Proceedings of the MSPLS Spring '95 Workshop

Gerald Baumgartner

Vincent R. Russo

Report Number:
95-057
Proceedings of the MSPLS Spring '95 Workshop

Saturday, 8 April 1995, located at Purdue University, West Lafayette, IN, U.S.A.

Technical Report CSD-TR-95-057
URL: http://www.cs.purdue.edu/MSPLS/Spring95/

Gerald Baumgartner and Vincent F. Russo (eds.)
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

gb@cs.purdue.edu and russo@cs.purdue.edu

7 September 1995

Foreword

Historically, the aim of MSPLS Workshops has been to allow an informal exchange of ideas in all areas of programming languages and systems among researchers and practitioners from midwestern universities and companies. Contributions span the range from presentations on preliminary research results to introductory lectures on ongoing research efforts.

This report is a collection of abstracts, extended abstracts, and/or papers of the talks given at the Spring '95 Workshop of the Midwest Society for Programming Languages and Systems, which took place at Purdue University on Saturday, 8 April 1995.

Program

Attendees

The list of attendees is only available to workshop attendees.
MSPLS Spring '95 Workshop
Program
Saturday, 8 April 1995

10:00-11:00 Registration and Breakfast

11:00-12:00

Language Constructs for High-Precision Specification of Real Time Constraints
Tai M. Chung and Henry G. Dietz (Dept. of Electrical and Computer Engineering, Purdue University)
Abstract, Paper

RTsynchronizer, a High-Level Language Construct for Specifying Real-Time Constraints in Distributed Concurrent Systems
Shangping Ren (Dept. of Computer Science, University of Illinois at Urbana-Champaign)
Abstract

12:00-1:30 Lunch

1:30-3:00

Business Meeting
Election of Henry G. Dietz as new MSPLS president.
Selection of Loyola University, Chicago, as the meeting site for the Fall '95 Workshop.

State in Programming Languages - Issues in Language Design
Uday S. Reddy (Dept. of Computer Science, University of Illinois at Urbana-Champaign)
Abstract
An Algebraic Semantics of Subobjects
Jonathan G. Rossie, Jr. and Daniel P. Friedman (Dept. of Computer Science, Indiana University)
Abstract, Extended Abstract

3:00-4:00 Break and Demos

4:00-5:30

Object Interactions as First Class Objects: From Design to Implementation
Mahesh Dodani, Kok Siew Gan, and Lizette Velazquez (Dept. of Computer Science, The University of Iowa)
Extended Abstract, Paper

A Framework for Higher-Order Functions in C++
Konstantin Läufer (Dept. of Mathematical and Computer Sciences, Loyola University of Chicago)
Abstract, Paper

Lazy Functional Programming for Full-Text Information Retrieval
Donald A. Ziff (Dept. of Computer Science, University of Chicago)
Abstract

6:00-9:00 Dinner
Language Constructs for High-Precision Specification of Real-Time Constraints *

Tai M. Chung and Hank G. Dietz

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907
tai@ecn.purdue.edu, hankd@ecn.purdue.edu

Abstract

In practice, the hard real-time systems are still implemented in low-level programming languages for time critical portions and manually tuned to meet all the timing requirements. Without a real-time language that supports an appropriate way of specifying time constraints for a generic hard real-time systems and the fine-grain timing analysis that is transparent to users, the users will ever suffer from the complex coding and accurate analysis, particularly for the systems requiring fast turn-around responses.

In this paper, we propose novel language constructs that can be added to any imperative programming language so that the extended language provides users a way to specify relative time constraints between arbitrary operations at instruction-level. The compilation techniques unique to transformation of proposed language are also presented as a part of the CHaRTS, Compiler for Hard Real-Time Systems, that generates a valid instruction sequence for a target execution model.

Key words: hard real-time, language, time constraint, CHaRTS

*This work is supported by the Office of Naval Research (ONR) under grant number N00014-91-J-4013.
1 Introduction

In our view of real-time systems, a real-time system consists of a controlling subsystem and controlled entities. A controlling subsystem is a set of computer systems, while the controlled entities can be any of a broad range of systems with mechanical behavior, any device from a simple blender to a complex robot [SR91] [BHJ+82]. Typically, a controlling subsystem executes control programs to receive input from the environment and/or to send commands to the controlled entities appropriately.

For a real-time system to function correctly, the control program must be logically correct and the controlling subsystem must execute the program without timing faults. Either a failure to perform an action at the appropriate time or a flaw in the control program's logic can yield catastrophic consequences in hard real-time systems. Thus, meeting the time constraints is extremely important in such systems.

Based on when the control program is scheduled, real-time systems can be divided into two categories: dynamic and static. Execution order of control tasks in a dynamic system is determined on-the-fly at run-time by a scheduler that examines the current status of the system; often, this scheduler is part of the operating system. Even though the dynamic systems are flexible, they suffer from scheduling overhead and unpredictable risks.

In contrast, a static system is scheduled at compile time based on analysis of the program, time constraints, and resource use predictions [Loc92]. A static system schedules the execution order at compile time based on the predicted behavior of the controlled entities and the timing properties of the controlling subsystems. Thus, the static system guarantees that properly scheduled code will function without a timing failure. However, despite the fact that guaranteed timely execution makes the static systems much more attractive than dynamic systems in a hard real-time environment, static systems have received less attention than dynamic systems. This is due to:

1. Unpredictable machine behavior: general-purpose processors often implement instructions with execution time dependent on operand values, pipeline conflicts, or memory hierarchy use (e.g., cache misses, dynamic RAM refresh cycles, virtual memory page faults). Any of these variables degrades the accuracy of the required timing analysis.
2. Scheduling complexity: both the instruction-level timing analysis and the code reorganization are NP-complete problems [GJ79]. Thus, static scheduling has focused on coarse-grain tasks to reduce the problem size.

3. Lack of programming support: no programming language supports a mechanism to express fully general real-time constraints.

The first problem is not easily solved, but can be avoided by careful design of the computer hardware. For example, the TMS320C40 high-performance DSP has completely static timing properties, provided that interrupts are disabled and local memory is implemented by SRAM [Inc88]. Even interactions between multiple processors can be made to have static timing properties. For example, PAPERS (Purdue’s Adapter for Parallel Execution and Rapid Synchronization) can provide fine-grain communication and synchronization with precise static timing properties [DMSM94] [DCMM95].

Although further work is needed, we believe that the techniques presented in [CD95] form the basic foundation for an appropriate static scheduling algorithm. Thus, the second problem is partially solved and a practical solution is likely to be found soon.

This leaves the problem of providing an appropriate programming model and language. Part of this problem is that specifying fine-grain time constraints in the context of a high-level programming language seems paradoxical. Assembly language hand coding to resolve fine-grain time constraints is not the answer; such programs are difficult to write and maintain, and automated scheduling is almost impossible [Inc86].

It is equally futile to directly use existing high-level real-time programming languages, such as ADA [Bre81], Edison [Han80], or programming languages extended from the general purpose languages like RTL-Euclid [KS86], FLEX [LN88], MPL [NTA90] and TCEL [GB93], because they do not provide a mechanism for specifying time constraints on any language construct finer in granularity than an entire task. These languages also suffer from use of a programming model that represents time constraints as relationships between task pairs that are lexicographically adjacent in the source program; this adjacency constraint is both artificial and insufficiently expressive. Timing relationships between arbitrary (potentially parallel) operations cannot be expressed because some such relationships cannot be mapped into lexicographically adjacent positions in the program.
Section 2 of this paper proposes novel language constructs that can be added to any imperative programming language so that the extended language provides users with a way to specify relative time constraints between arbitrary actions at the level of individual instructions. To aid the reader in understanding how these time constraints are extracted and processed, Section 3 provides a brief summary of the relevant compiler technology. Section 4 summarizes the contributions of this paper and suggests future directions for research.

2 Real-time language

To motivate the design of the new language constructs, consider the informally annotated control program fragment shown in Figure 1. Although this example is remarkably simple and embodies time constraints that could easily have been found in a real application, none of the existing programming languages is capable of expressing this in a form that would facilitate static timing analysis and scheduling.

\[ i_1 : \text{Load r1, sense1} ; \text{Get value from memory-mapped sensor 1} \]
\[ i_2 : \text{Store act1, r1} ; \text{Send value to actuator 1} \]
\[ i_3 : \text{Load r2, sense2} ; \text{Get value from memory-mapped sensor 2} \]
\[ i_4 : \text{Add r3, r1, r2} ; \text{Add the sensor inputs} \]
\[ i_5 : \text{Store act2, r3} ; \text{Send result to actuator 2} \]

Figure 1: An example of code segment for real-time control actions

2.1 Programming model

The real-time language we propose in this paper implements a graphical model, denoted as \( G = (\Gamma, \Theta) \) where the directed graph \( G \) consists of a set of nodes \( \Gamma \) and a set of edges \( \Theta \). The nodes and edges are associated with instructions and real-time constraints respectively. The graphical model, called a directed timed graph or DTG, is distinguished from conventional directed graphs in several aspects. Most importantly, each edge in a DTG does not necessarily indicate a precedence
or dependence relation, but represents the direction of the temporal relation between two control action.

An instruction in P can be classified as an externally viewed instruction (EVI) or an internally viewed instruction (IVI) based on the effect of the instruction. The effect of IVI is limited to the internal computation, while an EVI changes the status of the control environment. For example, the variables defined as volatile in the C language [X3.89] or commands to control robots using RCCL [LHILL88] are EVIs, and their execution must meet the timing constraints specified. Because EVIs may depend on values computed by IVIs, any such dependence also implies a relative time constraint (execution order) which must be preserved.

A directed edge in E is associated with multiple attributes: source object \( s \), sink object \( e \), relational operator \( \eta \), and offset \( \delta \). A constraint \( \theta_k \) is defined as \( \theta_k = < s, e, \eta, \delta > \). The type of relationship represented by each edge is specified by \( \eta \): before(\(<\)), after(\(>\)), concurrent(\(=\)), or exclusive(\(\neq\)). Indeed, any temporal relations between \( s \) and \( e \) for each edge type can be expressed with time value \( \delta \) as:

- before constraint: \( e < s + \delta \) (\( e \) must happen at latest \( \delta \) after \( s \))
- after constraint: \( e > s + \delta \) (\( e \) must happen at earliest \( \delta \) after \( s \))
- concurrent constraint: \( e = s + \delta \) (\( e \) must happen exactly at \( \delta \) after \( s \))
- exclusive constraint: \( e \neq s + \delta \) (\( e \) must NOT happen at \( \delta \) after \( s \))

An edge representing the direction of the temporal relation between the source instruction \( s \) and sink instruction \( e \) follows a simple rule. The \( s \) and \( e \) appear, respectively, in the RHS and the LHS of the constraint.

The \( s \) and \( e \) can be any arbitrary pair of instructions. Moreover, this method is sufficient to specify the constraints among \( n \) instructions using less than \( n(n-1)/2 \) constraints in this model. That is, any constraint among multiple instructions can be specified by our constraint model—even constraints which cannot be satisfied by a sequential schedule (i.e., constraints which imply parallel execution).

It is significant that simple ordering (precedence) constraints can be represented without introducing any additional types. Each precedence constraint of the form "\( x \) uses \( y \)'s result" can be
encoded as the after constraint "x happens at earliest 0 after y". Hence, both timing constraints and precedence constraints can be expressed using the same syntax.

2.2 Desired features in Real-time languages

The single most important aspect of any programming system is how easily that system can be used to create and maintain working programs. Although the relative ease of programming of various languages is largely a matter of "religious" debate, it is clear that familiar language constructs and programming tools are more effective than if a completely new program development style must be adopted. The real-time language should look and "feel" like a popular conventional language, e.g., C. Likewise, because there is no familiar syntax that can be borrowed for expressing time constraints, it is critical that the timing aspects of the code be directly observable and easy to modify. Any timing constraint involving two operations should be able to be expressed directly, without having to adjust other portions of the code.

