Vertical, Horizontal, and Behavioural Extensibility of Software Systems

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract

Object-orientation often claims to support extensible and modular programming. By distinguishing three different dimensions of extensibility, however – vertical (extensions of the type hierarchy), horizontal (extensions of the spectrum of operations available on types), and behavioural (extensions or even modifications of the original behaviour of operations) –, it is shown that typical object-oriented programming languages support only one of them in a direct and modular way: vertical extensibility. Horizontal extensibility in a modular fashion can only be achieved indirectly by disciplined application of design patterns such as the Visitor pattern, while behavioural extensibility is not supported at all. On the other hand, a new and surprisingly simple concept called *dynamic routines*, that is introduced in this paper, simultaneously supports all three dimensions of extensibility in a natural and flexible way, thus eliminating the need for some clumsy design patterns and at the same time offering the programmer additional freedom. To make the concept immediately applicable in practice, it has been implemented as precompiler-based language extensions for C++, Oberon-2, and Java.

1. Introduction

It has been pointed out frequently [17, 21, 6], that inheritance *mechanisms* of typical object-oriented programming languages are fraught with a number of different and partially conflicting inheritance *concepts*, such as subtype polymorphism, code reuse, genericity, and importation, to name only a few. Consequently, numerous proposals exist to conceptually separate these matters and support them with different and *orthogonal* mechanisms. To use the most prominent example, there are several approaches supporting a (more or less limited) separation of *types* and *implementations* and consequently a decoupling of *subtype polymorphism* and *code reuse*, e. g., interfaces in Java [9], signatures in extended C++ [2], and types and implementations in Timor [14] (where the latter constitutes the most uncompromising approach, because implementations are not treated as types at all). Experience with these and other approaches clearly indicates that separate and orthogonal mechanisms which can be freely combined *as needed* are much more powerful and useful than a prearranged, fixed combination of them.

The present paper goes another step further by decoupling *dynamic binding* from inheritance and subtype relationships and offering it instead as a separate, orthogonal concept called *dynamic routines*. Once again it turns out that this separation leads to programming languages which are not only considerably more flexible and expressive, but in addition support new dimensions of extensibility when compared to traditional object-oriented languages. Furthermore, since dynamic routines subsume other kinds of routines normally found in programming languages, e. g., global and member functions in C++ [20], normal and type-bound procedures in Oberon-2 [18], and static and instance methods in Java [9], their introduction actually leads to a conceptually simpler language, because these other kinds of routines are no longer required.

The paper is structured as follows. Section 2 commences by distinguishing three different dimensions of extensibility of software systems: (i) *vertical* extensions, i. e., extensions of the type hierarchy, (ii) *horizontal* extensions, i. e., extensions of the spectrum of operations available on types, and (iii) *behavioural* extensions, i. e., extensions or even retroactive modifications of the behaviour of operations. Using rather simple examples, it is shown that typical object-oriented programming languages support only the first dimension in a direct and modular way, while design patterns such as the Visitor pattern [7] are not really satisfactory to support the second dimension. Furthermore, the third dimension is not supported at all. Following that, Sec. 3 introduces the general concept of dynamic routines as well as specific adaptations and integrations into the programming languages C++, Oberon-2, and Java. Using the examples of Sec. 2, it is shown that dynamic routines indeed support all three dimensions of extensibility in a simple and flexible manner. Section 4 then describes the basic approach of implementing dynamic routines as precompiler-based extensions of the above languages. Finally, Sec. 5 presents a critical discussion of the concept itself and its relationship to other programming language concepts as well as some concluding remarks.

2. Dimensions of Extensibility

2.1 Example

Figure 1 depicts a typical object-oriented type hierarchy for the representation of arithmetic expressions. Expressions in general, represented by the abstract root type `Expr`, are subdivided into three major categories represented by the abstract subtypes `Atom` (atomic expression), `Unary` (application of a unary operator to a subexpression), and `Binary` (application of a binary operator to two subexpressions). These are in turn specialized into appropriate concrete subtypes, such as `Const` (constant expression), `Neg` (negation, i. e., unary minus), or `Add` (addition, i. e., binary plus). Types containing private data elements, e. g., `Const` containing

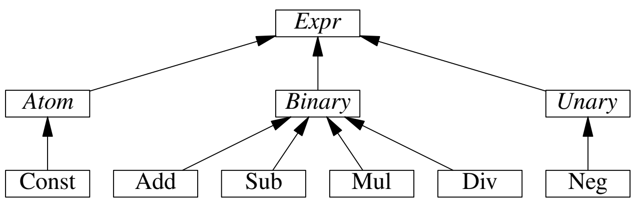

Figure 1: Object-oriented type hierarchy

the numeric value of the constant expression or `Binary` containing pointers to the two subexpressions of the binary expression, provide an appropriate constructor to initialize them and public member functions (methods) to query their value. Furthermore, it is assumed that two basic operations are defined for expressions, determining their value and printing them (e. g., on the standard output stream), which are implemented as virtual member functions `eval` and `print`, respectively. This leads to the C++ class definitions partially shown in Fig. 2.

```
// General expression (abstract).
class Expr { public:
  virtual int eval () = 0;
  virtual void print () = 0;
};

// Atomic expression (abstract).
class Atom : public Expr {};

// Constant expression (concrete).
class Const : public Atom {
  int val_;
public:
  Const (int v) : val_(v) {}
  int val () { return val_; }
  virtual int eval () { return val_; }
  virtual void print () { ...... }
};

// Binary expression (abstract).
class Binary : public Expr {
  Expr* left_;
  Expr* right_;
public:
  Binary (Expr* l, Expr* r)
   : left_(l), right_(r) {}
  Expr* left () { return left_; }
  Expr* right () { return right_; }
};

// Addition (concrete).
class Add : public Binary { public:
  Add (Expr* l, Expr* r) : Binary(l, r) {}
  virtual int eval () {
    return left()->eval() + right()->eval();
  }
  virtual void print () { ...... }
};

......
```

Figure 2: C++ class definitions

The advantages of using virtual member functions, i. e., methods which are *dynamically* bound, instead of normal global functions to implement such operations are obvious and well-known: When new subtypes are added to the type hierarchy later, e. g., to incorporate additional operators such as remainder or other kinds of atomic expressions such as variables, none of the already implemented classes and functions needs to be touched, because the operations for these new types of expressions can be simply implemented as member functions of the corresponding new classes (cf. Fig. 3). If these new classes are defined in a different translation unit, the original classes need not even be recompiled.

```
// Remainder (concrete).
class Mod : public Binary { public:
  Mod (Expr* l, Expr* r) : Binary(l, r) {}
  virtual int eval () {
    return left()->eval() % right()->eval();
  }
  virtual void print () { ...... }
};

// Variable (concrete).
class Var : public Atom {
  string name_;
  int val_;
public:
  Var (string n, int v) : name_(n), val_(v) {}
  string name () { return name_; }
  int val () { return val_; }
  void assign (int v) { val_ = v; }
  virtual int eval () { return val_; }
  virtual void print () { ...... }
};
```

Figure 3: Extending the type hierarchy

2.2 Vertical and Horizontal Extensions

If the “space” of operations implemented by virtual member functions is organized in two dimensions, type and operation, according to Fig. 4, *vertical* extensions, i. e., extensions along the type axis, are always possible without affecting existing code. However, *horizontal* extensions, i. e., later additions of completely new operations such as symbolic differentiation of expressions, require modifications and recompilations of existing classes, because all member functions of a class must at least be declared in the class itself. Besides the fact that this is inconvenient and undesirable from a methodological point of view, because it violates a basic principle of modularity, it might be simply impossible if the class’s source code is unavailable.

