The Interaction Between Static Typing and Frameworks

Gail C. Murphy and David Notkin

Technical Report 93-09-02
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle WA, USA 98195
{gmurphy,notkin}@cs.washington.edu

Abstract

Frameworks capture the commonalities in design and implementation between a family of related applications and are typically expressed in an object-oriented language. Software engineers use frameworks to reduce the cost of building complex applications. This paper characterizes the operations of instantiation, refinement and extension used to build applications from frameworks and explores how these operations are supported by static typing policies of common object-oriented languages. We found that both conservative contravariant and covariant static typing policies were effective in supporting the operations of framework instantiation and framework extension. However, both policies were ineffective at supporting the operation of framework refinement. Although it does not support the refinement operation itself, covariance is sufficiently expressive to support the instantiation of a properly refined framework. This result illustrates how programming languages can at times complicate rather than simplify the engineering of software.

1 Introduction

Frameworks [Deu89, JF88, NGT92] are one of several approaches to reducing the cost of building complex applications by exploiting commonalities between related applications. Several properties distinguish frameworks from the other approaches. First, in contrast to software architectures [Sha90], which primarily focus on the common aspects of the specification of the members of a family, frameworks capture key aspects of the design and implementation of the members. Second, in contrast to application generators, which define a special-purpose language for specifying the details of the desired family member, frameworks embed the shared design and implementation in a general purpose — and in practice, object-oriented — language that provides an engine for realizing applications. Section 4 compares frameworks to these and other approaches to easing application development.

Frameworks are found in many application domains. Most commonly, frameworks have been used to ease the development of graphical applications. For example, MVC [Deu89], InterViews [LVC89], ET++ [WGM88] and Gina [SB90] are all user interface frameworks. Frameworks are also found in domains such as operating systems (Choices [CIM92]), document preparation (VAMP [FM89]) and financial management (ET++SwapsManager [BE93]).

Since frameworks provide partial designs and implementations, software engineers must define application-specific code to instantiate a framework into an application. A common problem is that many errors made during instantiation appear as run-time errors during application execution rather than when the instantiation operation is applied.

This research was funded in part by NSF Grants CCR-9113367 and CCR-8858804 and SRA, Tokyo Japan.

A similar problem arises with two other framework operations that we have characterized. The refinement operation takes a framework and specializes it into another framework that is intended to simplify the construction of a related but more specific class of applications. The extension operation takes a framework and produces another framework that is intended to simplify the construction of a broader class of applications. Errors made when applying refinement or extension operations are even harder to find than for instantiation. This is because the refinement and extension operations produce non-executable frameworks rather than applications. Examples of these operations and the instantiation operation are given in Section 2.

In this paper, we explore the use of static typing as found in production object-oriented languages as a mechanism for maintaining invariants on the framework's structure and behaviour across framework operations. Static typing permits the integrity of the application or framework to be checked after each framework operation. One reason that static typing might help is that the framework operations of instantiation, refinement, and extension are analogous to the object-oriented operations of instantiation and inheritance. Another reason is that static typing naturally connects design (as interfaces) with implementation (as code bodies); this connection is essential for retaining the synergy that successful frameworks have in simultaneously capturing both design and implementation commonalities.

To study whether static typing helps with frameworks in practice, we took a basic framework — the heart of the model-view part of the well-known Smalltalk-80 Model-View-Controller (MVC) — and considered straightforward designs and realizations using two common typing policies, covariance (as realized in Eiffel [Mey88]), and "conservative" contravariance (as realized in Modula-3 [CDJ+89]).

The results, as detailed in Section 3, are summarized as follows:

- the instantiation and extension operations are supported by both of the typing policies,
- refinement is not supported by either of the typing policies, and
- instantiation of a properly refined framework — where the structural and behavioural relations of the source framework were maintained across the refinement operation — is supported by covariance.

Even if some typing policies help ensure integrity to some degree, no single typing policy seems generally supportive of framework operations. In the summary (Section 5), we discuss the use of tools to augment the benefits of current static typing policies to further ease framework use.

2 Frameworks

The most common frameworks are those supporting the construction of user interfaces. There is significant similarity in the construction of each of these user interface frameworks. We have defined a framework that captures the essential similarities of these frameworks. The definition of this framework allows us to experiment with different typing policies, as realized in different languages, without incurring the cost of rewriting the much larger frameworks.

Our sample framework is best explained in terms of the Model-View (MV) part of Smalltalk's MVC. A model in our MV framework represents the underlying data of the application that will, in some form, be displayed and manipulated through the user interface. The views represent one or more projections of the model data onto windows of the user interface. When the model data is changed, the views must be updated. In this framework, direct changes to the views cannot be made; instead such changes are written in terms of changes to the model.

Typically, this framework is represented in an object-oriented language by two classes, one that defines the model and another that defines the views. The Model class has two key methods: `registerView`, which takes a view as a parameter and registers interest in the model on behalf of that view; and `changed`, which announces an event that is delivered to all registered views when the model is modified. The View class has a single method, `update`, which is called for registered views whenever the `changed` method for the model is invoked.

2.1 Structural and Behavioural Relationships

The design of a framework like MV is captured not only in its class definitions, but also in the set of structural and behavioural relationships among its classes. Structural relationships specify the number of instances of a framework class and the interconnections between instances that must exist at run-time. The structural relationships are those that might commonly be represented on an entity-relationship diagram [Che76]. Some aspects of structural relationships, like the interconnections, may be realized in the framework implementation as messages. Other aspects, like the number of instances and the cardinality of the interconnections, cannot be explicitly represented within the implementation. The structural relationships of the MV framework include the requirements that exactly one instance of the Model and at least one instance of the View class must exist, and that one or more views must be registered with a model instance through the `registerView` method.

The messages of the framework classes that provide access to the function of the framework define the behavioural relationships. The behavioural relationships include the data and control flow between the classes (sequences of messages and information passed in each message). For instance, the invocation of the `update` method in a registered view as a result of sending a `changed` message to a model represents a behavioural relationship.

2.2 Framework Operations

A software engineer takes advantage of a framework by applying the operations of instantiation, refinement and extension.