Part of the ability to treat each constraint independently depends on the ability to combine multiple constraints that affect the same operation. Although most work on real-time systems suggests that such constraints are always combined by requiring all constraints to be met, it is sometimes necessary [GL91] to combine constraints by requiring that any one or more of these constraints be met. This concept of an oring constraint is quite new in the real-time community. Suppose an event A_1 can be execute either before A_2 or after A_3 as shown in Figure 2. This can be mathematically expressed as A_1 < A_2 or A_1 > A_3. The figure 2 shows that the possible range of A_1 could be different for the same constraint based on the temporal values of A_2 and A_3. Despite the usefulness and necessity of such a construct, oring constraints have been conspicuously absent from other real-time programming languages, perhaps because the scheduling is made more complex by supporting both types of constraint combining.

We further suggest that it is critical that the timing constraints be expressed at the finest possible grain level. Because finer grain yields more freedom in the compiler's scheduling, this yields the highest probability that all timing constraints can be met. Of course, more freedom in scheduling also implies a larger search space for schedules, but the user does not need to be aware of the additional complexity in the compiler.
The concept of a cyclic task in a program is a structure that is unique to real-time systems, but is commonly used. There is no obvious way to obtain the same effect using other constructs, and subtle methods destroy maintainability. Thus, we suggest that cyclic structures should be directly represented in the language, making them easy to recognize for both programmer and compiler. In fact, the language should allow multiple arbitrary cyclic tasks to be specified without concern for how the different length cycles will be converted into a single coherent schedule.

2.3 Parent language

Thus far, none of the existing real-time languages provides all the features mentioned above. Most of the real-time languages are not easy to program, not based on fine-grain tasks, and/or not designed to express relative constraints among arbitrary control operations. Thus, we propose an extension of an imperative language to facilitate the features. Even though any imperative language can be extended for those features, we develop novel language constructs onto a subset of the C language because C is used for most of the embedded real-time systems in conjunction with assembly macros for time critical portions.

The parent language we propose includes all expressions using arithmetic and relational operators, if statement, and while statement. At this time, neither aliasing nor floating point computation is supported.
2.4 Proposed language constructs

In order to specify the constraints on top of the parent language, we introduce several new constructs: timing block, temporal expression and cyclic block. A timing block defines a code segment to be manipulated as a unit for the purpose of constraint specification, while time expression specifies the relative constraints in terms of the timing blocks. A cyclic block is defined for the code sequence that runs indefinitely or until the program terminates. A program may contain more than one cyclic block.

Before illustrating the constructs, consider an example of a control program shown in Figure 3. The program is written to read data from memory locations where they are updated by the devices, and to write commands to memory locations where they are dispatched by a to the devices at desired times.

2.4.1 Timing block

A timing block is a user defined variable or a sequence of statements that could be viewed as a task in the traditional real-time systems. That is, it contains either a volatile variable (EVI) or statement(s) that consist of at least one EVI. If a timing block contains no EVI, the temporal expressions associated with the temporal variable are ignored, implying that the constraint is automatically satisfied.

The timing variable identifying a timing block appears between "@" and "::" that is followed by an EVI or a body of the block as shown in the syntax;

@tvar: variable
or
@tvar: { (statement)+ }

Here, tvar is a user defined name for the block and variable is a regular variable that is an EVI. The notation (statement)+ denotes one or more statements composing a timing block. For example, line 12 in Figure 3 defines temporal variables tv2 and tv3 that are associated with a variable sensor2 and a statement \( b = \text{sensor2} + 4 \). In this case, tv2 is a nested block of tv3.

\[1\]The volatile variables are translated into one or more instructions in low-level languages
main()
{
  volatile char action1, action2, action3;
  volatile int sense1, sense2, sense3;
  char command1, command2;
  int a,b,c,d;
  d = 23;
  cycle @tp1:(5) {
    /* Read in data which should be stored by actual sensors. */
    a = @tv1:sensor1 + 5 + d;
    @tv3: { b = @tv2:sensor2 + 4; }
    /* Specifying the timing between the reads */
    @: tv1. < tv2. + 1 | tv2. < tv1. + 1;
    @: tv3 < tv2. + 3;
    /* Action is performed based on the data read */
    if(a < b)
      @tv4: {action1 = command1;}
    else
      @tv5: {action2 = command2;}
    /* Specifying the timing for action */
    @: tv22 = tv2 + 13;
    @: tv4 > tv22. + 1;
    /* Also specifying the timing over the iterations */
    @tv6: {
      while(a < b) {
        @tv61: c = a + b - 23;
        c = c * 23 - foo();
        a = a + 1;
      }
      @tv8: {action2 = command1 + c;}
      @: tv8 > .tv8 +10;
    }
  }
  cycle @tp2:(10) {
    int c; c = 43;
    d = @tv11:sensor3 + sense1;
    @tv12: {action3 = d * c + 3;}
    @: tv12. < tv11. + 14;
  }
  /* The first action of cycle tp2 should be started
  before the the last action of cycle tp1 is completed */
  @: tp2 < tp1. + 2;
  exit(0);
}
A timing block has all the properties of regular block, such that it is lexically scoped and variable declarations are allowed in the beginning of the block. Also, the timing blocks can be nested unless they are interleaved. The interleaving of the timing blocks is disallowed because it increases the complexity and destroys the structured language feature.

### 2.4.2 Cyclic block

Cyclic block is a special case of timing block in the sense that the block is executed repeatedly until the program is terminated. Thus, compiler should treat this block differently from other blocks for analysis as well as scheduling. A cyclic block is identified by a temporal variable `tvar` and the period `δ` as defined here.

```plaintext
cycle @tvar:(δ) { statement(s) }
```

The `cycle` is a keyword indicating that the following block enclosed by `{ and } is a cyclic block, `tvar` is a temporal variable name for the block, and `δ` specifies the timing requirement that the cyclic block must complete within. That is, the statements in the timing block and looping overhead must be completed within `δ`.

The time constraints associated with a cyclic block may be expressed in the temporal expressions described later. For example, the program in Figure 3 consists of two cyclic blocks `tp1` (line 9 to 35) and `tp2` (line 37 to 42), and a temporal relation between the cyclic blocks is specified in line 45. The temporal expression in line 45 implies that the first action of `tp2` should not be executed later than 2 time units after the last action of `tp1`.

### 2.4.3 Temporal expression

A temporal expression is classified as either a temporal assignment or a temporal relation. A temporal assignment is a way of defining another temporal variable that has temporal distance from an existing temporal variable, while a temporal relation is used to express the relationship between the temporal variables.

```plaintext
@: tvar = tvar (+ | -) δ;
```
In this syntax, \( \text{tvar} \) is the name of a temporal variable, and \( \text{etvar} \) is an extended temporal variable indicating start time or completion time using prefixed-dot or postfix-dotted respectively. In Figure 3, \( \text{tv1} \) in line 14 indicates the completion time of \( \text{tv1} \) while \( \text{tv3} \) in line 15 indicates the start time of \( \text{tv3} \). Also, \( \eta \) is one of the temporal relation operator defined in the previous section, and \( \delta \) is the offset that is any time unit as small as a clock cycle. Also, the temporal relation \(( \mid \text{etvar}_1 \eta \text{etvar}_2 (+ \mid -) \delta)\) indicates zero or more temporal relations to specify ordering constraint.

As we have seen in the syntax of the temporal assignment, computation upon the temporal variables is allowed. For example, temporal assignment \( \text{tv2} = \text{tv1} + 7 \) and temporal relation \( \text{tv3} < \text{tv2} + 7 \) together implies that the start time of \( \text{tv3} \) must happen earlier than 14 time units after the start time of \( \text{tv3} \) which is equivalent to \( \text{tv3} < \text{tv1} + 7 + 7 \).

The Figure 3 illustrates one unique temporal expression for the time constraints involving the iterations. When a block that appears in iterated loop is considered, it is necessary to specify time constraints between the instances in different iterations. We introduce array of temporal variable to resolve this problem. As we see in line 31 of Figure 3, the distance between the instances are expressed as the index of the array. For example, line 31 says that the loop should be executed within 25 time units because the temporal expression requires that the first action of \( (i + 1) \)th iteration should be started earlier than the 25 time units after last action of \( i \)th iteration. At this point, we formally define \( \text{etar} \) as:

\[
\text{etvar} : \text{tvar} \{[i]\} \mid \text{tvar} \cdot \{[i]\}
\]

where \( \text{tvar} \) is a temporal variable and \( \{[i]\} \) is an optional expression of \([i]\) denoting \( i \)th instance of the iteration.

### 2.4.4 Illegal expressions

The lines 17 to 20 of the Figure 3 defines the temporal variables \( \text{tv4} \) and \( \text{tv5} \) and lines 22 to 23 shows corresponding temporal expressions, temporal assignment and temporal relation. It says
that the action in *then* clause should be performed later than 14 time units after \( tv2 \) is completed. Thus, the timing constraints is taken into account only when \( a < b \) holds. However, any temporal expression who has a pair of instructions \( \gamma_s \) and \( \gamma_e \), one from *then* clause and the other from *else* clauses are illegal because it is impossible to run both branches in an execution.

Another illegal expression is the statement whose pair of instructions necessarily include a statement with unpredictable timing property between them. Those statements are any instructions whose timings are not predictable such as `break` in a loop, unpredictable number of iterations, and unpredictable instructions involving interrupts. For example, the execution of \( tv6 \) (line 26 to 35) is repeated as long as \( a < b \) holds. Hence, if the range of \( a \) and \( b \) are non-deterministic, the temporal expression in line 34 is an illegal expression.

Temporal expression specifying a timely conflict actions is also classified as an illegal expression. Consider two expressions in line 13 and 14, and modify line 13 to \( tv3. > tv2 + 4 \). Then, \( tv3. > tv2 + 4 \) and \( tv3 < tv2 + 3 \) has no solution, implying that the code cannot have valid schedule. Thus, those two temporal expressions are illegal. These illegal expressions are found only by thorough semantic analysis.

### 3 Compilation techniques

In this section, we briefly explain the compilation techniques unique to process the language transformation. Particularly the compiler organization, data structure of intermediate representation, and cyclic block scheduling are focused because they are either very much different from conventional compilation techniques [ASU86] [Die87] or novel concept to the compilation.

#### 3.1 Compiler organization

As we see in Figure 4, CHaRTS consists of four software modules: syntax analysis, semantic analysis, scheduling, and code generation module. Each module is designed and implemented to be independent so that the software can be individually replaced for enhancement. For example, the scheduling algorithm we developed [CD95] can be replaced with better one without affecting any other modules except minimal modification on interface between them. In our implementa-
tion, those modules are written in the C language with the support of PCCTS (Purdue Compiler Construction Tool Set).

The syntax analysis module performs building an intermediate representation from a source program. The intermediate representation of CHaRTS is unique as described in section 3.2 because of the timing constraint specification and cyclic blocks.

The function of the semantic analysis module is to verify a program context against the language semantics including:

- Does the timing block contain at least one EVI?
- Is the temporal expression semantically legal?
- Is it compliant to the parent language?

The objective of the code scheduling module is to reorganize the instruction sequence to meet all the timing constraints as well as the logical correctness. This module and code generation module are entirely system dependent and the timing properties of the hardware components and the instruction sets can be either hard-wired into the compiler or read from system file as shown in the Figure 4. The input to the scheduling module is the tuples whose timing characteristic is close to the object code so that the execution time of the tuples are preserved at code generation time.
3.2 Intermediate representation

In addition to the traditional role of a lexical analyzer and parser, the syntax analysis module extracts constraints and information of timing blocks from the program and builds an intermediate representation. As depicted in Figure 5, the data structure of the intermediate representation consists of two tables and three trees; The tables are a symbol table (SymTab) and a table for temporal variables (TVarTab), while the tree structures include an abstract syntax tree (AST), a cycle syntax tree (CST), and a temporal syntax tree (TST) that are all child-sibling trees [PDC92].

Although only AST and SymTab are sufficient data structure in conventional compilers, the compiler for real-time systems we propose needs more information, and they are organized in TVarTab to store information of the timing blocks such as block type, entry point, and exit point, CST to manipulate all the blocks to be executed periodically, and TST to represent the temporal expressions.

In particular, manipulation of CST in conjunction with temporal expressions specifying the timing constraints among the cyclic blocks is quite complex and is new concept to the compiler community because multiple cyclic blocks need to be executed indefinitely, considering the period and timing constraints among the cyclic blocks.