		Operation	
		`eval`	`print`
Type	`Expr`	`Expr::eval()`	`Expr::print()`
	`Const`	`Const::eval()`	`Const::print()`
	`Add`	`Add::eval()`	`Add::print()`
	⋮	⋮	⋮

Figure 4: Two-dimensional space of operations

Of course, this problem is not only present in C++, but in many other object-oriented languages, too. Its reason is mainly technical: In a typical implementation of virtual member functions (or methods in other languages), the addresses of all virtual functions of a particular class are stored in the class’s *virtual function table* from which they are accessed via a unique index value. Because a derived class inherits all virtual functions of its base class(es), its virtual function table is constructed as an extension of the base class's table. This is only feasible, however, if the number of virtual functions of the base class is known to the compiler when a derived class is declared. Adding new virtual functions to a class later is therefore impossible.

Of course, it is possible to define a new operation `diff` for symbolic differentiation as a global function according to Fig. 5, but this solution is obviously not vertically extensible, i. e., if new subtypes of `Expr` are introduced later, the functions's body must be extended and recompiled.

```
// Differentiate expression x along variable n.
Expr* diff (Expr* x, string n) {
  if (Const* c = dynamic_cast<Const*>(x)) {
    return new Const(0);
  }
  if (Var* v = dynamic_cast<Var*>(x)) {
    if (v->name() == n) return new Const(1);
    else return new Const(0);
  }
  if (Add* a = dynamic_cast<Add*>(x)) {
    return new Add(diff(a->left(), n),
     diff(a->right(), n));
  }
  ......
}
```

Figure 5: Implementation of `diff` as a global function

It is sometimes claimed, that the set of operations applicable to a type can always be extended by defining the new operations for a derived type. In the present example, it is of course possible to define a new class such as `DiffExpr` with a virtual member function `diff` (cf. Fig. 6). Since `DiffExpr` is derived from `Expr`, the member functions declared in `Expr` are also applicable to instances of `DiffExpr`, *but not vice versa*. So, to be actually useful, it would be necessary to change all existing code using class `Expr` to now use class `DiffExpr` instead. This might be even worse than changing just class `Expr` to directly contain the new member function `diff`.[1]

Furthermore, extending only the root of a whole type hierarchy with a derived type providing a new operation is not sufficient to make the new operation available to the other types of the hierarchy. In fact, in a language supporting single inheritance only, it would be necessary either to extend all other types analogously – losing, however, their essential subtype relationship –, or to build a completely new parallel hierarchy rooted at `DiffExpr` requiring all virtual member functions of the original hierarchy to be reimplemented for the new hierarchy. In languages with multiple inheritance, such as C++, the latter can be avoided in principle by merging these alternatives (cf. Fig. 7), but obviously none of these solutions is really satisfactory. In particular, the last alternative suffers from unintended repeated inheritance of the root class `Expr` and its direct subclasses which could be avoided only by using virtual inheritance even in the original type hierarchy.

[1] By disciplined application of another design pattern, namely some kind of factory pattern, this problem can be circumvented in principle.

```
// Differentiable expression.
class DiffExpr : public Expr { public:
  // Differentiate expression x along variable n.
  virtual DiffExpr* diff (string n) = 0;
};

// Differentiable binary expression.
class DiffBinary
: public DiffExpr, public Binary {
  ......
};

// Differentiable addition.
class DiffAdd
: public DiffBinary, public Add { public:
  ......
  virtual DiffExpr* diff (string n) {
    return new DiffAdd(left()->diff(n),
     right()->diff(n));
  }
};

......
```

Figure 6: Implementation of `diff` as a virtual member function of a derived class

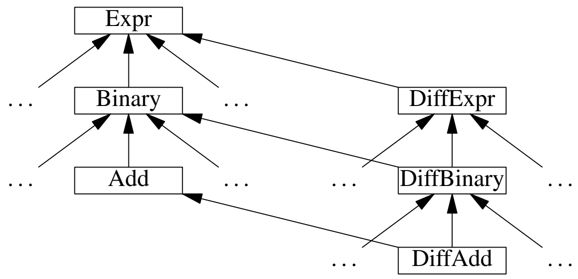

Figure 7: Original and derived type hierarchy

Because the author is not the first who discovered this fundamental limitation of object-oriented languages [4, 3], design patterns such as the *Visitor pattern* [7] have been developed to overcome it in principle. Despite the fact, that such a pattern can be used to solve the problem of horizontal extensions in a pragmatical way if it is strictly employed from the very beginning, the resulting solution is rather unnatural and complicated and thus not really satisfactory from a methodological and aesthetical point of view. Without going into details – and without wanting to discredit the work on design patterns, which often are the only practical means to overcome such limitations of programming languages –, building an extensible software system by employing the Visitor pattern would be similar to building an extensible car by always providing it with, say, a roof rack where all later extensions – such as a hook for a trailer or fog lamps – have to be mounted on because it is impossible to mount them anywhere else (cf. Fig. 8).

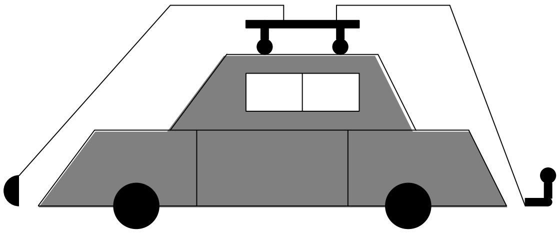

Figure 8: Possible illustration of the Visitor pattern

2.3 Behavioural Extensions and Modifications

But even if a class such as `Expr` and its subclasses have been designed very carefully to avoid any necessity of later horizontal extensions, it might happen that extensions (or even modifications) along a third dimension become necessary, namely extensions (or later modifications) of the implemented behaviour. The following is an arbitrary list of possible requirements that might arise in specific application contexts of this class hierarchy:

1. If the class hierarchy, together with other code using it, shall be employed in a multi-threaded program, calls to the member function `print` must be synchronized to prevent intermixing of multiple outputs. Because modifications and recompilations of this code are undesirable or impossible (if source code is not available), a mechanism for retroactively *augmenting* or "bracketing" the function bodies of `print` with synchronization code would be necessary.
2. Because integer division with one or two negative operands might behave differently on different hardware architectures (always rounding down versus always rounding towards zero), it would be desirable to retroactively *replace* the implementation of the member function `Div::eval` with an architecture-independent implementation which always divides the absolute values of the operands and determines the sign of the result itself. Alternatively, the actual rounding behaviour might depend on a global or environment variable.
3. If division by zero is not handled by the original implementation of `eval` or its replacement mentioned above, or the implemented behaviour (e. g., immediate program termination) is not appropriate for an application, it would be necessary to retroactively *prefix* the implementation of `Div::eval` with a test whether the second operand is zero and an appropriate behaviour for that case.
4. If the new behaviour of `eval` in that case is to return a special *null value* (which might be represented for instance as the smallest available integer value), all other instances of `eval` would have to be retroactively prefixed with a test, too, whether one or both of the operands is equal to that null value, returning again the null value in that case instead of executing the normal evaluation code.

Obviously, none of these requirements can be fulfilled with a normal object-oriented (or imperative) programming language without modifying and recompiling existing code.[2] On the other hand, it is interesting to note that other kinds of languages, such as the Common Lisp Object System (CLOS) or some scripting languages, are able to meet at least some of these goals (cf. Sec. 5.2).

