Using Predicate Fields in a Highly Flexible Industrial Control System

Shay Artzi* Michael D. Ernst

MIT Computer Science & Artificial Intelligence Lab
32 Vassar St, Cambridge, MA 02139 USA
{artzi,mernst}@csail.mit.edu

ABSTRACT

Predicate fields allow an object's structure to vary at runtime based on the object's state: a predicate field is present or not, depending on the values of other fields. Predicate fields and related concepts have not previously been evaluated outside a research environment. We present a case study of two industrial applications with similar requirements, one of which uses predicate fields and one of which does not. The use of predicate fields was motivated by requirements for high flexibility, by unavailability of many requirements, and by high user interface development costs. Despite an implementation of predicate fields as a library (rather than as a language extension), developers found them natural to use, and in many cases they significantly reduced development effort.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*User interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*

General Terms

Design, Experimentation, Languages

Keywords

Predicates, Predicate Fields, Classifiers, Structure, User Interface Development, Experimental Control System

1. INTRODUCTION

At the heart of object-oriented programming is the ability for run-time values to affect program behavior. For example, dynamic dispatch selects which implementation of a method to invoke, based on the run-time class of the receiver object. Researchers have proposed other, more powerful ways for run-time values to affect both the *behavior* and the *structure* of objects.

Multiple dispatching [2, 7, 11] permits dynamic dispatch to depend on the types of all arguments, not just the receiver. Predicate dispatching [13, 22] uses arbitrary predicates (boolean expressions) to select the most specific applicable implementation, so dispatch can depend on any aspect of program state.

*Work done while at Rafael, Israel Armament Development Authority, Ltd.

OOPSLA'05, October 16–20, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

Predicate fields (the subject of this paper) give a way of affecting object structure rather than object behavior. A predicate field is a field of an object that is present (can be read and written) depending on the values of other fields. In other words, the structure of an object varies at run time depending on the values of its own fields.

Other research combines aspects of both lines of research, permitting dynamic changes in both behavior and structure. Predicate classes [8] use predicates to dynamically classify an object to its subclasses. Classifiers in the KEA programming language [19, 23], which are based on the value of a single enumerated-type attribute, are also used to automatically classify an object to mutually-exclusive subclasses. Taivalsaari's Modes [28] are another similar mechanism.

Despite lively interest from the research community, we are not aware of any prior substantive evaluation of predicate-oriented programming. We have implemented a library that supports the creation of dynamic objects, which are objects with predicate fields. While the library forces users to use a special syntax for accessing dynamic objects, and the predicates are of limited expressiveness, the library was nonetheless powerful enough and usable enough to be the basis for a mission-critical industrial system.

This paper presents a case study of two systems built to similar requirements. The first implementation is a command and control system for a facility in which physical experiments are performed. The second implementation is similar, though more ambitious, and was designed around the use of predicate fields in order to mitigate deficiencies in the first implementation. This use of predicate fields was successful: the resulting system is very flexible, the user interface is highly configurable, and development costs, especially in the user interface (which can account for 50% of effort expended [24]), were greatly reduced.

We believe this is the first evaluation of predicate-oriented computing in a large-scale development effort. Our observations help to indicate circumstances in which predicate fields are most useful, and the practical obstacles to their use. Our observations also suggest that many important benefits are obtained even in the absence of sophisticated predicate expressions and dispatching mechanisms. One implication for practitioners is that it can be advantageous to use even a primitive implementation of this language feature.