The instantiation operation is used by a software engineer to produce an application from a framework. The application instantiated from the framework may be used stand-alone or composed within a larger application. Most frameworks contain abstract classes that defer details of the implementation to individual applications. An important activity of instantiation is completing the implementation of the framework by subclassing the abstract classes. During instantiation, a software engineer must also ensure that the structural and behavioural requirements of the framework are met by instantiating objects from classes, and connecting the instances of the framework classes in the correct numbers. For example, the MV framework is instantiated by: subclassing the abstract View class to provide an implementation for the `update` method; creating at exactly one instance of the Model class and at least one instance of the View class; and registering the View instance with the Model instance.

The refinement operation is used to make a framework more specific to a domain. The activities of this operation include subclassing existing framework classes, adding new classes, extending existing structural and behavioural relationships, and adding new structural and behavioural relationships. Existing structural and behavioural relationships are extended by adding new message sends between the classes and by specializing the data flows of the original framework. In particular, data passed as arguments to messages may be refined. For example, the MV framework may be refined to support the display of graphs by: subclassing

Model to a Graph class which maintains the coordinates of the data points; subclassing View to a GraphView class; extending the `update` method on GraphView to accept as an argument the Graph instance that has changed, and implementing the `update` method for GraphView to query the Graph instance for the data to present. The refined framework is considered properly refined if the structural and behavioural relations are maintained and specialized appropriately across the refinement operation.

The extension operation is applied to make a framework useful in a more general domain. The activities of this operation include subclassing existing framework classes and adding new structural and behavioural relationships. For example, the MV framework may be extended to support the changing of the model's application data through manipulation of a view by: subclassing View to a WritableView class that adds `registerModel` and `changed` methods to those inherited from View; and subclassing Model to an ExtendedModel class that adds an `update` method to those inherited from Model.

The framework operations are generally applied in combination. Figure 1 shows how the three operations may be used to build a user interface for a graphing package that permits the display of a graph as a plot in a window and the manipulation of the graph data through that display. This is accomplished by applying the extension operation to our MV framework to provide a framework that supports modification of the Model through the View — a writable-view MV framework. The ExtendedModel class is a subclass of Model, and the WritableView class is a subclass of View. The refinement operation is then applied to the writable-view MV framework to produce a graph framework that supports the storage of coordinates on a Graph class (a subclass of ExtendedModel), and the viewing and modification of a plot of the data on a GraphView class (a subclass of WritableView). Finally, the refined framework is instantiated into the desired application by creating and connecting an instance of Graph with instances of GraphView.

When applying the refinement and extension operations, we also need to understand the effect of the operation on the relationship between the source and target frameworks. In particular, we are interested in the how the operations effect the substitutability of an instantiation of the target framework for an instantiation of the source framework. This is because applications are sometimes composed of multiple instantiated frameworks. For example, a personnel system may be composed of an instantiated user interface framework and an instantiated database framework. If the user interface framework is refined or extended, we want to not only maintain the structural and behavioural relationships in the target framework, but also assure substitutability of the refined framework for the source framework in the personnel system.

3 Typing

In this section, we explore the use of static typing in common object-oriented languages to maintain the structural and behavioural invariants during framework operations. The ability of the type system to check substitutability of a target framework for a source framework during refinement and extension operations is also considered. Other approaches to supporting the use of frameworks are discussed in Section 4.1.

The constraints we would like to enforce with static typing depend on the framework operation being performed. In the case of instantiation, static type annotations can specify the types participating in a structural relationship and some of the data participating in a behavioural relationship. For example, a static type system can be used to ensure that only instances of type View are registered as dependents with a Model, a structural relationship. A static type system can also be used to ensure that the `update` method[1] on the type

[1] In this section, we use the term *method* to refer to operations on a type to avoid confusion with the framework operations.