![Figure 5: Data Structure in Intermediate Representation of CHaRiS](image)

Figure 5: Data Structure in Intermediate Representation of CHaRiS
3.3 Cyclic block scheduling

Scheduling the cyclic blocks is very different from the conventional compilation techniques because conventional compilation does not have a notion of interleaving loops. The cyclic blocks need to be integrated into a single cyclic block that can be executed without an explicit dynamic scheduler or a dispatcher. In this section, we develop a novel scheme to integrate the given cyclic blocks, associated with different periods, into a single cyclic block. The integrated cyclic block executed indefinitely or semi-indefinitely until the program terminates.

Let $\psi_1, \psi_2, \ldots, \psi_n$ be the cyclic blocks associated with the periods $p_1, p_2, \ldots, p_n$; that is, the execution of the first temporal variable in block $\psi_i$ should be started every $p_i$. The re-arrangement of the cyclic blocks into a single cyclic block $\Psi$ is processed as follow using the fundamental property of scheduling of the schedule for a periodic task [LM80].

1. compute $p = \text{LCM}_{i=1}^{n}(p_i)$.
2. $\forall i$, replicate the instructions in $\psi_i$ for $p/p_i$ times.
3. $\forall i$, expand the timing constraints. Notice that the expansion is a simple addition of temporal
expressions in our model because dependence is a subtype of simple constraint. Define $\gamma^k_{i(x)}$ as the instruction $x$ in $k$th instance of $\psi_i$, say $\psi^k_i$, and $\psi_i$ as the original cyclic block associated with period $p_i$. Likewise, $\gamma_{i(x)}$ represents the instruction $x$ in $\psi_i$. The constraint expansion is performed based on the following analysis.

- **self expansion**: The replicated instructions add timing constraints to maintain the execution order of those replicated instructions. At this expansion, the periodic constraint is enforced on the first control action. Namely, $\forall i$ and $\forall j$, a temporal expression $\gamma^k_{i(x)} < \gamma^{k+1}_{i(x)} + \delta$ is added where $\delta = p_i$ if $\gamma_x$ is the first control action in $\psi_i$ or $\delta = 0$ otherwise. The graphical representation of self-expansion is depicted in (a) of Figure 6 when instruction $I$, $\gamma^1_{i(2)}$ is the first control action in $\psi_i$.

- **cycle-carried expansion**: Suppose $\gamma^k_{i(y)}$ is dependent on $\gamma^k_{i(y)}$, say $\gamma^k_{i(y)} < \gamma^{k+1}_{i(y)}$ for $k > 0$. In this case, the dependence is enforced for every replicas of $\psi_i$, adding temporal expressions of $\gamma^k_{i(y)} < \gamma^{k+1}_{i(y)}$ for $1 \leq k \leq n-1$. This expansion is applied to every cyclic carried dependences and shown in (b) of Figure 6. In the figure, $\gamma^{k+1}_{i(2)}$ uses the result of $\gamma^k_{i(3)}$; thus, the temporal expression $\gamma^k_{i(3)} < \gamma^{k+1}_{i(2)}$ for $k > 0$.

- **intra-cycle expansion**: Temporal expressions specifying the constraints between the instructions in the same block are replicated as necessary. That is, for $\gamma^0_{i(x)} < \gamma^0_{i(y)} + \delta$, $\gamma^k_{i(x)} < \gamma^k_{i(y)}$ is added in every replicas of $\psi_i$. In (c) of Figure 6, the temporal expression $\gamma^k_{i(3)} > \gamma^k_{i(2)} + \delta$ is replicated in $\psi_i$.

- **inter-cycle expansion**: Temporal expressions specifying the constraints between the instructions beyond one block (not within a block) are replicated as necessary. Let $\xi^k_{i(x)}$ be a starting time of $\gamma^k_{i(x)}$ in $\psi^k_i$. Then, the range of $\xi^k_{i(x)}$ can be expressed as $(k-1) \times p_i \leq \xi^k_{i(x)} \leq (k+1) \times p_i$. Thus, the inter-cycle temporal expression $\gamma_{i(x)} < \gamma^p_{j(y)} + \delta$ yields the temporal expressions $\gamma^k_{i(x)} < \gamma^p_{j(y)} + \delta$, $\forall p$ such that $\psi^p_i$ satisfies $\xi^p_{i(x)} \wedge \xi^p_{j(y)} \neq 0$. Consider (d) of Figure 6. When we assume that $p = \nu 1$ or $\nu 2$ for $\xi^k_{i(2)} \wedge \xi^p_{j(1)} \neq 0$, temporal expressions $\gamma^k_{i(3)} < \gamma^p_{j(2)} + \delta$ and $\gamma^k_{i(3)} < \gamma^p_{j(2)} + \delta$ are added.

4. $\Psi$ is scheduled by an instruction level fine-grain scheduling algorithm. One of the approaches to find a valid schedule by using a genetic search based algorithm is given in [CD95].

---

3 We assume that the $p_i$ implicates that the first control action in $\psi_i$ should be repeated every $p_i$. The other instructions in $\psi_i$ are correctly executed as long as the expanded dependences are not violated.
4 Conclusion

Building a statically-scheduled real-time system is not as difficult as it first appears— if we have an appropriate real-time language and compiler technology. After making significant strides toward development of the compiler technology, we believe that the key to building the language is providing a very general model which both the programmer and the compiler can easily understand and manipulate. This paper has proposed such a language.

Basic features of this language include:

- The basic syntax of a conventional language (in this paper, C) and compatibility with that language's programming idioms
- The ability to place relative time constraints on arbitrary pairs of fine-grain operations or on cyclic blocks
- The ability to specify arbitrary types of relative time constraints (e.g., any of before, after, concurrent, and exclusive combined by either anding or oring)

Some of these features greatly complicate the compiler's job by enlarging the schedule search space or by adding complexity to the evaluation of constraints. However, we have made good progress toward solving these problems, and nothing in the language lies far beyond the ability of the new compiler technology we have developed thus far.

Currently, the compilation techniques like the control structure analysis, the code scheduling with a hierarchical decomposition scheme, and the code generation for a parallel execution model (PAPERS) are under development. Indeed, current and future work centers on the implementation of a complete compiler for this language: CBaRTS (Compiler for Hard Real-Time Systems).
References


RTsynchronizer, a High-Level Language Construct for Specifying Real-Time Constraints in Distributed Concurrent Systems

Shangping Ren
Dept. of Computer Science, University of Illinois at Urbana-Champaign

We argue that the specification of an object’s functional behavior and the timing constraints imposed on it may be separated. Specifically, we describe RTsynchronizer, a high-level language construct for specifying real-time constraints in distributed concurrent systems. RTsynchronizers describe the real-time relation between events over a group of objects. Objects in our system are defined in terms of the actor model extended with timing assumptions. A number of examples are given to illustrate the use of RTsynchronizers -- including periodic events, simultaneous events, exception handling, producer-consumer, and a control process.
The handling of state manipulation in different programming languages varies widely. This talk surveys what the major paradigms are, the design tradeoffs and the issues they raise. Of special focus are Algol-like languages, Lisp-like languages and recent functional programming languages with state-manipulation. (This talk was originally given as the opening tutorial at the recent SIPL workshop in San Francisco.)
Foundations for a Semantics of Subobjects
Extended Abstract

Jonathan G. Rossie, Jr. and Daniel P. Friedman
{jrossie,dfried}@cs.indiana.edu
Department of Computer Science, Indiana University
215 Lindley Hall, Bloomington, Indiana 47405

May 17, 1995

Abstract

We find that by answering three essential questions about the static
properties of inheritance hierarchies, we gain significant insight into the
kind of inheritance best exemplified by C++. Multiple inheritance, the
dominance rule for disambiguation, and the presence of virtual and non-
virtual base classes lead to quite a complicated semantics. By abstracting
the notion of a subobject, we have designed an algebra for answering the
three questions in a way that manages all of these complications.

Overview

The multiple-inheritance model found in C++[2, 6], which derives largely from
Krogdahl's multiple inheritance for Simula[3], can be seen as the result of one
fundamental design imperative: that distinct storage should be allocated for
each instance variable, regardless of name identity. This design choice, which
we refer to as the subobject-integrity imperative, leads to a space of closely-
related semantics in which the C++ and Simula models represent two points.

It has been our ongoing project to find a semantic framework for subobject
integrity such that the semantics of the C++ model, for example, may be seen
as a specialization. The key issue that complicates such a semantics is the
ambiguity that can arise in method calls and instance-variable references.

The problem stems from the lack of unique labels. As mentioned earlier, the
names of the instance variables may not be unique. The same holds for method
names: a method must be associated with the expected instance variables,
so name equivalence does not imply method equivalence. Moreover, it is not
sufficient to label instance variables and methods with the class name from
which they are inherited, since the same class, when reached through different
inheritance paths, can result in distinct storage.

Both authors were partially supported by NSF grant CCR-9302114.
The solution to the problem of labels is to lift the notion of a subobject out of the implementation and into our semantics. We devise a labeling scheme for subobjects and use these subobject labels as the unique labels for identifying instance variables and methods.

The labeling scheme for subobjects is complicated by the presence of shared inheritance. In C++, virtual classes provide a means of specifying that repeated inheritance from the same base class should not lead to distinct storage in the derived class; that is, it should not lead to separate subobjects. We call this shared inheritance. Without shared inheritance, a subobject may be uniquely specified by an inheritance path of class names; the arcs in the inheritance path are homogeneous. With shared inheritance, however, the arcs are either shared or proprietary, and the labels must only encode those classes in the path that lie above the highest shared arc. (We use the common intuition that a base class is above its derived classes.)

Shared classes unfortunately contribute to the complexity of the inheritance model. Cargill[1] makes an ample case for the complexity of the system, as do the various attempts at formalizing the C++ object model, most notably Snyder's[5]. Cargill, arguing against the introduction of Stroustrup's multiple inheritance system into C++, complains that (p.71) "Multiple inheritance in C++ is complicated to learn, write and read." He is particularly opposed to shared base classes, which he feels require too much nonlocal information to understand. Snyder’s model is similar to ours in that it deals with subobjects on an abstract level, but it does not include shared base classes, nor does it model the effects of non-shared repeated inheritance of the same class, which Snyder refers to as a corner case of the language, distinctive to C++ multiple inheritance: (p.10)" ... the extra complexity needed to handle this case is not justified."

It is not our purpose to argue for or against the inheritance model, but merely to devise a concise expression of it. Rather than attempting to formalize the entire language, or even the entire object system, we restrict our formalism to the resolution of the three questions that we feel are at the heart of understanding the complexities introduced by the subobject integrity imperative:

**Question 1 (subobjects)** What is the set of subobjects that comprises an instance of a given class?

**Question 2 (methods)** For a specific method name and an instance of a given class, which subobject will be cast-to\(^1\) as a result of the call? (Or will it be ambiguous?)

**Question 3 (instance variables)** For a specific instance-variable name and an instance of a given class, which subobject will contain the value? (Or will it be ambiguous?)

\(^1\)Since we have no explicit notion of a type, casting is merely a matter of accessing a particular subobject.
In answering just these three questions, each of which deals with static properties of the hierarchy, we are able to strip away an enormous amount of complication, including protection/privatization, method values, instance-variable values, and even the class/instance distinction. We consider it essential, however, that the features that genuinely complicate the inheritance model—multiple inheritance, shared and non-shared classes, virtual and non-virtual methods, and lack of linearization—are retained. Despite our formal simplifications, and partly thanks to them, we have found this model to be an invaluable aid in our practical understanding of this kind of inheritance[4].

References


Object Interactions as First Class Objects: From Design to Implementation

Mahesh H. Dodani, Kok Siew Gan and Lizette Velazquez
Department of Computer Science
The University of Iowa
Iowa City, Iowa 52242
email: {dodani, gan, velaz}@cs.uiowa.edu
Phone: (319) 353-5134
Fax: (319) 335-3624

Abstract

Collaborations between objects make up the dynamic behavior of OO software. These collaborations among objects require careful design and implementation. Treating the interactions as responsibilities that are integrated in the participating objects, results in tight coupling between objects. Tight coupling increases complexity and reduces reusability. Object interactions need to be first class objects from design to implementation. Our research provides a unified approach to model and implement these interactions as first class objects. During analysis and design, they are modeled using DynaSpecs. During implementation, they are coded with a new language construct called Compositions. DynaSpecs and Compositions provide a consistent support for object interactions within each phase of the OO lifecycle.