The remainder of this paper is organized as follows. Section 2 reviews predicate fields, and Section 3 discusses use of our predicate fields library. Sections 4–7 present our case study: requirements, system implementation, use in practice, and evaluation. Section 8 details the library implementation. Section 9 surveys related work, and Section 10 concludes.

```
// A statement in an interpreter for the MML
// programming language (Section 4.1).  Either
// a manual statement, or an automatic statement.
class Statement;
enum StatementType
         {ManualStatement, AutomaticStatement};

// PStatement is a predicate that is true
// when the object is of type Statement.
// dynamicType is a built-in field; see Figure 10.
pred PStatement
         dynamicType = Statement;
pred PManualStatement
         statementType = ManualStatement;
pred PAutomaticStatement
         statementType = AutomaticStatement;

// The field named statementType (of type
// StatementType) exists in any object for which
// the predicate PStatement evaluates to true
// (namely, any object of class Statement).
field StatementType statementType when@PStatement;
field string description when@PManualStatement;
field Device device when@PAutomaticStatement;


Statment s = new Statement();
s.description = "Please push red button"; // ERROR
s.statementType = ManualStatement;
s.description = "Please push red button";
s.device = new Oscilloscope(); // ERROR
s.statementType = AutomaticStatement;
s.device = new Oscilloscope();
```

Figure 1: Using predicate fields to emulate dynamic classification for IDE development (stylized syntax). Lines marked as "ERROR" result in run-time exceptions.

2. PREDICATE FIELDS

A predicate field is a field whose presence in an object depends on the run-time value of a predicate — a boolean expression. Predicate fields allow an object's structure to vary at run time. When the predicate is satisfied, the field is part of the object's structure and can be read and written. If the predicate later becomes unsatisfied, the field ceases to be part of the object's structure.

Different proposals for predicate-oriented programming use predicates of varying complexity. Predicate classes [8] permit arbitrary boolean expressions, while Kea classifiers [19, 17] are determined by the value of a single attribute of an enumerated type. Our implementation is similar to Kea in that predicates are based on equality of a single variable to a specified value. However, the variable's type may be a string or any primitive type.

Predicate fields may seem to be a limited mechanism. However, they can be used to emulate predicate classes (dynamic classification of an object into subclasses), as shown in Figure 1. This is similar to, but more convenient than, design patterns such as decorator or state aimed at a similar purpose [14].

Here we note four motivations for predicate fields. (Other research has provided additional motivations.)

First, predicate fields allow an object to change its structure during its life cycle. For example, an employee might initially have a field for his hourly wage. After a promotion to salaried status, such a field is no longer relevant, but a monthly or yearly salary field is necessary. Instead of using an elaborate class hierarchy to, for instance, separate the employee object from his method of payment, a developer can use predicate fields to create the suitable object structure.

Second, predicate fields permit the user to recover from errors, such as changing the type of a created object while preserving all the data which is mutual to both types. The alternative, deleting the old object and creating a new one, can cause lost work and errors.

```
pred hasThesisProposal
             thesisProposal != null;

class StudentProgress
{
  ...
  String thesisProposal;
  Date anticipatedDate when@hasThesisProposal;
}
```

Figure 2: Definitions for controlling visibility of fields in a graduate student progress report.

Third, predicate fields can ease development of graphical interfaces (and other components) by presenting a unified view on arbitrarily many collections of fields. As one example, if reflection properly recognizes the fields, then even though there may exist many different varieties of object, each with its own collection of fields, the code for manipulating objects can be simple and compact.

Fourth, predicate fields permit fine-grained customization of objects or object behavior, without requiring new class definitions. For example, extending the employee example above, a specific employee's human resource information might be tagged with special fields relevant only to that employee.

2.1 Predicate field example

As another example of use of predicate fields, consider a web site where a graduate student enters information about progress toward his degree. The information includes (among others) PhD thesis proposal, and expected date for the completion of the PhD thesis.

The expected date for the PhD thesis is not meaningful until the PhD thesis proposal is submitted: the former field should not exist, or at least not appear in the web form, when the PhD thesis proposal is empty. A possible object oriented solution would be to abstract the notion of the field's existence in the student object, by creating an association object between the student object and its various fields. User interface code would consult this association object when generating or updating the screen. Figure 2 shows another solution using predicate fields. When `thesisProposal` gets set, the field `anticipatedDate` becomes part of the object's structure. The developer can even generate the user interface using reflection to automatically incorporate field changes into the user interface.