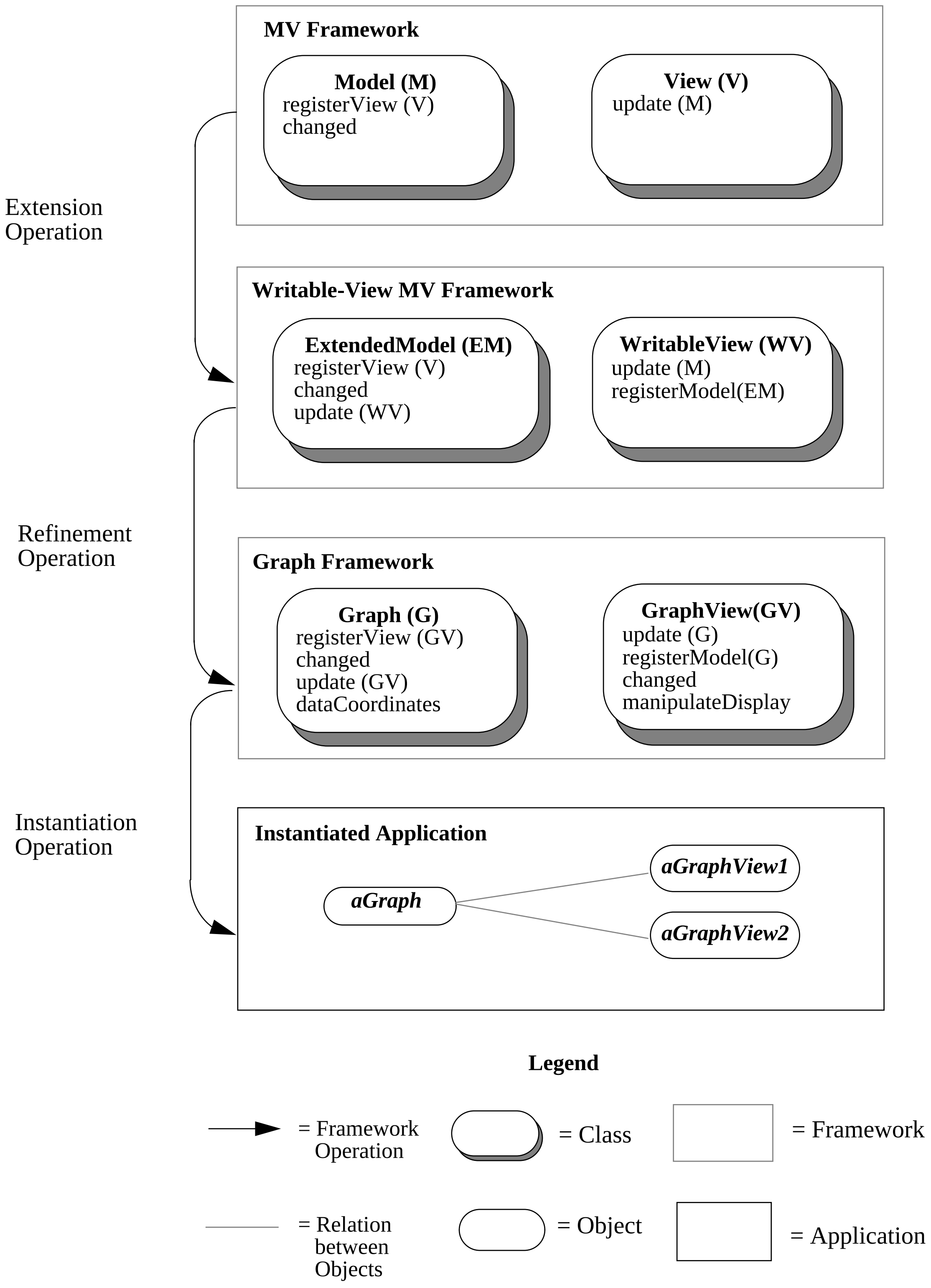


Figure 1: Combining Framework Operations

View receives only instances of the type Model as the parameter. This captures part of a behavioural relationship between a Model and a View.

The operations of refinement and extension place different requirements on a static type system. Unlike instantiation, these operations map a source framework to a target framework. The static type system can help to maintain the structural relationships from the source framework to the target framework. This includes ensuring that instantiation constraints in the source framework are refined consistently. For example, if we refine a Model type into a Drawing type that has specialized information about handling graphical figures, and we refine the View type into a DrawingView type that is able to access the specialized Drawing information, the refined instantiation constraints must ensure instances of the DrawingView type, and not instances of the View type, are registered with a Drawing. Since the source and target frameworks co-exist within the same library and may be instantiated within the same application, it is important that the static type system not only capture the constraints on the target framework, but also maintain the original instantiation constraints on the source framework.

3.1 Covariant and Conservative Contravariant Typing Policies

Two static typing policies are found in production object-oriented languages: conservative contravariance and covariance. The two typing policies differ in how the types of formal arguments to methods found in both a type and its subtype are related [Coo89]. Consider a type `Person` and a subtype `Retiree` (of `Person`) with a common method, `setAge` (this example is based on [WZ88]):

```
type Person                                 subtype Retiree of Person
   setAge (Integer (0..120))                   setAge (Integer (0..120))
```

In a conservative contravariant typing policy, the type of a formal argument for a method found in both the subtype and the type must be the same as the type of the argument specified for the method in the type.[2] A conservative contravariant typing policy ensures that instances of the subtype can be substituted for instances of its type.

In a covariant typing policy, the type of a formal argument for a method in the subtype may be a subtype of the type specified for the argument in the type. In this case, we may express:

```
type Person                                 subtype Retiree of Person
   setAge (Integer (0..120))                   setAge (Integer (65..120))
```

A covariant type policy is useful for expressing a frequent form of refinement between types where the subtype can accept more specific information through its method arguments than its type. This gain in expressiveness with a covariant type policy is offset by the loss of substitutability. As Wegner and Zdonik show:

```
p: Person;
r: Retiree;

p := r;
p.setAge (40);     --> run-time error
```

[2] A regular contravariant type policy permits the type of the formal argument to be a supertype and does not force it to be the same. A conservative contravariant type policy is, in practice, sufficient for developing applications as it is seldom that less information from an argument is accessed by the method in the subtype than in the type.

When the instance of subtype `Retiree` is assigned to the variable of the type `Person`, a value of 40 may be passed to the `setAge` method. Although a value of 40 is acceptable to a variable of type `Person`, it is not acceptable to a `Retiree`, the actual type of the variable `p` after the assignment of `r` to `p`. The result is an error at run-time.

Production-oriented statically-typed object-oriented languages are available that are representative of each of the typing policies. Modula-3, for example, uses a conservative contravariant typing policy [CDJ+89]. The Eiffel language uses a covariant type policy. To address type safety problems in earlier versions of the language, an extra level of checking, called system validity checking, has been designed for Eiffel Version 3 [Mey92]. System validity checking identifies potential compromises to type safety by considering, at compile-time, the set of types each expression within the system may potentially assume at run-time. This is accomplished through the determination of dynamic type sets for all expressions within the implementation. Since flow analysis is not used to determine an expression's dynamic type set, the approach may reject some systems that are type safe. No formal proofs are available showing that all type unsafe programs do in fact fail system validity checking. As compilers are only beginning to appear that support some level of system validity checking, there is little practical experience with this approach.