Other situations, not related to the example of arithmetic expressions, where behavioural extensibility would be very helpful in practice, include:

- Retroactively augment the implementation of the C library routine `malloc` (or that of the C++ operator `new`) with code that performs bookkeeping and gathers statistical information about the application's usage of this routine.
- Suffix the same routine with a test whether the result would be a null pointer, causing controlled program termination in that case instead of actually returning the null pointer.
  This could be very useful to retroactively fix program code using `malloc` which has been written by sloppy programmers who have forgotten to always check that exceptional case.
- Redefine the implementation of the C library routine `realloc` to make the case `realloc(p, 0)` equivalent to `free(p)`, if the available implementation does not already do that.
- Augment the Unix system calls `read` and `write` (or whatever functions are ultimately called by all standard I/O routines to actually read and write data) with code that copies all data read from the standard input stream and written to the standard output stream to another output stream, thus creating a log of the application's interactive I/O behaviour.
- Quickly augment some critical routines of a program with diagnostic outputs to trace their execution paths.
- Patch erroneous library routines by retroactively replacing their implementation.

3. Supporting Extensibility with Dynamic Routines

Having explained these fundamental limitations of object-oriented languages, which continuously turn out to hinder truly extensible and modular programming in practice, *dynamic routines* will be introduced in the following as a simple yet powerful general concept to overcome them. Afterwards, specializations of the general concept will be described which have been designed and implemented for particular programming languages.

3.1 General Concept

The basic idea of dynamic routines, which is rather simple, has been inspired by the concept of *partially defined functions* in mathematics. For example, the function *sign* determining the sign of a number x is typically defined by three complementary *branches* defining its behaviour on different subsets of the overall domain:

$$sign(x) = 1, \quad \text{if } x > 0;$$
$$sign(x) = 0, \quad \text{if } x = 0;$$
$$sign(x) = -1, \quad \text{if } x < 0.$$

Similarly, a partially defined function such as

$$f(x) = x \sin(1/x), \quad \text{if } x \neq 0$$

might be complemented by another partial definition

$$f(x) = 0, \quad \text{if } x = 0$$

to extend its domain to all real numbers $x \in \mathbb{R}$. If necessary,

[2] Again it is possible in principle, by strictly applying some kind of factory pattern, to perform behavioural extensions by introducing subclasses that override methods of the original classes. See [12, 13] for a critical discussion of this possibility.

mathematicians do not even hesitate to redefine a previously defined function such as

$$sub(x, y) = x - y \quad \text{(without restrictions on } x, y \in \mathbb{R}\text{)}$$

on a subset of its domain, e. g.,

$$sub(x, y) = 0, \quad \text{if } x < y,$$

to adapt it to a particular context (here, to make sure that *sub* does not return negative values). Without being explicitly expressed in normal mathematical texts, the rule for finding the appropriate definition of a particular function is to search *backwards* from the "point of invocation" (i. e., the place where the function is used) to the *latest* definition of the function encountered so far whose accompanying condition is satisfied by the actual parameter values. By searching backwards from the point of invocation instead of forward from the beginning of the text, the case of redefinition illustrated in the last example is handled as desired. In the other examples, where the conditions of the different branches of a function are logically disjoint (i. e., at most one of them is satisfied by any actual parameter values), the search order is irrelevant.

In a programming language, a *dynamic routine* is a function, procedure, or method (whatever terminology is used in the particular language) with an associated *guard*, i. e., a Boolean-valued expression acting as a *precondition*. In contrast to normal routine definitions, it is allowed to define multiple instances or *branches* of the same dynamic routine (i. e., having the same signature) in the same scope, typically (but not necessarily) with different (and in most cases even logically disjoint) guards.

Dynamic routines are called in the same way as normal routines of the language, i. e., there is no difference from a client's perspective. To find the branch that will be actually executed, the branches' guards will be evaluated in reverse order of definition until the first satisfied condition is found; then the corresponding routine body is executed. (If no "matching" branch is found, a runtime error occurs.) Similar to the way a member function or method can call the method of its base type that it is overriding, a dynamic routine can call the *next applicable branch*, i. e., the next branch in reverse definition order whose condition is satisfied. In contrast to normal routine calls, however, such a call never takes explicit parameters because the original parameter values of the dynamic routine are implicitly passed unchanged, even if the formal parameters have been modified.

3.2 Dynamic Functions in C++

To remain compatible with established terminology [20], dynamic routines are called *dynamic functions* in C++. In analogy to virtual member functions, which can be overridden in derived types, they are also called *global virtual functions*, because they are global functions that can be overridden. They are declared and defined like normal functions prefixed by a new keyword `dynamic`. Alternatively, the existing function specifier `virtual` can be used which is not applicable to global functions in standard C++. Furthermore, the head of a dynamic function usually contains one or more guards of the form `if (cond)` with a Boolean-valued expression `cond`. Figure 9 demonstrates the basic principles by implementing the mathematical examples of the previous subsection as dynamic functions.

The three definitions of `sign` show the most basic form of dynamic functions, where the guard is explicitly coded as an `if` clause in the function head. In contrast to that, the two definitions of `f` — which are deliberately scattered to demonstrate that the branches of a dynamic function need not appear together — illustrate a more compact and convenient way of integrating guards into the parameter declaration list. In general, a *guarded parameter declaration* such as `int x != 0` is equivalent to the normal parameter declaration `int x` plus the guard `if (x != 0)`.

```
dynamic int sign (double x) if (x > 0) {
  return 1;
}
dynamic int sign (double x) if (x == 0) {
  return 0;
}
dynamic int sign (double x) if (x < 0) {
  return -1;
}

dynamic double f (double x != 0) {
  return x * sin(1/x);
}
dynamic double sub (double x, double y) {
  return x - y;
}

dynamic double f (double x == 0) {
  return 0;
}
dynamic double sub (double a, double b)
if (a < b) {
  return 0;
}
```

Figure 9: Mathematical examples of dynamic functions

Finally, the first definition of `sub` demonstrates the way of transforming a normal function without any restrictions on its parameter values into a dynamic function that can be redefined later, by simply prefixing it with the keyword `dynamic`. (Formally, such a definition is accompanied by the tautological guard `if (true)`.) Furthermore, the second definition of `sub` illustrates that parameter *names* might vary from branch to branch, because they are not part of a function's signature.

Of course, in these simple examples, implementing an operation such as `sign` as a dynamic function with multiple branches is unnecessarily cumbersome and more elaborate than implementing it as a single normal function with a corresponding selection statement in its body. But as the subsequent examples will clearly demonstrate, employing dynamic functions is the only way to keep an implementation open for expected and unexpected extensions along all three dimensions described in Sec. 2.

Figure 10 shows a re-implementation of the operations `eval` and `print` (cf. Fig. 2) as sequences of dynamic function definitions, together with a correspondingly modified class hierarchy that does not contain any virtual member functions except for a dummy function that is necessary to make the classes polymorphic (i. e., allow dynamic type tests on their instances using `dynamic_cast`) and to distinguish abstract from concrete classes.