3. USING PREDICATE FIELDS

This section discusses how a programmer uses our implementation of predicate fields.

3.1 Library implementation

We have implemented predicate fields as a library in C#. Previous predicate-oriented systems [17, 19, 8, 28, 13, 22] are based on language extensions, some of which are accompanied by a prototype implementation. Our decision to use a library is a pragmatic one.

Developing a library is much easier than modifying or creating a new compiler, not to mention the continuing work of adapting it to new versions of evolving languages such as Java and C#. Use of a library allows the software to remain compatible with existing tools (IDEs, compilers, editors, etc.), and it is congruous with the programmer's knowledge and practices.

Choosing a library implementation over a language extension

```
// Statement represents the dynamic type
// definition for the statement hierarchy.
DynamicType Statement {
    storageType storage = DataBase;
    String tableName = TStatements;
}
```

Figure 3: Stylized dynamic type definition representing statements in the MML programming language. For simplicity of presentation, most figures use a simplified syntax that assumes that predicate fields are built into the C# language. The actual implementation is as a library (see Section 8), making the syntax less convenient than that shown here.

has negative consequences as well. Using the library is more cumbersome. A library may suffer from worse performance than a language implementation. The choice of a library also caused us to restrict the expressiveness of the predicate fields. The predicates are based on the value of a single field, and only changes to object structure (not to object behavior as in predicate classes or predicate dispatching) are supported.

As noted in Section 2, predicate fields can emulate subclassing, and reflection can be used to respond to structure changes. One of the contributions of our work is that the relatively impoverished version of predicate fields that we implemented was nonetheless of significant practical benefit, and was sufficient for the needs of a substantial industrial project.

3.2 Defining dynamic objects

Our library supports the creation of *dynamic objects*, objects in which each field is a predicate field. A dynamic object is an instance of a *dynamic type*. Conceptually, the declaration of a dynamic type lists the (predicate) fields contained in each object of the type. However, our library implementation uses distinct syntax for type definitions, fields, field types, and predicates. Together, these support declaration, creation, and use of dynamic types and dynamic objects.

The following sections show how the four pieces (dynamic types, fields, field types, and predicates) are used to create an automatic statement, which is a statement in the MML programming language (see Section 4.1) that can be executed without human interaction. When the user defines an automatic statement, he must select a device type, a command that is valid for this device type, and the target device of this type.

3.2.1 Dynamic type declaration

Each dynamic object has a type, called its *dynamic type*; the dynamic types partition the set of all dynamic objects. The dynamic type is stored in a dynamic object upon creation and is permitted to change subsequently. The fields that belong to a dynamic type are not listed in the dynamic type declaration, but are separately specified (Section 3.2.2), permitting predicate fields to be added to a dynamic type without changing the type declaration.

In our case study, the experimental control system supports persistent objects, so the dynamic type definition indicates where objects of the given type should be stored to and loaded from. Two common choices for the storage location are a database table name or an XML file name; the system also supports storage in memory.

Figure 3 gives an example of a stylized definition of the `Statement` dynamic type.

3.2.2 Fields

A field definition contains the following information:

Fundamental information name, field type (see Section 3.2.3), predicate (see Section 3.2.4). The field exists in any object for which the predicate is true.

Persistence information How to store to or load from a database. Field values are automatically loaded and saved. The storage information includes the column name, table name, foreign key 1, and foreign key 2. The last three properties determine the field's storage location, when it is not stored in the object table as defined in the dynamic type. See Section 8.2.

User interface information How to display, edit, and parse values. The user interface information includes category (used to separate the fields into different sections), description, and visibility. This user interface information cannot be specified in the field type, since the same field type might be used for several fields, each of them has different user interface properties; consider the `String` field type as an example.