1 Introduction

Current OO methodologies [Boo94, Cole94, Jac92, Rum91] model the static structure and dynamic behavioral aspects of OO systems. Static models use entity relationship diagrams to describe attributes in objects and their structural relationships. There are two extremes of modeling relationships: explicit and implicit models. Implicit models use attributes in the associated classes. Explicit models of relationships model them as first class citizens [Tao95].

The dynamic model aim to describe the complex interactions between objects at run time. Interactions among collaborating objects are typically identified in the analysis phase through scenarios (use cases) that expose the functionality of the system. Most methods use object interaction diagrams to model interactions during the design phase. These diagrams capture the objects involved in the interaction and the corresponding flow of message passing. The interactions among objects encompass the assumptions that each makes about other. To understand objects interactions we need to know not just the associated message flow and the participating objects, but also what the state changes and interaction behavior of each the participating objects.

In current methodologies all this information is distributed over the entire spectrum of models: object models, operational models, interaction graphs and state machines. Furthermore, in the design of object classes, the behavior of interactions is scattered across the collaborating classes. This distribution of information and behavior across models increases the redundancy of information, complexity of the classes, and the coupling among classes. This results in difficulties in establishing invariants and properties of the system, difficulty in ensuring proper access to methods, and difficulty in ensuring consistent update of participating objects to reflect dynamic changes.

The disadvantages of treating object interactions as integral behavior of the participating objects are as follows:

- increase complexity when additional interaction responsibilities are placed on the participating objects,
- tight coupling between participating objects,
• reuse of interactions may include unnecessary tie to other interactions irrelevant to the reuse,
• difficulty in handling many to many relations (in which of the participating object should the interaction responsibilities be placed), and
• abstracting the object interaction is difficult.

Object interactions should be supported as first class values in Object-Oriented software. They need to be first class objects from design to implementation. Our research provides a unified approach to model and implement these interactions as first class objects.

At analysis and design our research develops DynaSpecs. DynaSpecs supports the specification of objects and interactions as first class objects. DynaSpecs model the object interactions using state transitions. The state transitions represent events of message calls that mirror the dynamic behavior of object interactions. The formal semantics of the model is based on the idea of a history which captures the sequence of operations within interactions along with state changes undergone by the participating objects.

Current OO programming language allow programmers to implement object interactions as first class objects by using classes with variable references to participating objects. However, this implementation have the following disadvantages:
• the variable reference allows access to all public methods of the participating object, even methods that are not relevant to the interaction,
• there are no clear separation between participants in the class encapsulation, the participants are accessed through variables,
• in a statically typed language like C++, the participating objects must belong to the variable class type. A suitable participant from a different class type cannot be a participant safely.

We provide a new construct called Composition to implement object interactions. Compositions, as well as DynaSpecs allow programmers and systems designers to
• clearly define each participant in the interaction,
• focus on functionality of the interactions by defining the behavior and responsibility of each participant,
• provide an explicit dynamic lifetime of an object interaction,
• restrict visibility to only methods of participants relevant to the interaction,
• facilitate reuse of interactions by allowing parameterized interactions, and inheriting Compositions,
• provide the necessary context to clearly separate responsibilities of participants, an encapsulation of the entire interaction, encapsulation of each participant and limited access to participating objects, and
• allow the selection of suitable participating objects based on the methods they implement, not the class hierarchy they belong to.

The semantics of DynaSpecs and Compositions have been precisely defined using denotational semantics. They have been tested with prototype executable of the constructs written in ML. These constructs can be added to any existing OO system known to us. We have successfully implemented the constructs and incorporated them in an Object-Oriented development environment. In this paper, we will explain how to use our constructs to model and implement object interactions. We have modeled the MVC framework using DynaSpecs and implemented the new MVC design using our prototype language. The paper will present our results.

The rest of this paper is organized as follow. Section 2 explains and motivates the need for DynaSpecs and Composition through a case study. It begins by explaining the interactions in the MVC framework. It then models and implements them using DynaSpecs and Compositions. Section 3 concludes the paper.

2 Case Study

One of the most well known framework is a Graphical User Interface framework called Model, View and Controller [Kras88, Mod88]. We will identify the object interactions in a MVC framework, and model and implement them using DynaSpecs and Compositions. The new MVC design will be used to implement a calculator. Throughout this case study, we explain how to use interactors in modeling and implementing the calculator with MVC.
2.1 Model, View and Controller Framework

There exist three major object interactions in applications developed using (MVC) framework:

- Views register themselves as dependents of their model and keep themselves consistent with the state of their model (i.e. with the internal state of the application).
- Controllers and models interact to translate user input to invoke system functionality.
- View-Controller interaction is needed to determine the controller in charge of user interaction.

Current approaches to model the MVC interactions, make the model, view and controller classes tightly coupled. For example the view class includes behavior that deals with updating itself to keep consistency with the model, behavior to interact with the controller and behavior to traverse the view hierarchy. The associated methods access the state of the model or interpret the state of the model, translates controller inputs, and finds a view that has the cursor. Understanding and reusing the MVC framework require dissecting each of the classes to pinpoint the behavior involved in each object interaction.

Our constructs allow explicit definition of each MVC object interaction. This provides a clear separation of each object's basic behavior and separation of interactions and their participants. The interactions described above, can be modeled by the following interactors:

- Consistence interactor to keep dependents consistent with the state of an object.
- Translate interactor to translate controller inputs into operations for performers.
- Control interactor to determine which controller gets control.

These interactors perform the functionality of the MVC framework. Control interactors within an application represent a tree structure responsible for invoking the correct controller to get user input. In particular, a top control interactor determines which of its sub-control wants control. Once this is determined, it sends "controlActivity" to the sub-control. In "controlActivity", it asks each of its translate interactor to translate any user input. The translate interactor asks its performers to perform the input operation. The performers make their dependents consistent using the consistence interactor. The figure below illustrates our explanation.

![MVC Interactors](image)

Control interactor usually have only one corresponding translate interactor. Each translate interactor usually has one performer. Each performer participant of a translate interactor has exactly one consistence interactor.

The interactors described here can be documented using design patterns. Design patterns describe design decisions using a consistent format to help provide a common vocabulary, to help to understand and reuse the design. The format includes discussion of implementation issues and provides sample code. This format aids implementation of the design. Interactors provide a construct to implement some structural and behavioral design patterns [Gam95]. The close resemblance of the interactor code to design patterns illustrates the ease in capturing the design using interactors.

The consistence interactor keeps dependents consistent with the state of an object. According to [Gam95], the consistence interactor is similar to an observer design pattern. After a model changes its state, the view will receive an "update" message to invalidate its view. The following figure illustrates how consistency is achieved in...
MVC. The figure shows the message flow between the participants in the interaction. Note that the sequence of messages is ordered as shown by the numbering.

The consistency composition consists of an object participant and a list of dependent participants. It handles consistency by sending the message “makeConsistent” to the consistency composition object. The consistency object gets the object participant’s value, maps it for each dependent, and passes the value to the dependents, so they may update themselves if necessary. The following diagram illustrates the message flow.

To emphasize how the responsibilities for each object interaction are decoupled from the model and dependents, the patterns in the figure stress where they are placed. In the MVC, the Model class includes View responsibilities, the Controller includes Model and View responsibilities, and the View class includes View hierarchy responsibilities. The interactor liberates basic objects from having interaction responsibilities as part of their behavior by placing them in the corresponding interactor. Interactors act as an abstraction of the object interactions and provide communication between the collaborating objects.

2.2 Calculator Application

To reuse the MVC framework architecture, we will apply the interactors to design and implement a simple calculator. Each of the interactors described earlier will be inherited with extensions relevant to the calculator. The calculator consists of a control hierarchy which is captured through the control interactor. This hierarchy replaces the traditional view hierarchy used in current designs and implementations. Each view has a translator which translates the input received by its associated controller. When the calculator model changes, the display view must be updated. This is captured in calculator consistency interactor. Conversely, when the display view changes the number of display digits, the calculator model needs to take that into account. If the calculated result is larger than the number of display digits, the calculator model will be in an error state. This is captured in view consistency interactor. Each of the composition described will be inherited with extensions relevant to the calculator. The basic classes are inherited to CalculatorModel, DigitView, KeypadView, ButtonView, DigitController, KeypadController and ButtonController. They perform operations specific to their responsibilities with no operations related to any object interactions. Figure 4 shows the calculator along with its basic objects.
The interactors in the calculator application are:

- CalculatorConsistence and ViewConsistence are consistence interactors.
- DigitTranslator, KeypadTranslator and ButtonTranslator are translate interactors.
- StandardControl, DigitControl, KeypadControl and ButtonControl are control interactors.

The following figure provides an object diagram for the calculator. Note that basic MVC objects have no references to other objects, they are responsible for their own basic behavior. Only interactor objects are involved in coupling objects specific to the interaction.

Note that once the implicit reference (dotted arrow lines) are linked in the interactors, the responsibilities and access to the basic objects are clearly defined. Each interactor will access only methods that they need, unlike the MVC implementation without interactors.

2.3 DynaSpecs

This research develops DynaSpecs, a specification method for describing the dynamic behavior of object interactions. The models of interactors are parameterized machine classes. An interactor defines the interaction...
between two or more entities. We call these entities participants. The formal generic parameters are binded to object and/or interactor classes to create concrete interactor machine classes. The classes given as arguments constitute the participant classes.

Apart from its own attributes and methods, an interactor may require the participants to define some attributes and/or methods. These requirements are specified in the interactor’s definition and conform the contractual aspects of the specification. The contractual aspects determine the minimum requirements for classes to be able to participate in an interactor. It is a contractual responsibility of the participant’s classes to define their corresponding deferred attributes and methods. The methods and attributes that conform the contractual obligations are used to define the interactor’s behavior.

The required attributes and methods defined by the participant classes do not need to have the names used by the interactor. In such cases a table of mapping must be provided. This table allows the mapping of names of required attributes and methods as specified by the interactor class to the corresponding attributes and methods on the participant classes, and vice versa. It is basically a translation table used by the interactors to talk to its participant entities in the language they understand.

Actual interaction behavior is defined in terms of a finite state machine. We will use StateCharts [Harel87] as the state machines. Each machine has a default transition

\[
\text{(post-condition)} \rightarrow \text{State}.
\]

This transition expresses the expected initial values of the attributes within in its post-condition. Other transitions have the following format

\[
\text{State}_1, \{\text{pre-condition}\} \rightarrow \text{State}_2.
\]

Pre-conditions specify the conditions that must hold for a transition to fire. Post-conditions specify the changes to attributes as a result of firing the transition. There could be transitions with no specified pre-condition. This simply indicates that the transition can be fired when the message is received. Underlined attributes refer to the value of the attributes before execution of the method on the transition. An action can be optionally specified on a transition. An action refers to a message to be sent if the transition is fired.

A transition of the form

\[
\text{State}_1, \text{TR(condition)} \rightarrow \text{State}_2
\]

is a transition that is processed every time a transition is fired. If the condition on the TR transition is true then the corresponding TR transition is fired.

In this section we concentrate in the modeling of the consistence interactor, and the basic object classes DigitView and Calculator using DynaSpecs.