The first branches of `eval` and `print` contain an explicit guard using a `dynamic_cast` to test whether the dynamic type of `Expr* x` is a `Const* c`; if this is true, the value `c->val()` of the constant expression `c` is returned or printed, respectively. The second definitions of these functions demonstrate a more convenient form of dynamic type tests in dynamic function heads using a colon as a pseudo comparison operator resembling Java's `in-`

```
// General expression (abstract).
class Expr {
  // Pure virtual dummy function to make
  // the class both abstract and polymorphic.
  virtual void dummy () = 0;
};

// Atomic expression (abstract).
class Atom : public Expr {};

// Constant expression (concrete).
class Const : public Atom {
  // Formally implement the dummy function
  // to make the class concrete.
  void dummy () {}
  int val_;
public:
  Const (int v) : val_(v) {}
  int val () { return val_; }
};
dynamic int eval (Expr* x)
if (Const* c = dynamic_cast<Const*>(x)) {
  return c->val();
}
dynamic void print (Expr* x)
if (Const* c = dynamic_cast<Const*>(x)) {
  ......
}

// Binary expression (abstract).
class Binary : public Expr {
  Expr* left_;
  Expr* right_;
public:
  Binary (Expr* l, Expr* r)
   : left_(l), right_(r) {}
  Expr* left () { return left_; }
  Expr* right () { return right_; }
};

// Addition (concrete).
class Add : public Binary {
  void dummy () {}
public:
  Add (Expr* l, Expr* r) : Binary(l, r) {}
};
dynamic int eval (Expr* x : Add*) {
  return eval(x->left()) + eval(x->right());
}
dynamic void print (Expr* x : Add*) {
  ......
}

......
```

Figure 10: Re-implementation of `eval` and `print` with dynamic functions

`stanceof` operator. In general, a guarded parameter declaration such as `Expr* x : Add*` is equivalent to the normal parameter declaration `Expr* xx` (with some unique name `xx`) plus the guard `if (Add* x = dynamic_cast<Add*>(xx))`. Thus, the static type of the formal parameter `x` is `Expr*` in the function's signature, but `Add*` in its body.

Figure 11 shows additional definitions (possibly in a different translation unit) extending both the type hierarchy with a new type `Var` including accompanying branches of the dynamic functions `eval` and `print` (vertical extension) and the spectrum of operations with a new dynamic function `diff` (horizontal extension). As this example shows, it is of course possible to implement `diff` in a single branch of a dynamic function for all concrete subtypes of `Expr` known so far using a selection statement with multiple branches, while at the same time remaining open for later vertical extensions.

```
// Variable (concrete).
class Var : public Atom {
  string name_;
  int val_;
public:
  Var (string n, int v) : name_(n), val_(v) {}
  string name () { return name_; }
  int val () { return val_; }
  void assign (int v) { val_ = v; }
};
dynamic int eval (Expr* x : Var*) {
  return x->val();
}
dynamic void print (Expr* x : Var*) {
  ......
}

// Differentiate expression x along variable n.
dynamic Expr* diff (Expr* x, string n) {
  if (Const* c = dynamic_cast<Const*>(x)) {
    return new Const(0);
  }
  if (Var* v = dynamic_cast<Var*>(x)) {
    if (v->name() == n) return new Const(1);
    else return new Const(0);
  }
  if (Add* a = dynamic_cast<Add*>(x)) {
    return new Add(diff(a->left(), n),
      diff(a->right(), n));
  }
  ......
}
```

Figure 11: Vertical and horizontal extensions

Finally, Fig. 12 shows some behavioural extensions and modifications of the functions `print` and `eval` implementing the requirements mentioned in Sec. 2.3.

Extension 1 shows the typical pattern for bracketing an existing implementation of a function with a prelude and a postlude. To avoid the introduction of another new keyword into the language, the next applicable branch of a dynamic function can be called by using the keyword `dynamic` as the name of a parameterless pseudo function.

Extension 2 is one of the rare examples where the original implementation of a function is completely overridden, i. e., the next applicable branch of the dynamic function is not called at all. To

```
// Extension 1:
// Augment print with synchronization code.
dynamic void print (Expr* x) {
  // Mutex allowing multiple recursive locks
  // by the same thread.
  static pthread_mutex_t m
   = PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP;

  pthread_mutex_lock(&m);   // Prelude.
  dynamic();                // Orig. function.
  pthread_mutex_unlock(&m); // Postlude.
}

// Extension 2:
// Redefine division with negative operands.
dynamic int eval (Expr* x : Div*) {
    int l = eval(x->left());
    int r = eval(x->right());
    int sign = (l >= 0) == (r >= 0) ? 1 : -1;
    return abs(l) / abs(r) * sign;
}

// Null value.
const int null = INT_MIN;

// Extension 3:
// Handle division by zero.
dynamic int eval (Expr* x : Div*) {
  if (eval(x->right()) == 0) return null;
  else return dynamic();
}

// Extension 4:
// Handle null valued operands.
dynamic int eval (Expr* x : Unary*)
if (eval(x->body()) == null) {
  return null;
}
dynamic int eval (Expr* x : Binary*)
if (eval(x->left()) == null
|| eval(x->right()) == null) {
    return null;
}
```

Figure 12: Behavioural extensions and modifications

the contrary, extensions 3 and 4 are typical examples where the next applicable branch is called conditionally, either explicitly using `dynamic()` in the body or implicitly if the guard's condition is not satisfied.

Given these definitions, the call to `eval` in the following code will be executed as described below:

```
Const* y = new Const(10);
Const* z = new Const(-4);
Div* x = new Div(y, z);
int v = eval(x);
```

- The last branch of `eval` defined in Fig. 12 is executed with the actual parameter `x`.
  Its implicit guard `Expr* x : Binary*`, which is evaluated first, is satisfied since the dynamic type of (the object pointed to by) `x`, i. e., `Div`, is a subtype of `Binary`. Thus, its explicit guard is evaluated next, causing recursive calls to `eval` with actual parameters `x->left()` and `x->right()`, i. e., `y` and `z`, respectively.
- Because the dynamic type of these objects is `Const`, these calls finally end up with calling the first branch defined in Fig. 10, returning the values `10` and `-4`, respectively.
- Since both these values are different from `null`, the explicit guard of the last branch is not satisfied, causing an implicit call to the previous branch.
  The implicit guard `Expr* x : Unary*` of this branch is not satisfied, too, causing in turn an implicit call to its predecessor.
- Since the implicit guard `Expr* x : Div*` of this branch is satisfied, its body is actually executed.
  Because the condition `eval(x->right()) == 0` of its if-then-else statement is not satisfied, the next applicable branch is explicitly called in the else part.
- This reaches the branch "Extension 2," whose body will be executed since its implicit guard `Expr* x : Div*` is satisfied. After evaluating the operands `x->left()` and `x->right()`, the final result value `-2` will be computed and returned.

Remark: If extensions 2, 3, and 4 are actually implemented in the same translation unit, it would be more efficient to combine them into a single branch of the dynamic function `eval` to avoid repeated recursive calls to evaluate the subexpressions `x->left()` and `x->right()`. On the other hand, Fig. 12 demonstrates that it is indeed possible to implement these extensions and modifications in a very modular and orthogonal way.

3.3 Dynamic Procedures in Oberon-2 and Related Languages

Similar to C++, dynamic routines can be integrated as *dynamic procedures* into Oberon and Oberon-2 [18], where they can be used instead of *type-bound procedures* (Oberon-2's equivalent of dynamically bound methods) or hand-coded implementations of dynamic binding in Oberon (by means of *procedure records*, i. e., records explicitly containing the procedures associated with an object). It is even possible to integrate the concept into purely procedural languages such as Pascal, Modula-2, or C, yielding a language that supports horizontal and behavioural extensions, but – due to the lack of an appropriate type system – no vertical extensions.