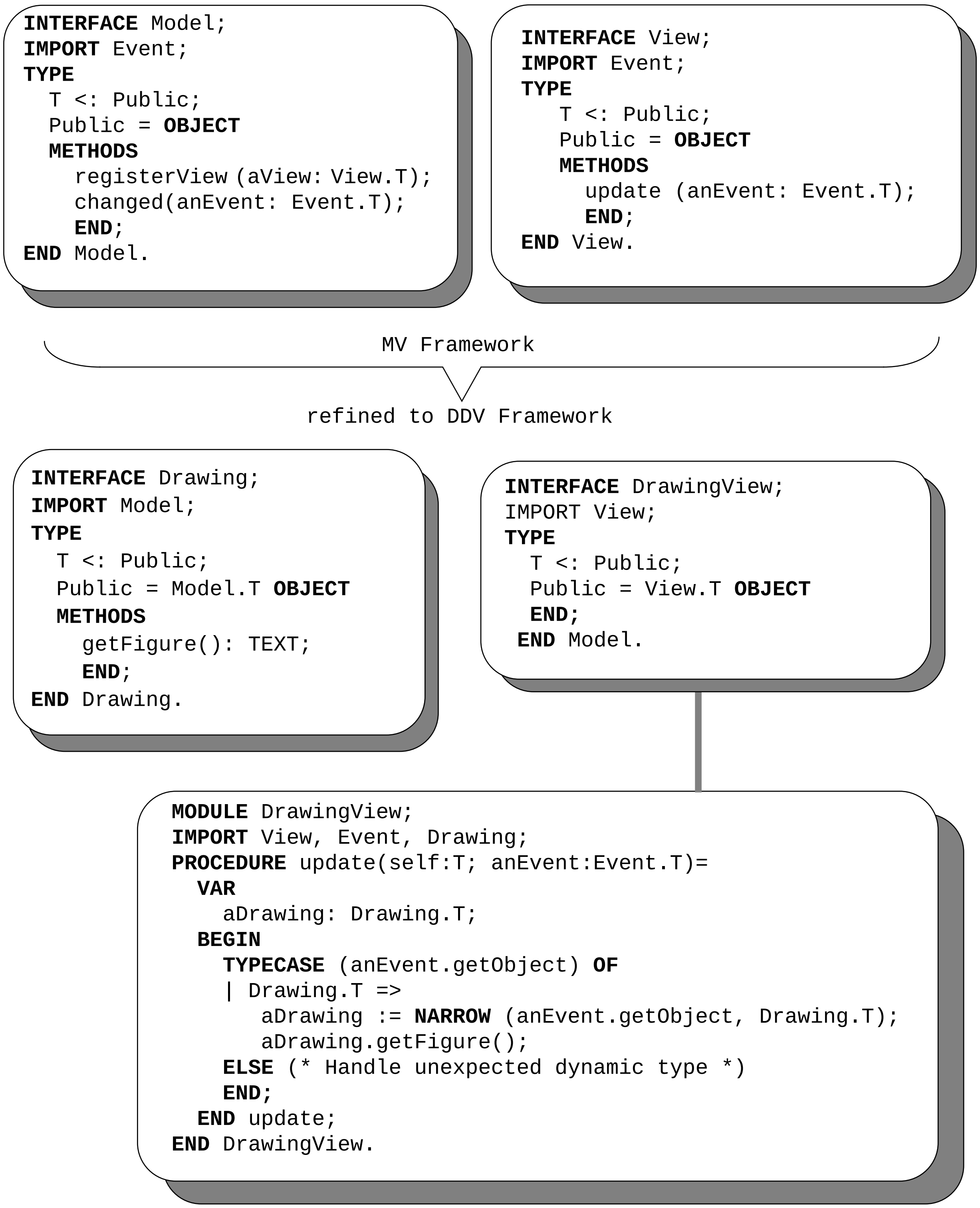
To determine the strengths and weaknesses of each of these typing policies to enforce invariants across framework operations, we built the MV framework in both a conservative contravariant (Modula-3) and a covariant (Eiffel Version 2) statically typed object-oriented language and applied a similar set of usage operations. We considered the ability of the typing policy to capture constraints on the instantiation of the MV framework, and on its refinement to support a graphical editor framework. The graphical editor framework consists of a Drawing that maintains information on graphical figures and a DrawingView that permits display of the Drawing. The ability of the typing policy to capture constraints on framework extension is also discussed.

3.2 MV Using Conservative Contravariant Typing

Figure 2 shows, in part, the interface portion of the implementation of the MV framework in Modula-3. The methods supporting the framework instantiation operation are the `New` methods (not shown) for each of the Model and View types,[3] and the `registerView` method of Model. Model and View participate in a structural relationship supported by the `registerView` method which connects one or more instances of View to an instance of Model. Model and View also participate in a behavioural relationship. When the `changed` message is sent to an instance of Model, all registered View instances are sent the `update` message. This is a behavioural data flow relationship where information is exchanged via the parameter of type Event. It is necessary to introduce the Event type because Modula-3 restricts Model and View from being self-referential. The typing policy of Modula-3 is useful in capturing and enforcing these constraints on the instantiation operation.

The contravariant typing policy fails to support the operation of framework refinement. Figure 2 also shows the result of applying a refinement operation to the MV framework to build a Drawing-DrawingView (DDV) framework. The Drawing subtype shares the common methods of `registerView` and `changed` with its type Model and introduces a new method, `getFigure`, to retrieve a graphical figure stored in a Drawing instance. The DrawingView type is introduced as a subtype of View. Figure 2 provides a portion of the implementation description of DrawingView to illustrate how the refined information about graphical figures in Drawing is accessed by DrawingView. Since the formal argument to the `update` method in

[3] Technically, Model and View are not types themselves, but are rather the names of modules. The actual types involved are Model.T and View.T. For simplicity, we refer to the types as Model and View.



```
INTERFACE Model;
IMPORT Event;
TYPE
  T <: Public;
  Public = OBJECT
  METHODS
    registerView (aView: View.T);
    changed(anEvent: Event.T);
    END;
END Model.
```

```
INTERFACE View;
IMPORT Event;
TYPE
   T <: Public;
   Public = OBJECT
   METHODS
     update (anEvent: Event.T);
     END;
END View.
```

```
INTERFACE Drawing;
IMPORT Model;
TYPE
  T <: Public;
  Public = Model.T OBJECT
  METHODS
    getFigure(): TEXT;
    END;
END Drawing.
```

```
INTERFACE DrawingView;
IMPORT View;
TYPE
  T <: Public;
  Public = View.T OBJECT
  END;
 END Model.
```

```
MODULE DrawingView;
IMPORT View, Event, Drawing;
PROCEDURE update(self:T; anEvent:Event.T)=
  VAR
    aDrawing: Drawing.T;
  BEGIN
    TYPECASE (anEvent.getObject) OF
    | Drawing.T =>
       aDrawing := NARROW (anEvent.getObject, Drawing.T);
       aDrawing.getFigure();
    ELSE (* Handle unexpected dynamic type *)
    END;
  END update;
END DrawingView.
```

Subtyping is specified in the Interface specifications using the OBJECT clause. The subtype inherits methods from its type. For example, Drawing is a subtype of Model since the type T (for Drawing) is defined as a subtype of the Public type which (in Drawing) is constrained to Model. Both the interface and implementation aspects of DrawingView are shown. The TYPE-CASE statement checks the type of the control expression. The NARROW statement returns the first argument as a value of the type given in the second argument.