### 2.3.1 Modeling Consistence Interactor with DynaSpecs

DynaSpecs allows interactors to be explicitly modeled. Figure 6 shows the parameterized DynaSpecs specification for ConsistenceInteractor. It requires Model to provide the information to be displayed. It also requires the participant View to know how to update itself using the provided information to display. The consistence interactor requires View to be updated. As a result, View redisplay itself according to the Model’s information provided by the interactor. This knowledge is kept and managed by the corresponding interactor, ConsistenceInteractor. Now we have an encapsulated interactor that explicitly describes the interaction between objects. All behavior and aspects of the collaboration are explicitly stated within the interactor. The effects of the collaboration upon the participating objects as well as their contributions can be found within the interactor. It is not scattered among the collaborating objects. This DynaSpec can be instantiated by providing the actual collaborating classes: Calculator and DigitView.

```plaintext
CONSISTENCEINTERACTOR[MODEL, VIEW]
  METHODS    [MakeConsistent;]
  ATTRIBUTES [information;]
MODEL       METHODS    [GetValue;]
VIEW        METHODS    [Update;]
```
2.3.5 Modeling the Calculator Application

The specification of the class View is as shown on Figure 7.

\[
\text{DIGITVIEW ATTRIBUTES } \{\text{displayedInfo; }\}
\]

\[
\text{DIGITVIEW METHODS } \{\text{Display; Update; ClearDisplay; }\}
\]

\[
\text{DIGITVIEW}
\]

\[
\text{Valid} \quad \text{Invalid}
\]

\[
\text{Update(displayedInfo)/ClearDisplay}
\]

\[
\text{ClearDisplay/Display}
\]

Transitions:
1. \(\text{} \) \(\rightarrow\) \text{Invalid}
2. \text{Invalid} \(\rightarrow\) \text{Update(displayedInfo)/ClearDisplay} \(\rightarrow\) \text{Invalid}
3. \text{Invalid} \(\rightarrow\) \text{ClearDisplay/Display} \(\rightarrow\) \text{Invalid}
4. \text{Invalid} \(\rightarrow\) \text{Valid}
5. \text{Valid} \(\rightarrow\) \text{Update(displayedInfo)/ClearDisplay} \(\rightarrow\) \text{Invalid}

\[
\text{Figure 7: Specification of Object Class View}
\]

The object class Calculator is the actual model of the application. Figure 8 defines its behavior. Calculator accepts expressions entered in infix notation. It accepts one digit or operator at a time. Calculator provides methods that respond to each of the keys on its keypad (e.g. digits: one, two, three,... and operators: add, multiply,...). In the Calculator state machine digit \(\in\{\text{zero ... nine}\}\) and op \(\in\{\text{add, multiply, ...}\}\).

\[
\text{CALCULATOR ATTRIBUTES } \{\text{accumulator; operand; operator; }\}
\]

\[
\text{CALCULATOR METHODS } \{\text{Multiply; Add; Equal; Clear; One; Two; Three; }\}
\]
Transitions:
1. \{accumulator = 0 AND operand = 0 AND operator = NIL\} \rightarrow \text{AcceptDigit}
2. \text{ProcessEquation}() \text{Clear} \{\text{operator} = \text{NIL AND accumulator} = 0 \AND \text{operand} = 0\} \rightarrow \text{ProcessEquation}
3. \text{AcceptDigit}() \text{digit} \{\text{operand} = 10 \ast \text{operand} + \text{digit}\} \rightarrow \text{AcceptDigit}
4. \text{AcceptDigit}() \text{op} \{\text{operator} \neq \text{NIL} \Rightarrow \text{operand} = \text{solve(accumulator, operator, operand)} \AND \text{operator} = \text{op}\} \rightarrow \text{OperatorAccepted}
5. \text{OperatorAccepted}() \text{digit} \{\text{accumulator} = \text{operand AND operand} = \text{digit}\} \rightarrow \text{AcceptDigit}
6. \text{AcceptDigit}() \text{operator} \{\text{operand} = \text{solve(accumulator, operator, operand)} \AND \text{operator} = \text{NIL}\} \rightarrow \text{EqualAccepted}
7. \text{EqualAccepted}() \text{digit} \{\text{accumulator} = \text{operand AND operand} = \text{digit}\} \rightarrow \text{AcceptDigit}
8. \text{EqualAccepted}() \text{op} \{\text{operator} = \text{op}\} \rightarrow \text{OperatorAccepted}

Figure 8: Specification of Object Class Calculator

Note that neither DigitView or Calculator has knowledge of their mutual existence or collaborations. They have no knowledge of the way they interact. This facilitates the reusability of these models in other applications. Similarly, the encapsulation and abstraction of interactions modeled as parameterized interactors in DynaSpecs facilitates reusability and extendibility of interactions.

A concrete interactor machine is obtained from a parameterized interactor machine by binding the parameter list with actual machine classes. A concrete interactor machine class is defined by its class name and the class names of its participant classes, the list of contractual attributes, mapping tables, and the state machines.

Let ConsistencyInteractorMachine be the parameterized interactor machine class defined previously. The object classes DigitView and Calculator are defined as follows:

\[
\begin{align*}
\text{DigitViewMachine} &= (\text{DIGITVIEW, [displayedlnfo], [Update, ClearDisplay, Display]}, \text{StateMachine}). \\
\text{CalculatorMachine} &= (\text{CALCULATOR, [operand, operator, accumulator]}, \\
&\quad [\text{One, Two, Three, Add, Multiply, Equal, Clear, ...}], \text{StateMachine}).
\end{align*}
\]

The binding of ConsistencyInteractor to Calculator and DigitView as actual parameters defines the concrete interactor CalculatorConsistency class

\[
\begin{align*}
\text{CalculatorConsistencyClass} &= \text{BindParameterizedInteractor}[\text{ConsistencyInteractorMachine}] \{\text{Calculator, DigitView}\} \\
&\quad = ((\text{ConsistencyInteractor, [Calculator, View]}), \\
&\quad ((\text{information}), [[\text{makeConsistent}, \text{GetValue}, \text{Update}])), \\
&\quad \text{StateMachine})
\end{align*}
\]
The CalculatorConsistenceClass definition specifies that Calculator and View are actual participant classes. The semantics of DynaSpecs does not restrict the participant classes to be object classes only. They can be interactor classes as well. The only restriction is that the proposed participant classes have to conform to the contractual obligations specified by the parameterized DynaSpec.

An instance of an object class is a 4-tuple defined by a unique identifier, its class name, its memory, and its current state. A memory keeps the attributes along with their corresponding values. The state of an instance refers to its current state with respect to the state machine that defines its behavior.

A new object is created by providing the object machine class and a unique identifier. Let's create an instance of Calculator called calculator.

```plaintext
calculator = NewObject[calculator] CalculatorMachine
= (calculator, Calculator, MEMORY, STATE)
where MEMORY = (calculator.l(operand, 1.), (operator, 1.), (accumulator, 1.)) and
STATE = NIL
```

The initial state of each machine component is NIL (i.e. no transition has been fired); and the initial values of the attributes in the object's memory are undefined (i.e. no values have been assigned to the attributes).

To create an interactor instance, a unique identifier for the new interactor, the interactor class that defines its behavior and the list of participants' ids must be provided. An instance interactor has a composed class name. It is formed by the interactor class name and the participants' classes. Since the Id of each participant is part of interactor instances, every interactor knows exactly who its actual participants are. An interactor cannot be created unless the actual participant entities exist. The actual consistency interactor, calculatorConsistence, is defined as follows:

```plaintext
consistency = NewInteractor[calculatorConsistence] CalculatorConsistenceMachine
[calculator, digitView]
= ((calculatorConsistence, [calculator, digitView]), (CalculatorConsistence,
[Calculator, DigitView]), MEMORY, STATE)
where MEMORY = [(information, 1.)], and STATE = NIL
```

A history is generated from a given test list [Jac92] according to the state of the entities in the environment. A test list is a list of test cases. A test case is a message sent to a particular entity within the environment. If the message has formal parameters, their actual parameters must be provided in the test case.

[(calculatorConsistence, MakeConsistent) is a test list composed of one test case. The message MakeConsistent is sent to calculatorConsistence.

Let us assume that the calculator operand value is 1, and that digitView is the Invalid state. The following is the history generated for the test list

```plaintext
[(calculatorConsistence, MakeConsistent, Inconsistent),
(calculatorConsistence, GetValue, Inconsistent),
(calculator, GetValue, ProcessEquation),
(calculatorConsistence, Update(1), Inconsistent),
(digitView, Update(1), Invalid),
(digitView, ClearDisplay, InValid),
(digitView, Display, Valid)].
```

2.4 Compositions

This section explains the Composition language construct to implement interactors. Following the example above we will concentrate on the consistence interactor.

2.4.1 Consistence Composition

Before diving into the code, a few words about the syntax is in order. To define a composition, the programmer declares shared variables, defines the interfaces to access participating objects, and defines composition procedures and initialization statements.
Each participant definition in the composition is a class definition. The optional word "PARTS" is used when defining a list of participants of the same participant type. For example, the consistency composition may have multiple dependents which can be defined as "PARTS" participant. The composition procedures (Procedures) are visible to all participants. The responsibility of each participant may be defined in the participant class or defined as abstract procedures. The abstract procedures will be replaced by procedures belonging to actual objects attached to the participants. The actual objects are bound to the participants using the RELATE clause (explain later). Therefore, only objects that define the abstract procedures can fulfill the obligation as the participant in the composition.

Now, the consistency composition code.

```
COMPOSE Consistence ()
CLASS ModelParticipant ()
   ABSTRACT getValue ():INTEGER;
END;

PARTS CLASS DependentParticipants ()
   ABSTRACT setValue (newValue: ModelParticipant);
   ABSTRACT update ();
   PROCEDURE setDependentValue (newValue: ModelParticipant) =
      BEGIN SELF.setValue (newValue.getValue()) END;
END;

PROCEDURE makeConsistent (changeContext: INTEGER):INTEGER =
VAR dependentValue: ModelParticipant;
   dependents: LIST DependentParticipants;
   dependent: DependentParticipants;
BEGIN
   dependents:= CSELF.DependentParticipants ();
   WHILE LENGTH (dependents) > 0 DO
      dependentValue:= CSELF.ModelParticipant.getValue ();
      dependent:= HEAD (dependents);
      dependent.setDependentValue (dependentValue);
      dependent.update ();
   END;
   END;
BEGIN END;
```

The abstract procedure "getValue" in ModelParticipant, "update" and "setValue" in DependentParticipants will be replaced when the Consistence is instantiated. Each participant may be referred to itself as "SELF" which is done in the "setDependentValue" method. The composition may be referred to itself as "CSELF" which is done in the composition procedure "makeConsistent". To refer to a specific participant procedure, it must be qualified with the participant name. For example, ModelParticipant.getValue(). The entire
list of objects for each "PARTS" participant is returned when the participant name is called. For example, calling CSELF.DependentParticipants() return the list of all dependent objects. As illustrated in figure 3, the Consistence composition object receives the "makeConsistent" message. This message will make all the calculator dependents consistent. To make them consistent, it must first get the model state by sending the "getValue" message to its model. It then send the "update" message to all its dependents.

We instantiate and relate objects using the RELATE clause. Objects are related by passing actual object as arguments. The syntax for the RELATE clause is

\[
Id_{\text{compose object}} = \text{RELATE} \; \text{Typename}_{\text{compose}} (\text{object}_1, \ldots, \text{object}_n)
\]

Figure 11: RELATE syntax.

The object arguments may be a list of objects if they are bind to "PARTS" participants. The following code instantiate a calculator's Consistence composition and relate its model and dependents to specific participant objects.

```
digitView:= NEW (DigitView);
calculator:= NEW (Calculator);
calculatorConsistence:= RELATE CalculatorConsistence (calculator, [digitView]);
```

Figure 12: Instantiating calculator consistence composition

CalculatorConsistence is a subCompose of Consistence composition. It includes code specific to maintaining the consistency between a display view of the calculator call digitView and its model, the calculator. Calculator is the model and digitView is the dependent of calculatorConsistence. The model, calculator implements and replaces the abstract procedure "getValue". The dependent, digitView implements and replaces the abstract procedures "update" and "setValue". Since the DependentParticipants is a "PARTS" declaration, there may be more than one dependents. The binding of dependents with one element in a list, the digitView, instantiate only one "PARTS" participant.

### 2.4.2 Calculator Compositions

The translate and control compositions are implemented similar to the consistence compositions. They define the abstract design pattern for the interaction. Each composition is then subComposed to include application specific code to implement the calculator. They are the CalculatorConsistence and ViewConsistence as subCompose of the consistence composition, DigitTranslator, KeypadTranslator and ButtonTranslator as subCompose of the translate compositions, and StandardControl, DigitControl, KeypadControl and ButtonControl as subCompose of the control composition.