To demonstrate the basic principles, figures 13 and 14 show some typical examples of dynamic procedures in Oberon-2. In analogy to C++, a new keyword `DYNAMIC` has been introduced which is used both to distinguish dynamic from normal procedure declarations and to invoke the next applicable branch in the body of a dynamic procedure. In contrast to C++, guards are always integrated into the formal parameter list, either as *guarded formal parameter sections* such as `x : Expr IS Unary`, or as additional Boolean-valued expressions such as `BE.eval(x.body) = null`, since a separate `IF` clause would not fit well into a procedure head. To add branches to a dynamic procedure that is imported from another module, the procedure name must be qualified with the name of the exporting module (or an alias name such as `BE` declared in an `IMPORT` clause). Since such branches cannot be called directly as procedures of their enclosing module, it would not make sense to export them.

For readers which are not familiar with the syntactic details of Oberon-2, it should be noted that `*` and `-` denote *export marks* in-

```
MODULE BasicExpr;
  TYPE
    (* General expression. *)
    Expr* = POINTER TO ExprRec;
    ExprRec* = RECORD END;

    (* Atomic expression. *)
    Atom* = POINTER TO AtomRec;
    AtomRec* = RECORD (ExprRec) END;

    (* Constant expression. *)
    Const* = POINTER TO ConstRec;
    ConstRec* = RECORD (AtomRec)
      val-: INTEGER;
    END;

    (* Binary expression. *)
    Binary* = POINTER TO BinaryRec;
    BinaryRec* = RECORD (ExprRec)
      left-, right-: Expr;
    END;

    (* Addition. *)
    Add* = POINTER TO AddRec;
    AddRec* = RECORD (BinaryRec) END;

    ......

  DYNAMIC PROCEDURE eval*
   (x : Expr IS Const) : INTEGER;
  BEGIN RETURN x.val
  END eval;

  DYNAMIC PROCEDURE eval*
   (x : Expr IS Add) : INTEGER;
  BEGIN RETURN eval(x.left) + eval(x.right)
  END eval;

  ......
END BasicExpr.
```

Figure 13: Dynamic procedures in Oberon-2

dicating full and read-only export, respectively, of an identifier from a module. Furthermore, the base type of an *extended record type* is included in parentheses after the keyword `RECORD`, and the keyword `IS` denotes a dynamic type test.

3.4 Dynamic Class Methods in Java (cf. [12, 13])

In contrast to C++ and Oberon-2, Java [9] (like many other object-oriented languages) does not directly support global functions or procedures which could be used as the basis for dynamic routines. Nevertheless, *class methods* (also called static methods, in contrast to instance methods) provide essentially the same functionality and, when compared to imported procedures in Oberon-2, even follow the same calling syntax: a qualified identifier followed by an argument list. Therefore, it seems obvious and reasonable to map the general concept of dynamic routines to some sort of class methods in Java which are called *dynamic class methods*.

To be concrete, figures 15 and 16 show the same functionality as figures 10 and 12, but implemented with dynamic class methods in Java. (Note that integer division in Java always rounds towards zero, so extension 2 is unnecessary.) As in C++, a new keyword `dynamic` is used to distinguish dynamic class methods from both static class methods and instance methods as well as to invoke the next applicable branch of a dynamic class method. Guards also follow the same syntax as in C++, i. e., they might be specified explicitly by an `if` clause or implicitly by guarded parameter declarations (using the Java keyword `instanceof` for dynamic type tests instead of the newly introduced colon operator in C++). From a client's perspective, dynamic class methods are called just like static class methods, i. e., the method's name is qualified by the class name. Just like other methods, dynamic class methods can be declared `public`, `protected`, or `private` to specify their accessibility as well as `abstract`, `synchronized`, and `strictfp`, but not `static`, `final`, or `native`.

Similar to abstract instance methods, an abstract dynamic method declares the method's signature, but does not provide a real implementation. It is actually equivalent to a branch with the unsatisfied guard `if (false)` and an empty body. Therefore, in contrast to abstract instance methods, an abstract dynamic method may well appear in a concrete class.

If the method modifiers `synchronized` and `strictfp` are applied to a dynamic class method, their meaning is the same as for a static class method. In particular, a synchronized dynamic class method locks resp. unlocks the object corresponding to its *enclosing* class (which might differ from the class where the method has been defined originally) before resp. after executing its body, including the evaluation of its guards.

Similar to the way an imported dynamic procedure can be overridden or extended in another Oberon-2 module, an accessible dynamic class method of another class can be extended by using its qualified name in a method declaration. In particular, public dynamic class methods of a class may be extended by any other

```
MODULE ExtExpr;
  IMPORT BE := BasicExpr;

  CONST null = MIN(INTEGER);

  (* Extension 3: Handle division by zero. *)
  DYNAMIC PROCEDURE BE.eval
   (x : BE.Expr IS BE.Div) : INTEGER;
  BEGIN
    IF BE.eval(x.right) = 0 THEN RETURN null
    ELSE RETURN DYNAMIC()
    END
  END BE.eval;

  (* Extension 4: Handle null valued operands. *)
  DYNAMIC PROCEDURE BE.eval
   (x : BE.Expr IS BE.Unary;
    BE.eval(x.body) = null) : INTEGER;
  BEGIN RETURN null
  END BE.eval;

  DYNAMIC PROCEDURE BE.eval
   (x : BE.Expr IS BE.Binary;
    (BE.eval(x.left) = null) OR
    (BE.eval(x.right) = null)) : INTEGER;
  BEGIN RETURN null
  END BE.eval;
END ExtExpr.
```

Figure 14: Behavioural extensions to module `BasicExpr`

```
// General expression.
abstract class Expr {
  // Evaluate expression x.
  public dynamic abstract int eval (Expr x);

  // Print expression x.
  public dynamic abstract void print (Expr x);
}

// Atomic expression.
abstract class Atom extends Expr {}

// Constant expression.
class Const extends Atom {
  private int val;
  public Const (int v) { val = v; }
  public int val () { return val; }

  // Redefine dynamic class methods of Expr.
  dynamic
  int Expr.eval (Expr x instanceof Const) {
    return x.val;
  }
  dynamic
  void Expr.print (Expr x instanceof Const) {
    ......
  }
}

// Binary expression.
abstract class Binary extends Expr {
  private Expr left, right;
  public Binary (Expr l, Expr r) {
    left = l; right = r;
  }
  public Expr left () { return left; }
  public Expr right () { return right; }
}

// Addition.
class Add extends Binary {
  public Add (Expr l, Expr r) { super(l, r); }

  // Redefine dynamic class methods of Expr.
  dynamic
  int Expr.eval (Expr x instanceof Add) {
    return Expr.eval(x.left())
     + Expr.eval(x.right());
  }
  dynamic
  void Expr.print (Expr x instanceof Add) {
    ......
  }
}

......
```

Figure 15: Implementation of `eval` and `print` with dynamic class methods in Java

```
// Extension 1:
// Augment print with synchronization code.
class SyncExpr {
  dynamic
  synchronized void Expr.print (Expr x) {
    dynamic();
  }
}

// Extensions 3 and 4:
// Handle division by zero
// and null valued operands.
class NullExpr {
  public static final int NULL
   = Integer.MIN_VALUE;

  dynamic
  int Expr.eval (Expr x instanceof Unary)
  if (Expr.eval(x.body()) == NULL) {
    return NULL;
  }

  dynamic
  int Expr.eval (Expr x instanceof Binary) {
    if (Expr.eval(x.left()) == NULL
    || Expr.eval(x.right()) == NULL) {
      return NULL;
    }
    try {
      return dynamic();
    }
    catch (ArithmeticException e) {
      return NULL;
    }
  }
}
```

Figure 16: Behavioural modifications and extensions

class, while protected dynamic class methods may be extended by subclasses and classes belonging to the same package only. Because such an additional branch of a “foreign” dynamic class method cannot be called directly as a method of its enclosing class, its access modifier is actually ignored if present.