Figure 2: Modula-3 Implementation.

DrawingView is constrained to be the same type specified for the method in the View type, DrawingView has to use dynamic type checking and casting to use the argument specified as type Model as an instance of the Drawing type. We are thus unable to use the conservative contravariant static typing policy that guarantees substitutability to describe the constraint on the behavioural relationship refined from the MV framework.

The failure of the type system to support a behavioural relationship under the operation of framework refinement also impacts the ability of the type system to capture the instantiation constraints for a properly refined framework. For example, the Drawing subtype shares the `registerView` method with its type Model. An instance of Drawing, however, expects only instances of DrawingView, and not View, to be registered. This structural relationship can no longer be captured by the type system after framework refinement.

Although a conservative contravariant typing policy fails to support the operation of framework refinement, the typing policy is useful in supporting the operation of framework extension. Consider extending the MV framework to a target framework where changes may be made to a model's data through a view. The extended target framework may be built from the MV framework by introducing subtypes of Model and View. These subtypes in the extended framework may be substituted for the types of the source framework as the new functionality does not impact the existing functionality of the MV framework.

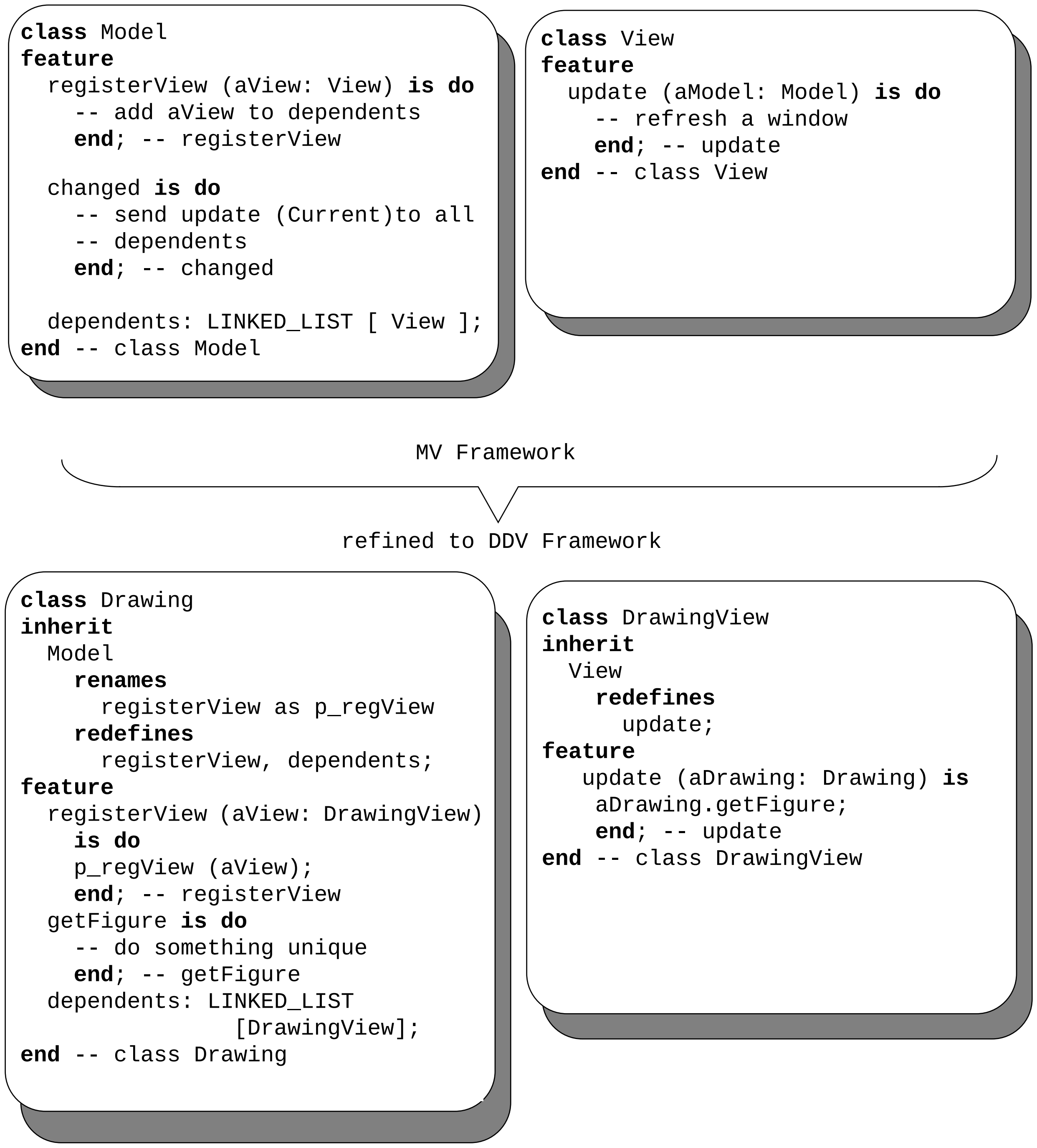
A conservative contravariant typing policy is thus useful for capturing and enforcing the constraints of instantiation for an non-refined, non-extended framework. Conservative contravariance also supports the operation of framework extension. However, the typing policy does not support the operation of framework refinement. Furthermore, conservative contravariance does not maintain instantiation constraints for a properly refined framework.

3.3 MV Using Covariant Typing

A portion of the Eiffel implementation of the MV framework is shown in Figure 3. This implementation of the framework consists of a Model class and a View class. In Eiffel, types are synonymous with classes. The methods supporting the operation of framework instantiation are the `Create` methods for Model and View (not shown) and the `registerView` method of Model. The formal argument to the `registerView` method in Model ensures that the structural relationship between Model and View is enforced. The covariant typing policy, like the conservative contravariant typing policy, thus supports the specification and enforcement of constraints on framework instantiation.

