used.) Such implementations would violate the principle that different aspects (i.e. different qualifiers) should be freely mixable and matchable, without needing a knowledge of each others' implementations.

Notice that a general purpose qualifier such as `RWsync` or `ACLprotecting`, although modularly written, could not simply be placed in any position in the qualifying list of `SetIntSummingImpl`, because in contrast with the specialised qualifiers used in this example, they have no knowledge of the semantics of the objects which they qualify, nor would they then be in a position to control access to the entire object. In order to achieve the required effects such qualifiers typically need to view the specialised object in its entirety and are therefore appropriately placed in a qualifying list which achieves this effect, typically by treating the more specialised attributes and qualifiers, together with the base item, as a `compositeItem`, e.g.

```
type SyncSetIntSumming {
extends: {RWsync} ({IntSumming;} Set<:int:>;);
}

impl SyncSetIntSummingImpl of SyncSetIntSumming {
state: {RWsync}
  ({^IntSumming is; DuplIgnore<:int:> di;} ^List<:int:> l;)
}
```

Alternatively, and more flexibly, they can be added dynamically, e.g. when an object of the type is created (cf. [7]), e.g.

```
SetIntSumming sis = new {RWsync} SetIntSumming.init();
```

In either case it is guaranteed, for example, that client accesses to the object are properly synchronised, including accesses to all new semantic routines, such as `enq int sum();` in the present example, since the additional level of bracketing guarantees that the state of the target and the state of its specialised attributes and qualifiers (e.g. the variable `theSum` in `IntSummingImpl` or in `IntSummingImpl2`) are viewed as a single state.

6 RELATED WORK

A comparison between qualifying types and other bracketing approaches, such as `inner` in Beta [12], mixins [1, 2], encapsulators [15], Java proxies and AOP [11], has already been provided in [7], and need not be repeated here.

One additional difference, which has become evident in this paper, is that Timor's qualifying types can be associated with the types which they qualify dynamically (as described in [7]) or statically, as described here, or even (as illustrated at the end of the previous section) in combination. To our knowledge no other bracketing technique supports a similar level of flexibility.

7 CONCLUSION

Qualifying types in Timor (along with attribute types) are a programming language counterpart for adjectives in natural languages. Just as natural languages can use a dynamic combination of adjectives and nouns to describe objects in the real world on a one-off basis, or can use them to define new nouns statically, so Timor's adjectival types can be combined with substantival types either dynamically or statically (or in combination). The grammar used to achieve static definitions of new types is closely analogous in structure to that of noun phrases in natural languages and adds a similar level of power and flexibility, providing a basis for a new programming paradigm, which we call component oriented programming, which builds on but goes beyond the object oriented paradigm.

ACKNOWLEDGEMENTS

Special thanks are due to Dr. Axel Schmolitzky for his invaluable contributions to discussions of Timor and to the ideas which have been taken over from earlier projects. Without his ideas and comments Timor would not have been possible.

REFERENCES

[1] G. Bracha and W. R. Cook, "Mixin-based Inheritance," ECOOP/OOPSLA '90, Ottawa, Canada, 1990, ACM SIGPLAN Notices, vol. 25, no. 10, pp. 303-311.

[2] L. G. DeMichiel and R. P. Gabriel, "The Common Lisp Object System: An Overview," ECOOP '87, Paris, 1987, Springer-Verlag, LNCS, vol. 276, pp. 151-170.

[3] J. L. Keedy, G. Menger, and C. Heinlein, "Support for Subtyping and Code Re-use in Timor," 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 35-43.

[4] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting from a Common Abstract Ancestor in Timor," in *Journal of Object Technology*, vol. 1, no. 1, May-June 2002, pp. 81-106, http://www.jot.fm/issues/issue_2002_05/article2.

[5] J. L. Keedy, G. Menger, and C. Heinlein, "Taking Information Hiding Seriously in an Object Oriented Context," Net.ObjectDays, Erfurt, Germany, 2003, pp. 51-65.

[6] J. L. Keedy, G. Menger, C. Heinlein, and F. Henskens, "Qualifying Types Illustrated by Synchronisation Examples," in *Objects, Components, Architectures, Services and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany*, vol. LNCS 2591, M. Aksit, M. Mezini, and R. Unland, Eds.: Springer, 2003, pp. 330-344.

[7] J. L. Keedy, K. Espenlaub, G. Menger, and C. Heinlein, "Qualifying Types with Bracket Methods in Timor," in *Journal of Object Technology*, vol. 3, no. 1, January-February 2004, pp. 101-121. http://www.jot.fm/issues/issue_2004_01/article1.

[8] J. L. Keedy, G. Menger, and C. Heinlein, "Inheriting Multiple and Repeated Parts in Timor," *Journal of Object Technology*, November-December, 2004 vol. 3, no 10, http://www.jot.fm/issues/issue_2004_11/article1

[9] J. L. Keedy, G. Menger, and C. Heinlein, "Diamond Inheritance and Attribute Types in Timor," *Journal of Object Technology*, November-December, 2004 vol. 3, no 10, http://www.jot.fm/issues/issue_2004_11/article2

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," ECOOP '97, 1997, pp. 220-242.

[11] G. Kiczales, E. Hilsdale, J. Hugonin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," ECOOP 2001 - Object-Oriented Programming, 2001, Springer Verlag, LNCS, vol. 2072, pp. 327-353.

[12] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard, "The Beta Programming Language," in *Research Directions in Object-Oriented Programming*: MIT Press, 1987, pp. 7-48.

[13] B. Liskov and J. M. Wing, "A Behavioral Notion of Subtyping," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811-1841, 1994.

[14] M. D. McIlroy, "Mass Produced Software Components," NATO Conference on Software Engineering, NATO Science Committee, Garmisch, Germany, 1968, Petrocelli-Charter, pp. 88-98.

[15] G. A. Pascoe, "Encapsulators: A New Software Paradigm in Smalltalk-80," OOPSLA '86, 1986, pp. 341-346.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat, "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language," 40th International

Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002), Sydney, Australia, 2002, Conferences in Research and Practice in Information Technology, vol. 10, pp. 53 - 60.

About the authors

J. Leslie Keedy is Professor and Head, Department of Computer Structures, University of Ulm, Germany, where he leads the Timor language design and the Speedos operating system design groups. His email address is keedy@jlkeedy.net. His biography can be visited at http://www.informatik.uni-ulm.de/rs/mitarbeiter/jlk/

Klaus Espenlaub received a Ph.D. in Computer Science from the University of Ulm in 2005. Currently he works as a research assistant in the Department of Computer Structures at the University of Ulm. His research interests include secure operating systems, protection mechanisms and computer architecture. His email address is klaus.espenlaub@uni-ulm.de.

Christian Heinlein received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently, he works as a scientific assistant in the Department of Computer Structures at the University of Ulm. His research interests include programming language design in general, especially genericity, extensibility and non-standard type systems. His email address is christian.heinlein@uni-ulm.de.

Gisela Menger received a Ph.D. in Computer Science from the University of Ulm in 2000. Currently she works as a scientific assistant in the Department of Computer Structures at the University of Ulm. Her research interests include programming language design and software engineering. Her email address is gisela.menger@uni-ulm.de.

Mark Evered is a Senior Lecturer in the School of Mathematics, Statistics and Computer Science at the University of New England in Armidale, Australia. He completed his PhD at the Technical University of Darmstadt in Germany. His research interests include Object-based Systems, Security, Persistence and Programming Language Design and Implementation. His email address is markev@mcs.une.edu.au.