3.5 Extensibility of Legacy Code

If the C library functions `malloc` and `realloc` as well as the Unix system calls `read` and `write` (which are actually library functions, too) were dynamic functions, their extensions mentioned at the end of Sec. 2.3 could actually be implemented as shown in Fig. 17. Unfortunately, however, this is normally not the case, thus limiting the applicability of dynamic routines to newly developed code.

However, using some nasty, system-dependent tricks, e. g., specifying appropriate options at link time, it is usually possible to *rename* the routines contained in a library and then *replace* them with new implementations that call the original routines with their new names. If these new implementations are dynamic functions (which can be generated mechanically), it is indeed possible to create a new version of an existing library whose functions are dynamic and thus fully extensible, without needing to edit or recompile the library’s source code.

```
dynamic void* malloc (size_t n) {
  // Gather statistical information.
  ......

  // Check result of original function
  // and abort when receiving a null pointer.
  if (void* p = dynamic()) return p;
  else abort();
}

dynamic void* realloc (void* p, size_t n)
if (n == 0) {
  free(p);
}

int log = ...; // File descriptor of log file.

dynamic ssize_t read
 (int fd, void *buf, size_t count)
 if (fd == 0) { // File descriptor 0 is stdin.
  ssize_t n = dynamic();
  if (n > 0) write(log, buf, n);
  return n;
}

dynamic ssize_t write
 (int fd, const void *buf, size_t count)
 if (fd == 1) { // File descriptor 1 is stdout.
  write(log, buf, count);
  return dynamic();
}
```

Figure 17: Retroactive extensions of C library functions

4. Implementation of Dynamic Routines

4.1 Basic Principle

In order to gain immediate practical experience with the concept of dynamic routines in different programming languages, it has been implemented as precompiler-based language extensions for C++, Oberon-2, and Java.

Despite their syntactic differences, the basic principle of transforming dynamic routines to normal routines of the corresponding language plus some auxiliary data structures, is the same for all languages: Every branch of a dynamic routine is transformed to a normal routine having the same parameter list and some unique name. Its body remains essentially unchanged, except that invocations of the next applicable branch, i. e., invocations of the pseudo-routine `dynamic` resp. `DYNAMIC`, are transformed to invocations of (the unique routine corresponding to) the previous branch of the same dynamic routine. In order to be able to pass the original parameter values even if the formal parameters have been modified, backup copies of their values are created at the very beginning of the routine's body. Furthermore, the branch's guard (if present) is moved into the body as a regular if-then-else statement whose then part is the original routine body and whose else part is an invocation of the previous branch. If a guarded parameter declaration contains a dynamic type test (using one of the operators colon, `instanceof`, or `IS`, respectively), an appropriate declaration or statement that statically converts the parameter to its dynamic type is added after (or instead of) the test.

In addition to these normal "branch routines," two additional routines are generated when the first branch is encountered. First, an additional branch routine called "branch zero" is generated which constitutes the previous branch of the first normal branch. If this branch gets called at runtime, it signals an error condition by throwing an (unchecked) exception (in C++ and Java) or directly terminating the program (in Oberon-2, which does not support exception handling). Second, a single "dispatch routine" is generated having the same parameter list and name as the dynamic routine and a body which simply calls its *last* branch. However, since the last branch of a dynamic routine is not known at this place – even if the precompiler would read ahead the whole translation unit – because it might be defined in a different translation unit or even in a library, module, or class that is dynamically loaded at run time, the dispatch routine actually calls it indirectly via a routine (pointer) variable.

4.2 Oberon-2 and Java Details

In Oberon-2 and Java, this variable is declared in the module resp. class containing the very first branch and initialized to the last branch defined in that unit. (Note that in these languages this unit is distinguished by the fact that the routine's name is used unqualified there.) If additional branches are defined in other modules resp. classes (where the routine's name is used qualified), the current value of this variable plays the role of the previous branch for the first branch defined there. Therefore, during initialization of that unit, the variable's value is copied to a local variable of the unit before it is updated to the last branch defined in that unit, and so on. Because in Oberon-2 and Java, the initialization order of the modules resp. classes of a program is well-defined – Oberon-2 modules are initialized in an order determined by their import relationships, while Java classes are (recursively, if necessary) initialized when they are used for the first time –, the overall order of the branches of a dynamic routine in a program is well-defined, too.

4.3 C++ Details

To the contrary, all translation units of a C++ program containing branches of a dynamic function are peers: There is no distinguishing property of the unit containing the "very first" branch, nor is the initialization order of the translation units of a program specified by the language. As a first consequence, this raises the question in which of the translation units the function pointer variable and the branch zero and dispatch functions associated with a dynamic function shall be *defined*, i. e., actually allocated. Fortunately, C++ provides a legal way to evade this question by defining the variable as a static data member and the functions as inlined friend functions of an appropriate template class in *every* translation unit containing branches of the dynamic function. Secondly, the overall order of the branches of a dynamic function actually depends on the order in which the translation units of a program are linked together, because this order (or the reverse of it, depending on the compiler) determines the initialization order of the units. To simulate the Oberon-2 rule that imported modules are initialized in order before the importing module, a simple auxiliary program can be used that interprets `#include` directives in C++ translation units as import relationships and constructs an appropriate ordering of the object files for a particular compiler. Alternatively, if this pragmatical solution is deemed inappropriate, the precompiler could generate an explicit C++ function containing initialization code for every translation unit instead of generating global variable definitions with initializer expressions. By introducing a new keyword such as `import` as another small language extension, the programmer would then be able to explicitly specify the desired invocation order of these initialization functions in analogy to `IMPORT` declarations in Oberon-2.

4.4 Additional Problems

The possibility of overloading function resp. method names in C++ and Java and the consequent possibility of overloading dynamic functions resp. methods, introduces some additional difficulties for the precompiler. For example, the name of the variable associated with a dynamic routine cannot be simply derived from the routine's name. Furthermore, Java does not directly support the analogue of C++ function pointers, while in C++ it is very hard in general to identify branches belonging to the same dynamic function, because the type names appearing in the signatures might be "disguised" by `typedef` and `using` declarations, namespace aliases, template instantiations, etc. Nevertheless, these problems can be solved using some tricks whose description is beyond the scope of this paper and actually not relevant for a user of the concept. (But see [12, 13] for a detailed description of the Java solution.)

To give the reader an impression of the precompilers' work, figures 18 and 19 show the (retroactively beautified and commented) result of applying the Oberon-2 precompiler to the modules `BasicExpr` and `ExtExpr` shown in figures 13 and 14.

4.5 Possible Optimizations

The precompiler-based implementation described so far is rather easy to implement – especially if a grammar of the base language is available which can be directly fed into a parser generator such as Yacc or JavaCC –, and the generated code is fairly efficient in practice. However, by carefully improving the precompilers or by integrating their work into a real compiler of the base language, several optimizations are possible which shall be briefly sketched:

- Instead of mapping each branch of a dynamic routine to a separate normal routine of the base language, all branches defined in the same translation unit might be combined into a single routine to avoid unnecessary routine call overhead for calls of the next applicable branch and to allow standard optimizations, such as common subexpression elimination, for the evaluation of their guards.
- The creation of parameter backup copies can be avoided if the formal parameters of a routine are not modified, which is frequently true.
- If the formal parameters are not modified and the next applicable branch is called at the end of a routine, which is in particular true for implicit calls due to unsatisfied guards, the parameters need not be duplicated on the runtime stack if the next applicable branch is executed in the same stack frame as the current branch. This is similar to optimizing tail-recursive calls in functional programming languages.
- Since guards often contain dynamic type tests on a single routine argument, actually resembling the standard dynamic dispatch strategy of object-oriented languages, similar techniques based on virtual function tables might be employed to reach an applicable branch more directly than by following the normal chain of branches.