The covariant typing policy does not support the operation of refinement itself, but it does support the expression of a properly refined framework to aid subsequent instantiation. Consider expressing a proper refinement of the MV framework to the DDV framework as shown in Figure 3. Drawing is a subclass of Model and DrawingView is a subclass of View. The covariant type policy supports the refinement of the formal arguments to the `registerView` and `update` methods. The `registerView` method on Drawing is refined to require a DrawingView rather than a View thus maintaining the structural relationship constraint. The `update` method on DrawingView is refined to require a Drawing rather than a Model, thus maintaining the behavioural relationship.

Although a covariant typing policy with system validity checking can express the results of framework refinement, it does not ensure the refinement operation is completed thoroughly and correctly. In particular, structural and behavioural relationships of the framework may not be correctly refined across the operation. This can result in instantiations of the framework that are type correct and system valid, but which do not behave correctly at run-time. Consider an alternate refinement of the DDV framework shown below:



```
class Model
feature
  registerView (aView: View) is do
    -- add aView to dependents
    end; -- registerView

  changed is do
    -- send update (Current)to all
    -- dependents
    end; -- changed

  dependents: LINKED_LIST [ View ];
end -- class Model
```

```
class View
feature
  update (aModel: Model) is do
    -- refresh a window
    end; -- update
end -- class View
```

```
class Drawing
inherit
  Model
    renames
      registerView as p_regView
    redefines
      registerView, dependents;
feature
  registerView (aView: DrawingView)
    is do
    p_regView (aView);
    end; -- registerView
  getFigure is do
    -- do something unique
    end; -- getFigure
  dependents: LINKED_LIST
                [DrawingView];
end -- class Drawing
```

```
class DrawingView
inherit
  View
    redefines
      update;
feature
   update (aDrawing: Drawing) is
    aDrawing.getFigure;
    end; -- update
end -- class DrawingView
```

The Eiffel class fragments shown above consist of a *class* clause defining the name of the class (e.g., class Model), an optional *inherits* class defining the inheritance (and subtype) relationships between classes (e.g., Drawing inherits and is a subtype of Model), and a *features* section introducing the methods (and attributes) of the class (e.g., `registerView` is a method on the Model class). Comment lines are prefixed with two dashes.

Figure 3: Eiffel Implementation

```
class Drawing                       class DrawingView
inherits Model;                     inherits DrawingView
                                       redefines update;
   features                         features
      getFigure is...                     update (ad: Drawing) is...
end -- class Drawing                end -- class DrawingView
```

As before, the DDV framework is instantiated by registering an instance of DrawingView with an instance of Drawing. Unlike the DDV framework presented in Figure 3, however, this version of the framework does not covariantly redefine the `registerView` operation in the class Drawing to ensure the structural relationship. A system that instantiates the framework by registering a View with the Drawing is type correct and system valid, but will not behave correctly when the application is executed. Covariance, then, has not ensured that the structural and behavioural relationships are maintained across the operation of refinement.

Since conservative contravariance is a subset of covariance, the covariant typing policy supports the operation of framework extension. To extend the Eiffel MV framework in the same manner as the Modula-3 framework described previously, we introduce a NewModel and NewView class. As a subclass of Model, NewModel would share the existing methods of Model and would also introduce a new `update` method. NewView would subclass the existing View and add `registerModel` and `changed` methods. In this extended framework, instances of Model would have Views as dependents, and instances of NewModel would support dependents of types View or NewView.

A covariant type policy is thus useful for capturing and enforcing the constraints of framework instantiation, and extension. Once properly refined, covariance also supports the instantiation of a refined framework. A covariant type policy does not support the operation of framework refinement.

3.4 Discussion

To summarize, we found that neither the conservative contravariant or covariant static typing policy were entirely satisfactory in supporting all forms of framework use. Both typing policies provide benefits for easing the instantiation of a framework that is neither refined nor extended. Both typing policies also support the operation of framework extension. However, neither policy supports the operation of framework refinement. A covariant type policy does permit the expression of a properly refined framework to aid instantiation.

Most current statically-typed object-oriented languages, including C++, use variations of the conservative contravariant type policy. Conservative contravariance is attractive because it ensures substitutability of subtypes for types. With frameworks, this substitutability property is compromised whenever dynamic typing is used to subvert the type system across a framework refinement operation. The result is that at runtime, the subtype is no longer substitutable for the type. This is not just a by-product of the designs we implemented in Modula-3, but is also seen in other frameworks expressed in statically typed languages. For example, the C++ Unidraw framework [VL90] that refines the InterViews framework also uses type casting.

For supporting framework operations, covariance is the most attractive static typing policy found in common object-oriented languages. The system validity checking rules defined for Eiffel regain a portion of the substitutability compromised by the covariant type policy. Further analysis, however, is required to determine if the restrictions the system validity checker places on substitutability limit the ability to express and use a refined framework.

Most object-oriented languages used to build frameworks do not separate the class and type hierarchies. Although the separation of these hierarchies is important to distinguish between specification reuse (subtypes) and implementation reuse (subclasses) [CHC90], it does not impact the framework refinement problem. This is because the separation of the hierarchies is an orthogonal issue to the selection of a static typing policy. Similarly, whether an object-oriented language is class or prototype based [Lie86] does not appear to effect the typing policy.

Separation of the class and type hierarchies, however, can enhance the safe substitutability of a refined framework for a source framework within a conservative contravariant type policy. For example, the operations of the Model class in the MV framework may be split across two types: a ModelType that defines the `changed` operation; and a subtype of ModelType called ModelSubType that defines the `registerView` operation. When the DDV framework is refined from the MV framework, a new type, DrawingType, is introduced that is a subtype of ModelType. The DrawingType is implemented by the Drawing class. The DrawingType, like the ModelSubType, defines a `registerView` operation. Unlike the ModelSubType definition of `registerView` which takes a variable of type ViewType as a parameter, though, the DrawingType defines a `registerView` operation that takes as a parameter a variable of type DrawingViewType, a subtype of ViewType. This is permissible within the conservative contravariant type policy as the ModelSubType and DrawingType are not related. The result of this factoring is that an instance of class Model or of class Drawing may be assigned to a variable of type ModelType. The type system will enforce that only the `changed` method is invoked on the instance to which the variable of type ModelType refers. This supports the substitutability of the target DDV framework for the MV framework once the DDV framework is instantiated. This gain in substitutability comes at the expense of possible fragmentation in the type system.