The following code instantiates and relates all objects in a calculator.

```
keypadView:= NEW (View);
keypadController:= NEW (KeypadController);
digitView:= NEW (DigitView);
digitController:= NEW (Controller);

calculator:= NEW (Calculator);
calculatorConsistence:= RELATE CalculatorConsistence (calculator, [digitView]);
viewConsistence:= RELATE ViewConsistence (digitView, [calculator]);

digitTranslator:= RELATE DigitTranslator ([digitView], digitController);
digitTranslator.PerformerParticipants ().initializeConsistence (viewConsistence);
digitControl:= RELATE ControlInteractor (digitController, digitView);
digitControl.ControllerParticipant InitializeTranslators ([digitTranslator]);

keypadTranslator:= RELATE KeypadTranslator ([calculator], keypadController);
```
keypadTranslator.PerformerParticipants ().initializeConsistence (calculatorConsistence);
keypadControl := RELATE KeypadControl (keypadController, keypadView);
keypadControl.ControllerParticipant.initializeTranslators ([keypadTranslator]);
keypadControl.buildButtons ([calculator], [calculatorConsistence]);

calculatorView := NEW (View);
calculatorController := NEW (Controller);
calculatorControl := RELATE ControlInteractor (calculatorController, calculatorView);
calculatorControl.appendSubInteractors (digitControl);
calculatorControl.appendSubInteractors (keypadControl);
calculatorControl.startUp ();

Figure 13: Instantiating calculator compositions

To reduce the code, we omitted the instantiation and relation of each Button. They are done in the procedure buildButtons ([calculator], [calculatorConsistence]). Note the similarity of instantiating and relating digit and keypad. One difference is the performer for their translator. The performer for digitTranslator is a digitView while the performer for keypadTranslator is a calculator. Another difference is the consistence. The consistence for digitTranslator is viewConsistence while the consistence for keypadTranslator is calculatorConsistence. Note that control passes its translator to its controller and translate passes its consistence to its performer. The calculatorControl sets up the view hierarchy using “appendSubInteractors” and start up using “startUp”.

3 Conclusion

This paper presents a unified approach to model and implement interactions as first class objects. During analysis and design, they are modeled using DynaSpecs. During implementation, they are coded with a new language construct called Compositions. Both DynaSpecs and Compositions provide a consistent support for object interactions within each phase of the OO lifecycle.

Making interactions first class citizens within OO systems requires support during all phases of software development. Our research has already developed a rich support environment for such interactions from analysis to implementation. Interactions are specified explicitly using DynaSpecs, a formal method for describing the dynamic model of object interactions. DynaSpecs model the object interactions using state transitions. The state transitions represent events of message calls that mirror the dynamic behavior of object interactions. The formal semantics of the model is based on the idea of a history which captures the sequence of operations within interactions along with the changes undergone by the participating objects. A major attraction of this model is that it is effective not only in the variable degree of abstraction of interactions among entities but also in the accurate representation of the application environments. This research defines the formal semantics for one and only one interaction environment. Allowing an application to have more than one interaction environments is appealing. Objects could move from one interaction environment to another within the same application. Each environment defining a context of interaction allows objects to have completely different sets of interactions in different environments within the same application.

The Composition construct clearly defines object interactions as first class values. Being first class values, interactors, through the composition construct and DynaSpecs, reduces complexity, loosens coupling, improves reusability of object interactions, and allows better abstraction of interactions. The main properties by interactors are

- definition of participants and their responsibilities,
- defining proper objects suitable to be participants through abstract procedures,
- restricted visibility to participant objects to only methods relevant to the interaction,
- reuse of interactions by plugging participants or inheriting compositions,
- providing a dynamic lifetime of an object interaction, and
- providing the necessary context to define the interactions.

The case study demonstrate how to capture object interactions using DynaSpecs and Compositions. The calculator provides an example of reusing the compositions. Prototype executable of both construct have been
implemented. The semantics of the prototypes are defined in denotational semantic. They depict the precise behavior of the construct and all its relevant context and scope. The one-to-one relationship between DynaSpecs developed during analysis/design and Compositions in the implementation provide a consistent approach. Thus, changes to the system can be made to DynaSpecs and translated to Compositions. This approach makes it easier to establish consistency between the analysis/design phase and the corresponding implementation and simplify reasoning and validating the modeled Object-Oriented software.

References

Abstract

C and C++ allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

In fact, function closures can be modeled directly in C++ by enclosing a function inside an object such that the local environment is captured by data members of the object. This idiom is described in advanced C++ texts and is used, for example, to implement callbacks.

The purpose of this paper is twofold: First, we demonstrate how this idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. A C++ implementation of the framework is available on request.

1 Introduction

The programming languages C [HS87] and C++ [ES90] allow passing functions as arguments to other functions in the form of function pointers. However, since function pointers can refer only to existing functions declared at global or file scope, these function arguments cannot capture local environments. This leads to the common misconception that C and C++ do not support function closures.

On the contrary, function closures can be modeled in C++ by enclosing a function inside an object such that the local environment or parts thereof are captured by data members of the object. This is possible because objects in C++ are essentially higher-order records, that is, records with fields that can contain not only values, but also functions [Red95].
The purpose of this paper is twofold: First, we demonstrate how the functoid idiom can be generalized to a type-safe framework of C++ class templates for higher-order functions that support composition and partial application. The framework could be translated to other object-oriented languages that support inheritance and genericity. Second, we explore the expressiveness of the framework and compare it with that of existing functional programming languages.

We show informally that there is a simple compositional translation from functional programs to the framework. We illustrate by means of various examples that object-oriented and functional idioms can coexist productively and can be used to enhance the functionality of common classes, for example, of nonlinear collections such as trees. To integrate the framework with C and C++ programs, we incorporate an existing mechanism to convert member functions back to nonmember functions. A C++ implementation of the framework is available on request.

In the remainder of this paper, Section 2 describes in detail the requirements, the implementation, and the structure of the functoid framework. Section 3 conducts a case study in which a typical functional program expressed within the framework. Section 4 explains how an existing conversion mechanism from C++ member functions to ordinary nonmember functions is incorporated in the framework. Section 5 concludes with an assessment of this work and a look at related and future work.

2 Functoids: An Abstraction of Functions

This section introduces the framework of functoids. In this framework, a functoid is an abstraction of the familiar concept "function".

Requirements

We first establish the requirements of the functoid abstraction. Our goal is to provide a type-safe abstraction, that is, there should be no need for type casts or untyped pointers at the user level. The abstraction should be provided in the form of C++ classes or class templates. We require that the abstraction supports the following essential operations performed on or by functions:

- **Application**
  Functoids can be applied to arguments. When a functoid is invoked by applying it to one or more arguments of appropriate types, the functoid returns a value of the appropriate result type.

- **Creation**
  Functoids can be created statically or dynamically. Upon creation, the functoid can capture and remember parts of the current environment.

- **Composition**
  Functoids can be composed with one another. When a functoid $f$ is composed with another functoid $g$, the result of the composition is a new functoid $h$. When $h$ is applied to an argument, it first obtains an intermediate result by applying $g$ to the argument and then returns as a final result the application of $f$ to the intermediate result.

- **Partial application**
  Functoids can be applied partially to fewer arguments than they actually accept. The result of partial application is a new functoid that accepts the remaining arguments. The conversion to a functoid that takes its arguments one at a time is known as "currying" in functional programming terminology.

- **Conversion**
  Functoids are equivalent to ordinary functions. An ordinary function can be converted to a functoid, and a functoid can be converted to an ordinary nonmember function when such a function is required, for example, as a callback function for an existing library. Conversion back to nonmember functions is difficult and will be addressed in Section 4.

- **Extension**
  Functoids provide extensible functionality. We can add application-specific operations to a
functoid besides the basic operations described above.

We implement our abstraction as a class template Fun that provides the interface to the abstraction functoid and is parameterized by the types of argument and result. The creation requirement will be handled by the constructors of this class, and the application requirement is captured by a function call operator of the appropriate type.

```cpp
template <class In, Out> class Fun {
public:
    Fun(FunImpl<In, Out> const &impl) : impl(impl) {}
    Fun(const Fun fun) : impl(fun.impl->copy()) {}
    ~Fun() { delete impl; }
    Out operator()(In arg) const
    { return (*impl)(arg); }
private:
    FunImpl<In, Out> const *impl;
};
```

The question arises how users of the functoid framework should incorporate their own functoid classes. The idea is that users derive concrete functoid classes from the class Fun, providing their own implementations of the function call operator. To make this approach work, the function call operator would have to be declared as virtual so that dynamic method selection is used, and functoids would always have to be passed and returned by pointer or reference [ES90]. On the other hand, memory management becomes an important issue when objects are not returned by value [Mey92]. What we want here is both call-by-value and dynamic method selection.

Fortunately, the envelope/letter idiom [Cop92], also known as the bridge pattern [GHJV93], gives us a way out of this dilemma. We apply this idiom to the framework as follows. We provide a class template called Fun to capture the interface of our abstraction. This is the envelope class, and functoids are passed and returned by value as instances of this class. We also provide an abstract class template called FunImpl for implementations of functoids. This class is an abstract letter class, and users of the framework provide their own functoid implementations by deriving from this class. The envelope class has a pointer to the letter class, and invocations of the function call operator in an envelope object are simply passed on to the letter object.

We first present the envelope class template Fun because it comes first conceptually, although it depends on the class template FunImpl. We provide two constructors, one to create a functoid from an existing functoid implementation, and a copy constructor that makes an explicit copy of the implementation of the functoid it copies. This is necessary so that no two functoid implementations are shared and the destructor can safely delete the corresponding implementation. Furthermore, we provide a function call operator that simply passes the function call on to the implementation, accessible via the pointer impl.

```cpp
template <class In, Class Out, Class Fun>
fun compose(f, g) = fn x => (g(f(x)))
```

The form "fn x => e" creates an anonymous function with argument x and body e. Thus the composition yields a new function with argument x and result f(g(x)). The composition is permitted only if the result type of g is compatible with the argument type of f. The new function has the same argument type as g and the same result type as f.

To avoid excess parameterization of the template Fun, we provide this functionality as a nonmember function that returns a functoid composed from the two functoid arguments. This resulting functoid is an instance of the class template Compose and holds the two functoids to be composed; the composition itself is carried out in the function call operator of this class. Both the class and the function templates have three type parameters for the argument, intermediate, and result types.

```cpp
template <class In, class Med, class Out> class Fun
```
Conversion from nonmember functions

The next requirement is conversion from an ordinary nonmember function to a functoid. The reverse direction is discussed below in Section 4. It is not hard to create a functoid from an ordinary function. Such a functoid can be implemented with a data member that points to the function and a function call operator that passes its argument on to the function pointer.

```
template <class In, class Out> class Global
{
public:
  typedef Out* FunPtr; // Fun:

  Global(FunPtr f) : theFun(f) { }
  virtual Out operator()(In arg) const
  { return theFun(arg); }

private:
  FunPtr theFun;
};
```

To enable automatic conversion from an ordinary function to a functoid, we add the following constructor to the class template Fun, where FunPtr is defined as in the class template Global.

```
Fun<In, Out>::Fun(FunPtr f)
  : impl(new Global<In, Out>(*f)) { }
```

Partial application

Another issue is how to deal with functoids that take more than one argument. In ML, a function that partially applies a function of two arguments to the first argument can be written as follows:

```
fun apply(f, x) = \( y \mapsto f(x, y) \)
```

The result of the partial application of \( f \) to the first argument \( x \) is a new function with a single argument \( y \) and result \( f \) applied to \( x \) and \( y \). The argument type of the new function is the type of the second argument of \( f \), and its result type is the result type of \( f \).

In the framework, the class template for functoids with multiple arguments would have to be parameterized by all argument types and the result type. Therefore the framework has to provide an envelope and a letter class for each number of arguments that could reasonably arise. If the maximum number of arguments is exceeded, a solution is to group several arguments in a single object. However, our partial application requirement can be satisfied only if the functoid accepts its arguments one-by-one. A better approach would thus be to automate the generation of the class templates depending on the maximum number of arguments desired in the application. The structure of the framework is sufficiently systematic to make this a feasible option.

We now illustrate partial evaluation for functoids of two arguments. First comes the abstract letter class Fun2Impl, followed by the corresponding envelope class Fun2. These classes differ from FunImpl and Fun in that they have two function call operators: one that takes two arguments instead of one, and one that takes a single argument and returns a new functoid. The second function call operator provides partial evaluation by applying the functoid to the first argument only.

```
template <class In1, class In2, class Out>
class Fun2
{
public:
  typedef Out* Fun2Ptr; // Fun2:

  Fun2(Fun2Ptr f)
    : impl(new Global2<In1, In2, Out>(*f)) { }

private:
  Fun2Ptr theFun;
};
```
To complete our implementation of partial evaluation, we must implement the function call operator that takes only one argument. This operator returns an instance of the class template Apply21, which keeps track of the first argument and the original functoid. When the function call operator of an instance of Apply21 is invoked with the second argument, the operator simply applies the original functoid to both arguments.