None of these optimizations has been actually implemented so far, because most of them can only be done in a real compiler. Even the first one, combination of multiple branches into a single routine, i. e., actually rearrangement of source code, is rather difficult for a precompiler because the exact meaning of some piece of code (including the fact whether it is syntactically and semantically correct) might depend on its *exact* position in a translation unit.

```
MODULE BasicExpr;
  ......

  (* Procedure type and variable *)
  (* containing last branch. *)
  TYPE evalXXXtype*
   = PROCEDURE (x : Expr) : INTEGER;
  VAR evalXXXvar* : evalXXXtype;

  (* Dispatch procedure. *)
  PROCEDURE eval* (x : Expr) : INTEGER;
  BEGIN RETURN evalXXXvar(x)
  END eval;

  (* Branch zero. *)
  PROCEDURE evalXXX0 (x : Expr) : INTEGER;
  BEGIN HALT(1)
  END evalXXX0;

  (* First branch. *)
  PROCEDURE evalXXX1 (x : Expr) : INTEGER;
    (* Parameter backup copy. *)
    VAR xXXX : Expr;

    (* Local procedure implementing *)
    (* keyword DYNAMIC. *)
    PROCEDURE DYNAMIC () : INTEGER;
    BEGIN
      (* Call previous branch. *)
      RETURN evalXXX0(xXXX)
    END DYNAMIC;
  BEGIN
    (* Init. parameter backup copy. *)
    xXXX := x;

    (* Type guard generated from *)
    (* guarded parameter declaration. *)
    WITH x : Const DO
      (* Original procedure body. *)
      RETURN x.val
    ELSE RETURN DYNAMIC() END
  END evalXXX1;


  (* Second branch. *)
  PROCEDURE evalXXX2 (x : Expr) : INTEGER;
    ......
  END evalXXX2;

  ......
BEGIN
  (* Initialize procedure variable. *)
  evalXXXvar := evalXXX2;
END BasicExpr.
```

Figure 18: Transformation of Oberon-2 module `BasicExpr`

```
MODULE ExtExpr;
  IMPORT BE := BasicExpr;

  ......

  (* Local procedure variable. *)
  VAR evalXXX0 : BE.evalXXXtype;

  (* Extension 3: Handle division by zero. *)
  PROCEDURE evalXXX1 (x : BE.Expr) : INTEGER;
    VAR xXXX : BE.Expr;
    PROCEDURE DYNAMIC () : INTEGER;
    BEGIN RETURN evalXXX0(x)
    END DYNAMIC;
  BEGIN
    xXXX := x;
    WITH x : BE.Div DO
      (* Original procedure body. *)
      IF BE.eval(x.right) = 0 THEN RETURN null
      ELSE RETURN DYNAMIC()
      END
    ELSE RETURN DYNAMIC() END
  END evalXXX1;

  (* Extension 4: Handle null valued operands. *)
  PROCEDURE evalXXX2 (x : BE.Expr) : INTEGER;
    VAR xXXX : BE.Expr;
    PROCEDURE DYNAMIC () : INTEGER;
    BEGIN RETURN evalXXX1(x)
    END DYNAMIC;
  BEGIN
    (* Type guard and conditional statement *)
    (* generated from guarded parameter list. *)
    WITH x : BE.Unary DO
      IF BE.eval(x.body) = null THEN
        (* Original procedure body. *)
        RETURN null
      ELSE RETURN DYNAMIC() END
    ELSE RETURN DYNAMIC() END
  END evalXXX2;

  PROCEDURE evalXXX3 (x : BE.Expr) : INTEGER;
    ......
  END evalXXX3;
BEGIN
  (* Store current value in local variable *)
  (* and update imported procedure variable. *)
  evalXXX0 := BE.evalXXXvar;
  BE.evalXXXvar := evalXXX3;
END ExtExpr.
```

Figure 19: Transformation of Oberon-2 module `ExtExpr`

For example, the two calls to the function `f` in the following C++ program have quite different meaning:

```
void f (double x) { ...... }
void g1 () { f(1); }
void f (int x) { ...... }
void g2 () { f(1); }
```

While the call contained in `g1` calls the first definition of `f` after converting the argument of type `int` to `double`, because this is the only matching definition known at this place, the call contained in `g2` calls the second definition, because it matches exactly. Thus, it would be impossible for a precompiler (that does not perform a complete semantic evaluation of its input) to combine `g1` and `g2` into a single function that simply calls `f(1)` twice.

5. Discussion

5.1 Critical Review

Dynamic routines are at the same time a powerful and a dangerous device. When used properly, they offer unique possibilities to extend and retroactively modify software systems, as has been illustrated in this paper. On the other hand, when used inappropriately, they make it quite easy to cause havoc by overriding routines in a completely nonsensical way. However, this dichotomy is typical for every effective tool, not only in programming languages (think, e. g., of pointers, global variables, inheritance, function and operator overloading, etc.), but also in real life (for tools such as knives, axes, or razor blades), and it would not make much sense to completely abandon a useful tool just because it *might* be abused. However, as with real-world tools, a certain amount of practical experience as well as some basic rules and guidelines such as the following might be helpful to avoid unintended misuse:

- To extend a routine to a new domain without altering its behaviour on the original domain, use a guard that is logically disjoint from the guards of all previous branches.
  A type test for a newly introduced subtype is a typical example of such a guard.
- To add orthogonal or "cross-cutting" behaviour, again without altering the original behaviour, define an unguarded branch that follows the prelude/postlude pattern with an embedded unconditional call of the previous branch.
- Make as few assumptions as possible about the behaviour of previous branches, i. e., treat them as a black box whenever possible. Just *add* the code you need for your purpose.
- Think twice before defining a "dead end" branch, i. e., a branch that does not call the previous branch. Consider calling the previous branch and ignoring its result to make sure that orthogonal extensions defined by other branches will be executed, instead of not calling it at all.
- Clearly document branches that violate any of the above rules if you really need them.

If well-meant guidelines and rules are not sufficient to prevent abuse, one might think about enacting strict laws which are enforced by some appropriate "authority." For instance, it might be reasonable in some circumstances to restrict the right of overriding or extending a dynamic class method to subclasses or classes belonging to the same package as the class where the method has been defined originally. Such laws could be enforced by the precompiler and/or the underlying compiler, if additional language constructs were available for specifying such restrictions. Currently, the same aim can be achieved indirectly by assigning the dynamic class method `protected` or default access and defining an additional public static method which simply calls the dynamic method. To achieve more flexible and fine-grained access control, which, however, cannot be enforced at compile time, but only at run time, it would be an interesting task to integrate the concept of dynamic class methods with the Java Security Framework [8].