A feature of several object-oriented languages, such as Eiffel, not exploited in the designs presented in Section 3 is parameterized types. Initially, parameterized types appeared to provide an attractive alternate design for the MV framework. Closer inspection, though, revealed that the Model and View types are mutually recursive. To remove the mutual recursion, we also considered a design that parameterized the Model and View types by an Event type which was used to pass information between the types (similar to the Modula-3 design in Section 3). This design, like its non-parameterized counterpart, required the use of covariance across the refinement operation to redefine the Event type to a specialized DrawingEvent type. As with the separation of the class and type hierarchies, then, the introduction of parameterization is orthogonal to the static typing policy.

Current research into typing policies for object-oriented languages focuses on the expression of recursively-defined types in the form of F-bounded polymorphism [CCOM89], [BCM+93], on alternatives to the use of procedural abstraction for treating encapsulation, such as Pierce and Turner’s work on object encapsulation through existential types [PT93], and on new definitions of the subtype relationship [LW93]. To support the operation of framework refinement, we need a typing system that will provide us even more. We need to express relationships between tightly coupled classes both as subsets (Drawing is a subset of Model) and as units (Model and View are the source for Drawing and DrawingView).

4 Related Work

Frameworks reduce the cost of developing a family of applications through the reuse of both design and implementation. This section considers related and complementary work on easing framework use, as well as other -- generally orthogonal -- approaches to reducing the cost of application development.

4.1 Frameworks

Most current approaches to increasing the leverage of frameworks fall into two categories: documentation and tools.

Documentation: Documentation approaches may be sub-divided into example-based and stylized. An example-based approach typically distributes framework source code along with a few examples of instantiated applications.The example-based approach is satisfying because it is simple for the framework designer and is often adequate for instantiating applications that are closely related to the provided examples. It is unsatisfying in the sense that it relies on informal information that is at times inconsistent, at times missing, and must, at times be inferred. Stylized approaches, like Patterns [Joh92], Contracts [HHG90] and multiple view documentation [CI92], address the limitations of an example-based approach by providing specialized documentation formats, languages and notations respectively to more precisely describe the design and use of a framework. Stylized approaches are limited by their distance from the implementation, making it difficult to automate enforcement of framework usage constraints.

Tools: Existing tool approaches to easing the use of a framework focus on the generation of refined frameworks and instantiated applications. Holland describes an environment for assisting the generation and integration of the framework classes based on an extension of the Contracts language [Hol92]. His approach focuses on easing the use of frameworks by leveraging the design. Brown developed the Zume tool to ease the use of an algorithm animation and multi-view editing framework called Zeus [Bro92]. Brown's approach, in contrast to Holland's, is to generate a refined framework and instantiate an application based on an existing framework implementation. Our attempt to ease framework use through static typing is most similar to Brown's as both focus on the implementation of the framework.

4.2 Application Development

Class Libraries: A class library supports the reuse of common components between multiple applications. Unlike a framework, components of a class library do not make visible structural and behavioural dependences with other library components. A class library component may use other components within the library, but this is invisible to the user.

Application Generators: Examples of application generators are spreadsheets, report generators and screen builders. Most application generators provide a tailored language and environment to reduce the cost of building an application. This differs from a framework that is embedded within a general purpose language. Application generators typically provide more leverage for building an application than a framework, but this is at a higher cost to produce the generator. Unlike frameworks, application generators are rarely used to generate other application generators. This is, in part, because application generators are more specific to their domain than a framework. It is also not generally possible to compose application generators to build a system.

Transformation-based Application-Specific Environments: Garlan et al. have proposed a transformation-based technique to automate the generation of application-specific environments [GLN92]. Transformations may be applied semi-automatically with the ASCENT tool to tailor a general purpose programming environment to a particular application domain. Applications built in the generated programming environment are then automatically transformed to execute within the general purpose programming environment and language that served as a base for the transformation. ASCENT thus provides a different kind of access to the general purpose language than a framework. With a framework, the general purpose language it is embedded within is always accessible, while in ASCENT, the general purpose language used is hidden from the application builder. The framework may be composed with other applications and systems through the general purpose language. When using the transformation-based generation approach, access to the general purpose language must be built in by the user.

A framework also differs from the ASCENT approach in how it is used. With ASCENT, a general purpose environment and language is tailored into an application-specific environment. The transformation is applied once. In the case of a framework, the framework operations of refinement, extension and instantiation may be applied repeatedly using the target framework from one operation as the source target for another operation. When a framework is reused in this manner, both the design and implementation are reused. A framework thus provide reuse at a finer granularity than the transformation-based approach and may be reused in multiple different ways.

Software Architecture: Research into software architecture [Sha90] focuses on developing more formal descriptions of the structure of a software system. This involves the use of existing formal notations [AG92] and the development of new notations and languages for describing the components and interactions between components of a software system [AAG93]. In this way, research into software architecture is similar to framework research. Frameworks differ significantly from software architecture in their simultaneous reuse of both design and implementation. In contrast, software architecture research focuses on specification rather than design and implementation.

Software Extension and Contraction: Parnas describes various techniques for supporting the extension and contraction of software [Par79]. An understanding of how to extend and contract software permits a change in focus from the building of a single program to the building of a family of related programs. One of the techniques proposed for supporting this change in focus is to design software according to the “uses” relation which introduces layering into the software system. The framework operations proposed in this paper are also mechanisms for handling the extension and contraction of software within a family of applications. The framework operations do not subsume but rather complement layering. By developing a framework and using the framework to build other frameworks and applications, the flexibility of the software is increased. The target framework that results from applying a framework operator to a source framework is not an instance of software layering, but a different mechanism for extending the software.

5 Summary

Frameworks reduce the cost of development by improving the reuse of design and implementation between related applications. Although frameworks have proven beneficial in practice, they are often criticized for being difficult to understand and use. In this paper, we characterized the operations software engineers apply to use a framework and investigated how static typing policies for common object-oriented languages may be used to improve the leverage of frameworks. In particular, we found that:

- both conservative contravariance and covariance support the instantiation of non-refined and non-extended frameworks,
- both typing policies support the extension operation,
- neither typing policy supports the refinement operation,
- a covariant typing policy supports the expression of a properly refined framework to aid with subsequent instantiation.