For additional flexibility in combining partial evaluation and composition, we also provide partial evaluation without application to any arguments. In ML, such a function is written as follows:

```
fun curry(f) = fn x => fn y => f(x,y)
```

This function converts its argument \( f \) to a new function that takes its arguments one after the other instead of both at the same time.

In the framework, the additional member function `curry1` in class `Fun2` converts a functoid of two arguments to a new functoid of the auxiliary class `Curry1`. The function call operator in the new functoid takes one argument (the first one) and returns a functoid that takes one argument (the second one) by invoking the partial function call operator in the original functoid on the first argument.

```
template <class In1, class In2, class Out>
    class Curry1 : public FunImpl<In1, Fun<In2, Out> >
    {
    public:
        Curry1(const Fun2<In1, In2, Out> & fun)
        : theFun(fun) {}
        virtual Fun<In2, Out> operator() (In1 arg1) const
        { return theFun(arg1); }
        virtual FunImpl<In1, Fun<In2, Out> > * copy() const
        { return new Curry1<In1, In2, Out> (*this); }
    }

private:
    const Fun2<In1, In2, Out> & theFun;
};
```

Adding methods to functoids

The last requirement addresses the extensibility of functoids. We will want to add application-specific member functions to the basic functionality provided by functoids. This can be done by deriving a class `UserFun` from the envelope class `Fun` and a class `UserImpl` from the abstract letter class `FunImpl`. Similarly to the function call operator, the additional member function `f` is implemented in `UserFun` as a wrapper that invokes the real one in `UserImpl`. We are facing a minor problem: not only is the pointer `impl` to the letter object private in class `Fun`, but it also is of class `FunImpl`, which does not have the new member function. We solve this problem by making `impl` protected in `Fun` and casting it to class `UserImpl` in the member function `f`. This cast is safe, since it is hidden from the user of the class.

The following example of a class `Cont` for continuations illustrates this requirement. Besides application to a consumer object of some other class `Consumer`, a continuation supports the method `done` to check whether the continuation has finished. We therefore make the envelope class `Cont` a subclass of `Fun` and the associated letter class `ContImpl` a subclass of `FunImpl`, each instantiat
ed with appropriate argument and result types. We extend the functionality of Fun and FunImpl by adding the member function done in the subclasses. The member function done in class Cont first casts the pointer impl to class ContImpl and then invokes the member function done in class ContImpl.

```cpp
class Cont
  : public Fun<Const Consumers>, bool
{
  public:
    bool done() const
    { return ((ContImpl *)impl)->done(); } 
};
```

The structure of the functoid framework

Since the framework uses the envelope/letter idiom, it consists of separate abstraction and implementation class hierarchies. There are sub-frame-
works for functoids of zero, one, and more arguments. We first describe the case of a single argument. The framework provides an abstraction class Fun, which is the class for functoids in user programs. The framework also provides an abstract implementation class FunImpl, from which users derive their own implementations of functoids by overriding the function call operator. Users of the framework derive classes UserFun from Fun to add constructors that instantiate the user-defined functoid implementation classes UserImplA, UserImplB, and so on, derived from FunImpl. The framework predefines several functoid implementation classes: Global implements wrappers around nonmember functions; Compose implements functoids resulting from composition; Apply2, Apply3, and so on, implement functoids resulting from partial application of functoids with more than one argument such that the resulting functoid takes the remaining single argument; finally, Curry1 implements functoids resulting from "currying" functoids with more than one argument with respect to the first argument.

The sub-frameworks for two or more arguments have a similar structure. In the case of K arguments, there is an abstraction class FunK and an abstract implementation class FunKImpl. As in the case of a single argument, users derive from both framework classes. Again, there are various predefined functoid implementation classes: GlobalK implements wrappers around nonmember functions of K arguments. For N > K, ApplyNK implements functoids resulting from partial application of a functoid of N arguments to N - K arguments. Since the resulting functoids take the remaining K arguments, the class ApplyNK is a subclass of FunKImpl. For K ≥ 2 the class CurryK describes curried functoids that take K arguments one at a time. The actual currying is carried out by the corresponding member function curryK in the class FunN, where N > K. As function composition is defined only for functions of one argument, we do not consider it for K ≠ 1.

We have not yet addressed the case of functions of zero arguments. We could treat functions without arguments as functions with one dummy argument of an enumerated type uni with a single value, but this approach would cause difficulties when creating wrappers for global functions with no arguments. We therefore provide a separate, simple sub-framework for this case consisting of only three class templates, Fun0, Fun0Impl, and Global0, whose roles are similar to the corresponding classes in the other cases. These class templates are parameterized only by the result type of the function.

We illustrate the structure of the framework in the notation used by Gamma et al. [GHJV93], which is an extension of the OMT (Object Modeling Technique) notation [R+91]. The framework for zero arguments is shown in Figure 1. The framework for one argument is shown in Figure 2. The framework for two or more arguments is shown in Figure 3, where K is the number of arguments. For simplicity, the classes CurryK are not shown.
3 Case Study: The Same-Fringe Problem

The purpose of this section is threefold. First, it demonstrates how functional programming styles can be incorporated directly in C++ programs. Second, it serves as a case study that shows the practical usefulness of our system. Third, it disproves claims that this style of programming is not supported by C++ [Bak93].

The same-fringe problem

The fringe of a finite tree is the enumeration of its leaves in left-to-right order. The same-fringe problem is the problem of deciding whether two finite trees have the same fringe. In practice, this problem occurs when comparing for equality two trees that store data only in their leaves. A brute-force solution to this problem would involve generating the fringe of each tree as a list and then comparing the two lists for equality. This shortcoming of this solution is that it goes through considerable work to construct the entire lists although there might be a mismatch at the beginning of the lists. We could do slightly better by constructing the fringe of only one tree and iterating through the other tree.

A far better solution to the same-fringe problem is to compare the first leaf of each tree and continue only if they match. Such a solution could be expressed in terms of coroutines, which would need unbounded storage to keep track of the current path in the tree. These coroutines could be modeled by external iterators in C++. The drawback of this approach is that the tree traversal has to be made explicit instead of implicit and recursive.

A solution in a functional language

In a functional language, this problem could be solved elegantly in terms of lazy streams [FW76]. A lazy stream is a recursive data structure that is either an empty stream or a data item paired with a function that evaluates to another stream when invoked. This technique allows us to delay the generation of the entire fringe: we seemingly construct the fringe like an ordinary list, but the actual construction is performed on demand. In the functional language ML [MTH90], a data structure for lazy streams could be defined as follows. The two cases are called Nil and Cons in analogy to ordinary lists in functional languages. In ML, functions without arguments take a single argument of type unit.

```
datatype 'a Stream =
    Nil
  | Cons of 'a * ('a Stream)
```

We now generate the fringe of a tree recursively. If the tree is a leaf, then the fringe is simply the pair of
the item and a function evaluating to an empty stream. Otherwise the tree is a node, and the fringe is the concatenation of the fringes of the subtrees. We actually concatenate two functions that evaluate to fringes when invoked to delay generating the entire fringes until requested. ML uses pattern matching to examine the structure of function arguments. The form "fn () => expr" is used to create an anonymous function closure on the fly. The comments identify the three different cases of anonymous functions we are creating.

```cpp
fun fringe (Leaf x) =
    Cons(x, fn () => Nil) (* Case 1 *)
  | fringe (Node(l,r)) =
    concat (fn () => fringe l) (fn () => fringe r) (* Case 2 *)
```

The concatenation of the two functions follows. If the first function evaluates to an empty stream, the fringe is simply the invocation of the second function. Otherwise the fringe is the first item of the first fringe paired with the concatenation of the rest of the first fringe and the second fringe:

```cpp
fun concat f g =
    case f () of
    Nil => g ()  (* Case 1 *)
    Cons(x, h) =>
        Cons(x, f => concat h g)  (* Case 2 *)
```

The next job is to compare two lazy streams for equality. If both are empty, then they are equal. Otherwise, their first items have to match and the remaining streams have to be equal. In all other cases, the two streams are not equal. The following recursive function captures this notion of equality:

```cpp
fun eq Nil Nil = true
  | eq (Cons(v1, f1)) (Cons(v2, f2)) =
    (v1 = v2) andalso eq (f1 ()) (f2 ())
  | eq s1 s2 = false
```

Now we are ready to define the same fringe function for two trees:

```cpp
fun samefringe t1 t2 =
    eq (fringe t1) (fringe t2)
```

For example, among the following three trees, t1 and t2 have the same fringe, although they do not have the same shape, whereas t0 has a different fringe:

```cpp
val t0 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 7))
val t1 = Node(Node(Leaf 3, Leaf 4),
              Node(Leaf 5, Leaf 6))
val t2 = Node(Node(Node(Leaf 3, Leaf 4),
              Leaf 5),
              Leaf 6)
```

### Translating the solution to C++

We show how to translate the ML solution directly into C++ using the functoid framework. For the sake of simplicity, we deal only with integer items, but we could also have used templates for the various classes. Assume we are given a tree class with the following public member functions:

```cpp
class Tree {
public:
    int label() const;
    bool isleaf() const;
    const Tree& left() const;
    const Tree& right() const;
    ...
};
```

Our first task is to express lazy streams in C++. One approach to representing a recursive data structure in C++ is as a tagged union, using an enumerated tag field to indicate which of the cases an object belongs to and providing data members for all components of the data structure. We choose a better, more object-oriented approach that models each case of the data structure as a different subclass of a class for the data structure itself. To facilitate passing streams by value, we again employ the envelope/letter idiom. The class Stream becomes the envelope class, and we have an abstract letter class StreamImpl with concrete subclasses NilStream and ConsStream for the two cases of the data structure. We first present the class Stream.

```cpp
class Stream {
private:
    StreamImpl* theStream;
}
```

We need constructors for both cases, a copy constructor, and a destructor. The constructors take as arguments the components of the corresponding cases of the data structure. We assume a forward declaration of the class Delay for functions evaluating to streams.

```cpp
class Stream {
public:
    Stream();
    Stream(int hd, const Delay& tl);
    Stream(const Stream& s) :
        theStream(s.theStream->copy()){};
    ~Stream() { delete theStream; }
```

Now we need to design an interface for the stream class that allows us to distinguish between the two alternatives and to extract the components in the second case. The function empty tells us whether a stream is empty; in the nonempty case, head extracts the item, and tail extracts the function.

```cpp
bool empty() const
```
We are going to implement the three cases as subclasses of Fun0Impl. We again omit the copy member function. Case 1 is a function of the form “fn () => Nil” evaluating to an empty stream. It is represented by the following functoid:

```cpp
class EmptyDelay : public Fun0Impl<Stream>
{
public:
    virtual Stream operator()() const
    { return Stream(); }
};
```

We now define the two subclasses corresponding to the two cases of the data structure. These subclasses implement the pure virtual member functions defined in class StreamImpl. For brevity, we omit the copy member function, which simply duplicates the receiver. A NilStream is always empty and does not have a defined head or tail.

```cpp
class NilStream : public StreamImpl
{
public:
    virtual StreamImpl() const
    { return StreamImpl(); }
    virtual bool empty() const { return true; }
    virtual int head() const { return 0; }
    virtual const Delay& tail() const { return nil(); }
};
```

A ConsStream is never empty. The head and tail member functions return the corresponding data members, a number and a function, respectively.

```cpp
class ConsStream : public StreamImpl
{
public:
    ConsStream(int x, const Delay& t) : hd(x), tl(t) {}
    virtual bool empty() const { return false; }
    virtual int head() const { return hd; }
    virtual const Delay& tail() const { return tl; }
};
```