5.2 Related Work

Ideas to support aims similar to those of dynamic routines can be found in many different areas. For instance, the concepts of open classes, multimethods, and before- and after-methods, found in different combinations, e. g., in MultiJava [3], CLOS [24], and Dylan [5], offer many of the possibilities of dynamic routines. The latter, however, provide additional flexibility by allowing dispatch strategies that are based on arbitrary properties of their arguments, not just their dynamic types. Furthermore, even properties of the "environment," such as values of global or class variables, user preferences read from an application's configuration file, etc., can be incorporated into the dispatch process if appropriate. Finally, complete redefinitions of, e. g., erroneous or incomplete library routines are possible, if the concept is applied consistently, i. e., if routines are always defined dynamically. (This idea is currently investigated in a new programming language where dynamic routines are the sole kind of routines available, thus eliminating the need for different kinds of routines such as global and member functions in C++, normal and type-bound procedures in Oberon-2, and class and instance methods in Java.)

Several scripting languages, including the Unix shell [1] and many others, allow complete redefinitions of routines at runtime. Furthermore, the archaic Unix typesetting system *troff* provides facilities to append additional code to an existing routine and to rename a routine before redefining it, thus allowing the new definition to call the original one in the same way dynamic routines can call their previous branch. By using these facilities carefully, it is possible with these rather "primitive" languages to achieve a degree of extensibility and flexibility that is not provided by most "advanced" programming languages.

Database triggers [22] are a completely different realization of basically the same idea by allowing the retroactive specification of one or more pieces of code that shall be executed instead of or in addition to data manipulation statements such as `insert`, `update`, and `delete` when a particular condition (corresponding to a guard) is satisfied.

Aspect-oriented programming, in particular the languages AspectJ [16] and AspectC++ [19], provide powerful concepts to achieve extensibility of software systems along all three dimensions by weaving the extensions defined by aspects into the source or byte code of the original system when compiling it as a whole. Dynamic routines, on the other hand, do not change at all the code of the system that shall be extended, while offering comparable flexibility. Furthermore, in contrast to aspect-oriented languages, the concept requires only marginal language extensions; to the contrary, as already mentioned above, when dynamic routines are introduced into a language, all other kinds of routines might be thrown out, actually yielding a simpler language.

Finally, the new programming language Timor, in whose development the author is involved, provides concepts called *qualifying types* and *bracket routines* [15] which are quite similar to dynamic routines. An essential difference, however, is the fact that the extensions or modifications implemented by bracket routines are applied only if an object is explicitly associated with an instance of a qualifying type, while the extensions or modifications implemented by additional branches of a dynamic routine are applied automatically. The latter is especially helpful to cope with unexpected behavioural extensions or modifications. On the other hand, Timor provides so-called *generalized bracket routines* which can be applied to *all* methods – separated into *enquiries* which cannot modify the state of their target object and *operations* which can do so – of *any* type. This is very useful for developing general-purpose software components, e. g., for synchronization, monitoring, and protection. When using dynamic routines for such purposes, it is necessary to augment each affected routine separately.

5.3 Conclusion

Dynamic routines have been suggested as a simple yet powerful concept to achieve horizontal and behavioural extensibility of software systems in a very general and flexible way. Concrete specializations of the general concept have been designed and implemented as precompiler-based language extensions for C++, Oberon-2, and Java in order to gain immediate practical experience with the concept in different programming languages. Together with a complementary concept called *open types*, which provides horizontally extensible data structures, the concept of dynamic routines has been successfully employed so far to implement an interaction manager synchronizing concurrent workflows [11] and a graphical editor for interaction graphs [10]. Additional projects are planned, especially to compare source code implementing the same functionality with and without dynamic routines, respectively, regarding its length, readability, extensibility, and run time efficiency.

Acknowledgement

Many thanks are due to Arne Vogel for doing an excellent job in implementing the precompiler for C++.

References

[1] L. J. Arthur: *UNIX Shell Programming* (Second Edition). John Wiley & Sons, New York, 1990.

[2] G. Baumgartner, V. F. Russo: "Signatures: A Language Extension for Improving Type Abstraction and Subtype Polymorphism in C++." *Software—Practice and Experience* 25 (8) August 1995, 863–889.

[3] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.

[4] W. Cook: "Object-Oriented Programming versus Abstract Data Types." In: J. W. de Bakker (ed.): *Foundations of Object-Oriented Language* (REX School/Workshop; Noordwijkerhout, The Netherlands, May/June 1990; Proceedings). Lecture Notes in Computer Science 489, Springer-Verlag, Berlin, 1991, 151–178.

[5] I. D. Craig: *Programming in Dylan*. Springer-Verlag, London, 1997.

[6] M. Evered, J. L. Keedy, A. Schmolitzky, G. Menger: "How Well Do Inheritance Mechanisms Support Inheritance Concepts?" In: H. Mössenböck (ed.): *Modular Programming Languages* (Joint Modular Languages Conference, JMLC'97; Linz, Austria, March 1997; Proceedings). Lecture Notes in Computer Science 1204, Springer-Verlag, Berlin, 1997, 252–266.

[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[8] L. Gong: *Inside Java 2 Platform Security*. Addison-Wesley, Reading, MA, 1999.

[9] J. Gosling, B. Joy, G. Steele: *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[10] C. Heinlein: "Workflow and Process Synchronization with Interaction Expressions and Graphs." In: *Proc. 17th Int. Conf. on Data Engineering (ICDE)* (Heidelberg, Germany, April 2001). IEEE Computer Society, 2001, 243−252.

[11] C. Heinlein: "Synchronization of Concurrent Workflows Using Interaction Expressions and Coordination Protocols." In: R. Meersman, Z. Tari (eds.): *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE* (Confederated International Conferences; CoopIS, DOA, and ODBASE 2002; Proceedings). Lecture Notes in Computer Science 2519, Springer-Verlag, Berlin, 2002, 54−71.

[12] C. Heinlein: "Dynamic Class Methods in Java." In: D. Rombach (ed.): *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8.

[13] C. Heinlein: *Dynamic Class Methods in Java*. Nr. 2003-05, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003. http://www.informatik.uni-ulm.de/pw/berichte

[14] J. L. Keedy, G. Menger, C. Heinlein: "Support for Subtyping and Code Re-use in Timor." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 35−43.

[15] J. L. Keedy, G. Menger, C. Heinlein, F. Henskens: "Qualifying Types Illustrated by Synchronization Examples." In: M. Aksit, M. Mezini, R. Unland (eds.): *Objects, Components, Architectures, Services, and Applications for a Networked World* (Int. Conf. NetObjectDays, NODe 2002; Erfurt, Germany, October 2002; Revised Papers). Lecture Notes in Computer Science 2591, Springer-Verlag, Berlin, 2003, 330−344.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 − Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327−353.

[17] W. LaLonde, J. Pugh: "Subclassing ≠ Subtyping ≠ Is-a." *Journal of Object-Oriented Programming* 3/91, 1991, 57−62.

[18] H. Mössenböck, N. Wirth: "The Programming Language Oberon-2." *Structured Programming* 12 (4) 1991, 179−195.

[19] O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53−60.

[20] B. Stroustrup: *The C++ Programming Language* (Special Edition). Addison-Wesley, Reading, MA, 2000.

[21] C. Szyperski: "Import is Not Inheritance. Why We Need Both: Modules and Classes." In: O. Lehrmann Madsen (ed.): *ECOOP'92* (European Conference on Object-Oriented Programming; Utrecht, The Netherlands, June/July 1992; Proceedings). Lecture Notes in Computer Science 615, Springer-Verlag, Berlin, 1992.

[22] J. Widom, S. Ceri (eds.): *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann Publishers, 1996.

[23] R. Wilhelm, D. Maurer: *Compiler Design*. Addison-Wesley, Wokingham, England, 1995.

[24] P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.