Neither conservative contravariant or covariant typing policies, the two typing policies found in production object-oriented languages, are satisfactory in supporting all three framework usage operations.

We believe that static typing policies will be most effective in conjunction with tools; particularly since support for current object-oriented languages is needed. For instance, the CCEL language and tools [MDR93] that permit software engineers to express constraints on the design and implementation of applications may provide a basis for enforcing the stylistic use of dynamic typing within a conservative contravariant typing policy. Additional tools would be necessary to determine and maintain substitutability given this approach. For languages using a covariant typing policy, flow analysis tools in conjunction with system validity checking may be sufficient to support the operation of framework refinement.

Acknowledgments

Craig Chambers, Lea Fester, William Griswold and Mike VanHilst provided valuable comments on an earlier draft of this paper.

References

[AAG93] Gregory Abowd, Robert Allen, and David Garlan. Using Style to Understand Descriptions of Software Architecture. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, December 1993. To appear.

[AG92] Robert Allen and David Garlan. A Formal Approach to Software Architectures. In *Algorithms, Software, Architecture. Information Processing 92. IFIP 12th World Computer Congress.*, pages 134–141, September 1992.

[BCM$^+$93] Kim B. Bruce, Jon Crabtree, Thomas P. Murtagh, Robert van Gent, Allyn Dimock, and Robert Muller. Safe and Decidable Type Checking in an Object-Oriented Language. In *Proceedings of the OOPSLA '93 Conference on Object-oriented Programming Systems, Languages and Applications*, 1993. To appear.

[BE93] Andreas Birrer and Thomas Eggenschwiler. Frameworks in the Financial Engineering Domain: An Experience Report. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 21–35, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Bro92] Marc H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. Technical Report SRC-075, DEC Systems Research Center (SRC), February 1992.

[CCOM89] Peter Canning, William Cook, Walter Olthoff, and John C. Mitchell. F-Bounded Polymorphism for Object-Oriented Programming. In *Proceedings of the Fourth International Conference on Functional Programming, Languages and Computer Architecture*, pages 273–280. ACM Press, September 1989.

[CDJ$^+$89] Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow, and Greg Nelson. The Modula-3 Type System. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 202–212, New York, NY, USA, January 1989. ACM, ACM.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. In *Conference Record of the 17th Annual ACM Symposium on Principles of Programming Languages*, pages 125–135. ACM, January 1990.

[Che76] P. P.-S. Chen. The Entity-Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, March 1976.

[CI92] Roy H. Campbell and Nayeem Islam. A Technique for Documenting the Framework of an Object-Oriented System. In *Second International Workshop on Object-Orientation in Operating Systems*, pages 288–300, Paris, France, October 1992. IEEE Computer Society Press.

[CIM92] Roy H. Campbell, Nayeem Islam, and Peter Madany. Choices, Frameworks and Refinement. *Computing Systems*, 5(3):217–257, 1992.

[Coo89] William Cook. A Proposal for Making Eiffel Type-safe. In S. Cook, editor, *Proceedings of the ECOOP '89 European Conference on Object-oriented Programming*, pages 57–70, Nottingham, July 1989. Cambridge University Press.

[Deu89] L. Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 System*, volume II of *Frontier Series*, chapter A.3, pages 57–71. ACM Press, 1989.

[FM89] Patrick J. Ferrel and Robert F. Meyer. Vamp: The Aldus Application Framework. In *Proceedings of the OOPSLA '89 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 185–190, October 1989. Published as ACM SIGPLAN Notices, volume 24, number 10.

[GLN92] D. Garlan, Cai Linxi, and R.L. Nord. A Transformational Approach to Generating Application-Specific Environments. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 68–77, December 1992. Published as SIGSOFT Software Engineering Notes, Volume 17, Number 5.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the OOPSLA/ECOOP '90 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 169–180, October 1990. Published as ACM SIGPLAN Notices, volume 25, number 10.

[Hol92] Ian M. Holland. Specifying Reusable Components Using Contracts. In O. Lehrmann Madsen, editor, *Proceedings of the ECOOP '92 European Conference on Object-oriented Programming*, LNCS 615, pages 287–308, Utrecht, The Netherlands, July 1992. Springer-Verlag.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[Joh92] Ralph E. Johnson. Documenting Frameworks using Patterns. In *Proceedings of the OOPSLA '92 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 63–76, October 1992. Published as ACM SIGPLAN Notices, volume 27, number 10.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of the OOPSLA '86 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 214–223, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing User Interfaces with InterViews. *Computer*, 22(2), February 1989.

[LW93] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 118–141, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[MDR93] Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the Structure and Style of Object-Oriented Programs. Technical Report CS-93-12, Brown University, Box 1910 Providence, RI 02912, April 1993.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, September 1992.

[Par79] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2), March 1979.

[PT93] Benjamin C. Pierce and David N. Turner. Simple Type-Theoretic Foundations for Object-Oriented Programming. *Journal of Functional Programming*, April 1993.

[SB90] M. Spenke and A. Backer. GINA – An Application Framework Based on a Language Binding for OSF/MOTIF and Common Lisp. In *Proceedings of the European X Window System Conference*, pages 46–53, November 1990.

[Sha90] Mary Shaw. Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5:119–128, 1990.

[VL90] J.M. Vlissides and M.A. Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[WGM88] André Weinand, Erich Gamma, and Rudolph Marty. ET++ – An Object-Oriented Application Framework in C++. In *Proceedings of the OOPSLA ’88 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 46–57, November 1988. Published as ACM SIGPLAN Notices, volume 23, number 11.

[WZ88] Peter Wegner and Stanley B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn’t Like. In S. Gjessing and K. Nygaard, editors, *Proceedings of the ECOOP ’88 European Conference on Object-oriented Programming*, LNCS 322, pages 55–77, Oslo, August 1988. Springer Verlag.

The Interaction Between Static Typing and Frameworks

Gail C. Murphy and David Notkin

Technical Report 93-09-02

October 13, 1993