Now we must define the class Delay, which in turn depends on the stream class. We integrate this class in the functoid framework. The class Delay is a subclass of an appropriate instance of the class template Fun0, and the implementations of Delay will be subclasses of instances of the class template Fun0Impl. The purpose of introducing the class Delay is to capture the mutual dependency with the class Stream and to introduce appropriate constructors for each implementation of this class that we want to create. In the ML solution above we identified three cases of anonymous function closures that correspond to three implementations of the class Delay.

```cpp
class Delay : public Fun0<Stream>
{
public:
    Delay();
    Delay(const Delay& f, const Delay& g);
    Delay(const Tree<int>& t);
};
```

Next, we present the abstract letter class StreamImpl. Its pure virtual member functions correspond to the member function of class Stream.

```cpp
class StreamImpl
{
public:
    virtual ~StreamImpl() {} 
    virtual bool empty() const { return true; }
    virtual int head() const { return 0; }
    virtual const Delay& tail() const { return nil(); }
    virtual StreamImpl* copy() const = 0;
};
```

We now define the two subclasses corresponding to the two cases of the data structure. These subclasses implement the pure virtual member functions defined in class StreamImpl. For brevity, we omit the copy member function, which simply duplicates the receiver. A NilStream is always empty and does not have a defined head or tail.

```cpp
class NilStream : public StreamImpl
{
public:
    virtual bool empty() const { return true; }
    virtual int head() const { return 0; }
    virtual const Delay& tail() const { abort(); }
};
```

Case 2 is a function of the form “fn () => fringe t” evaluating to the fringe of a tree. The corresponding functoid FringeDelay stores the tree t and invokes the function fringe,
a direct translation of the corresponding ML function.

```cpp
Stream fringe(const Tree& t)
{
    if (t.isleaf())
        return Stream(t.label(), Delay{1});
    else
        return concat(Delay{t.left()}, Delay{t.right()});
}
```

```cpp
class FringeDelay : public FunImpl::Stream {
public:
    FringeDelay(const Tree& t) :
        tree(t) {}
    virtual Stream operator()() const
    { return fringe(tree); }
private:
    const Tree tree;
};
```

Case 3 is a function of the form “fn() =⇒ concat gh”. The associated functoid ConcataDelay stores the two functions evaluating to the streams to be concatenated and invokes the function concat, again a translation of the corresponding ML function.

```cpp
Stream concat(const Delay& f,
               const Delay& g)
{
    Stream s = f();
    if (s.empty()) return g();
    else return Stream(s.head(),
                        Delay{s.tail(), g});
}
```

```cpp
class ConcatDelay : public FunImpl::Stream {
public:
    ConcatDelay(const Delay& f,
                const Delay& g) :
        fdelay(f), gdelay(g) {}
    virtual Stream operator()() const
    { return concat(fdelay, gdelay); }
private:
    Delay fdelay, gdelay;
};
```

Finally, we give the implementations of the three constructors for the class Delay. Each constructor creates an instance of the corresponding implementation class of the class Delay.

```cpp
Delay::Delay()
    : Fun<Stream>({new EmptDelay})
{
}
Delay::Delay(const Delay& f,
             const Delay& g)
    : Fun<Stream>({new CombineDelay(f, g)})
{
}
Delay::Delay(const Tree& t)
    : Fun<Unit, Stream>({new FringeDelay(t)})
{
}
```

We can now determine whether two trees have the same fringe by generating the corresponding streams and checking them for equality.

```cpp
bool samefringe(const Tree& t1,
                 const Tree& t2)
{ return fringe(t1) == fringe(t2); }
```

We extend the tree class in two ways. If we define equality of trees as having the same fringe, we can add the following equality operator to the class Tree:

```cpp
bool Tree::operator==(const Tree& t) const
{ return fringe(*this) == fringe(t); }
```

Furthermore, we can enhance the class Tree with an external (active) iterator class that traverses the fringe of the tree. This class TreeIterator enables us to define more than one iterator on the same tree.

```cpp
class TreeIterator {
public:
    TreeIterator(const Tree& t)
        : theTree(t) { restart(); }
    operator bool() const
    { return !theFringe.empty(); }
    int current() const
    { return theFringe.head(); }
    void next() {
        if (!theFringe.empty())
            theFringe = theFringe.tail();
    }
    void restart() {
        theFringe = fringe(theTree);
    }
private:
    const Tree& theTree;
    Stream theFringe;
};
```

The next function uses the assignment operator for the class Stream, which we define using the copy function.

```cpp
Stream& Stream::operator=(const Stream& s)
{
    if (this != &s)
    {
        delete theStream;
        theStream = s.theStream->copy();
    }
    return *this;
}
```

4 Converting Functoids to Ordinary Functions

The framework presented in Section 2 falls short of the requirement that functoids be convertible to ordinary nonmember functions. This shortcoming
The heterogeneity problem

The fundamental difference between member functions and nonmember functions was recognized by Young [You92] and is called the heterogeneity problem by Dami [Dam94]. Technically, a call to a nonmember function requires a stack pointer to store the actual arguments and the address of the function to be called. A member function invocation, on the other hand, requires a stack pointer to store the arguments, the address of the member function, and the address of the receiver. This fundamental difference in the calling mechanism makes it impossible to use a member function where a nonmember function is expected, for example, as a callback from an existing class library.

The proposed solutions [Fek91, You92, CL95] require that the programmer writes a nonmember function that explicitly invokes the C++ member function from a specific receiver. This solution is generally not very good because the programmer has to write a wrapper for every combination of a member function and a receiver to be used as a callback. More seriously, this solution does not work at all for the framework because we create functoids on the fly and thus cannot anticipate what wrappers to provide.

The solution using partial binding

Rescue comes in the form of a solution proposed and implemented by Dami [Dam94], which addresses a more general partial binding problem. In this solution, when we perform a partial binding, we create a data structure that stores the address of the function, the arguments, and code to complete the bindings and invoke the function later. This approach is compiler- and machine-dependent; it currently works with the GNU CC compiler [Sta94] on NeXT and Sparc architectures, but could be ported to other languages, compilers, or architectures. A similar mechanism that maps Scheme closure objects to C functions is described by Rose and Muller [RM92]. The ObjectKit system for Macintosh Smalltalk allows passing Smalltalk objects, including closures, to C functions [RM92, quoting P. Deutsch].

From the programmer's perspective, Dami's mechanism consists of the function curry, whose arguments are a pointer to the memory where the data structure should be allocated, the function to be invoked, the total number of arguments, and the number of arguments supplied here, and those arguments.

```c
typedef void (*)(* anyFunc)(
    void* mem, anyFunc f,
    int nargs, int cargs, ...);
```

This mechanism extends to object-oriented languages in the sense that the receiver of a message is an (implicit) first argument to the method invoked. We can thus convert a member function to a nonmember function by partial application to the receiver this. While the mechanism itself is not type-safe, we safely hide it inside the framework, and the user only sees it as a type conversion operator of functoids back to nonmember functions. We describe how the mechanism is implemented for functoids with a single argument; the implementation for functoids with more arguments is analogous.

The class template `FunImpl` gets an additional member function that performs the conversion of its function call operator to a nonmember function.

```c
typedef Out (*)(FunPtr)(In);

virtual FunPtr FunImpl::Out::cfun() const {
    return FunPtr(&funitty[0],
        anyFunc(this->operator()),
            3, 1, this);
}
```

The class template `Fun` is extended by a type conversion operator that invokes `cfun` on the implementation of the functoid. To make sure that the conversion is executed only once, we store the resulting function pointer in an additional data member `fun` of `Fun`, which the constructors initialize to the null pointer. The associated data must be deallocated using the `free` function when the functoid is destructed. The new members of `Fun` are as follows.

```c
template <class In, class Out, class Fun
{
public:
    typedef Out (*)(FunPtr)(In);
    ...
    virtual FunPtr Fun( )
    {{ delete impl; if (fun) free(fun); } operator FunPtr() const
        if (fun == NULL)
            ((Fun::In,Out::* this)->fun =
            impl->cfun()); return fun;
    }
    ...
private:
    FunPtr fun;
};

Now our conversion requirement is satisfied both ways, and functoids and nonmember functions are indistinguishable to the user. The example at the end of Section 2 illustrates this feature.

5 Conclusion

We have presented a type-safe generalized framework that supports higher-order functional programming styles within C++ programs. The framework is implemented entirely in the form of C++ class templates, except for a compiler- and machine-dependent mechanism for converting member functions to nonmember functions [Dam94]. The framework could be translated to other object-oriented languages that support both inheritance and genericity.

The main issues in the assessment of our framework are expressiveness and efficiency. To address the first issue, we compare our framework to existing functional programming languages.

It is a fundamental limitation of most class-based object-oriented languages that each distinct behavior must be given a class name [Ros95a]. Consequently, our framework does not provide a mechanism for creating anonymous function closures on the fly. This is in contrast to functional languages, in which anonymous closures are routinely passed to and returned from functions. Rose [Ros95b] describes an extension of C++ with parameterless anonymous functions called "thunks; a thunk can be converted to a parameterized function by specifying which variables used in the body of the thunk are to be treated as parameters.

Another limitation of the functoid idiom in general, not just of the functoid framework, is that the programmer must establish and maintain an explicit correspondence between variables used in the body of the closure and instance variables of the functoid. By contrast, functional and other languages with block structure and nested functions, such as Algol or Pascal, automatically capture all local variables that are used in the closure. Breuel [Bre88] solves this shortcoming in C and C++ by allowing functions to be nested. Thunks [Ros95b] provide a solution as well.

Another drawback of the functoid framework stems from the way type information is required in instantiations of C++ class templates. While the examples presented in this paper do not require lengthy type parameters, the type information required in more complex applications of the framework is likely to get out of hand, especially when higher numbers of arguments are involved. Dami [Dam95] suggests extending the compiler to keep track of the required type parameters automatically.

There are several sources of inefficiency in the framework as compared to typical implementations of functional languages. First, we use call-by-value to facilitate memory management. This approach requires a considerable amount of copying, depending on the size of the functoid implementations involved. The problem could be addressed by improving the memory management strategy, for example, by using garbage collection for functoids. Memory management could still be hidden from the user by overloading the new and delete operators for functoids. Second, the structure of the framework requires a virtual function call operator that is overridden in the user classes to allow dynamic selection of the appropriate functoid implementation. This problem is inherent in the design of the framework and has no simple solution. Third, unlike in functional languages, function closures are controlled by the programmer instead of the compiler. This precludes the sort of optimizations a compiler of a functional language would apply.

Other approaches that combine functional languages and C++ include an interpreter accessible within C++ [Kla93] and an interpreter written in C++ [RK88]. A detailed comparison with our work would go beyond the scope of this paper. While the translation outlined informally in Section 3 is not suitable at present as an efficient implementation of functional languages, the paper demonstrates that the framework provides access to various functional idioms within object-oriented languages.

Acknowledgments

Special thanks go to Gerald Baumgartner for his careful review of the initial version of this paper, to Laurent Dami for helpful comments and for providing his Curry code, to Thomas Kühne for stimulating email discussions on this and related material, and to John Rose for valuable comments and for sharing his insights on thunks with me.

References


[Bak93] H. Baker. Iterators: Signs of weakness in


Abstract

Lazy Functional Programming for Full-Text Information Retrieval

Donald A. Ziff
Dept. of Computer Science, University of Chicago

Very few applications have been written in lazy functional programming languages, and hardly any, except the compilers for those languages, are reported in academic literature. It is by no means a settled question whether “real” applications can be written in a lazy functional programming language. This is in part because these languages typically offer little or no support for interoperability, combining functional programs with programs or systems written in other languages.

This work describes an experimental textual information retrieval system, Philo/Philis 2, in which lazy functional programming is combined with a varied set of other applications techniques, from components written in other languages to off-the-shelf subsystems. Functional programming interfacing techniques, procedural and data abstraction, were used throughout the system, and greatly smoothed the overall implementation process.

In the retrieval engine implementation, called the Funser, for Functional Server, lazy functional programming is shown to be a powerful and elegant means of accomplishing several desirable concrete goals: delivering initial results promptly, using space economically, and avoiding unnecessary I/O. An innovative module in this system, the TOMS, Textual Object Management System, is designed as an abstract datatype for structured text; this design permitted the retrieval system written as its client to be, to a large extent, database independent. This work also features a new formal model of word-based textual information retrieval, the Matrix model.

Philo/Philis 2 is used by the ARTFL project, American and French Research on the Treasury of the French Language, as the basis of their on-line retrieval service.