Optional information default value, read-only, assignable-once (if true, the field becomes read-only after it is set for the first time), change notification (an object, such as a container, can register to receive an event when a field value changes).

```
// The command field in an automatic statement
// contains the command that is sent to the device.
Field command
{
    // Fundamental information
    fieldType DeviceCommand;
    pred PAutomaticCommand;
    name = "Command"

    // Persistence information
    column = "CommandId";

    // User interface information
    shouldRaiseEvent = true;
    category = "Device Information";
    description = "Command sent to the device";
    viewable = true;

    // Optional information
    defaultValue = nopCommand;

}
```

Figure 4: Stylized field definition.

Figure 4 contains an example, the `command` field declaration.

An implementation could create and allocate a predicate field lazily (only when it is needed); alternatively the field might always be part of the object structure, but unavailable until the predicate is satisfied [8]. Our library implementation allows both approaches, storing fields either in the database or in memory. The values of fields stored in the database are preserved over time. The predicates control which of those values can be accessed. Fields stored in memory are created and deleted when predicates' satisfaction changes and do not save their value over time.

3.2.3 Field types

Field types specify the type of the data that is stored in the field, a class used to edit its value, and a class used to convert its value to and from a printed representation.

As an example of a field type definition, consider the `DeviceCommand` field type in Figure 5, which defines a field type for device commands. A device command is used to abstract a method in the device's API.

As another example of customization of editors and converters, we replaced the Boolean type default editor with a type editor that lets the user choose from a list of native language words rather than

```
// FieldType Wrapper which allows a
// dynamic object to have dynamic
// command field.
FieldType DeviceCommand
{
    // The Command class is an abstraction
    // of a method in the device's API.
    type Command;
    // The CommandEditor is a class which will
    // query the object for its device type, then
    // it will build and present the list of
    // available commands for this device type.
    editor Editors:CommandEditor;
    // CommandConverter will convert the field
    // enumerated value to a viewable command
    // name and vice versa.
    converter Converters:CommandConverter;
}
```

Figure 5: Stylized field type definition.

```
pred PStatement dynamicType = Statement;
pred PDevice dynamicType = Device;
pred PDeviceType dynamicType = DeviceType;
pred PCable dynamicType = Cable;
pred PDeviceCommand dynamicType = DeviceCommand;
pred PGroup dynamicType = Group;
pred PSetup dynamicType = Setup;
```

Figure 6: Stylized predicate definitions.

"true" or "false". The converter was then used to convert true and false to native language and back.

Field types are distinct from dynamic types (Section *3.2.1*): we made a simplifying implementation decision disallowing a field from containing a dynamic object, and so one dynamic object cannot directly refer to another. Nevertheless, fields in different dynamic objects can share the same storage location (such as a particular database table), thereby allowing an indirect connection between the objects. For example, if a statement's device field refers to a device in the database, then the device name could be changed either in the statement, or in a dynamic object representing the device. Either modification will be reflected in the backing database and in the other dynamic object.

3.2.4 Predicates

In our library, a predicate is constructed from a pair of a field name and a value of the field's type. Predicates are not arbitrary boolean formulas. Our predicates can be used to capture any desired property of dynamic objects, so the syntactic limitation is not a semantic one.

This decision implies that developers can define a predicate on every possible value of the object's state. Setting a special field name for the dynamic type allows us to define the group of predicates shown in Figure 6; these predicates are used to define the common structure of the type.

4. CASE STUDY: EXPERIMENTAL CONTROL SYSTEM

We performed a case study involving two systems that control facilities for complex experiments. The two implementations serve as command and control software for experiment-conducting facilities. Their goal is to enable scientists and technicians to define, control, execute, and examine these experiments. The experiments demand control of complex events involving vast numbers of accurate measurement devices and often very delicate equipment. The experiments' results must be saved. Those results are later processed and analyzed by the experiments' orderers.

1. Manual("Input object serial number, or zero to stop");
2. Setup(vacuum_machine, MRM);
3. Manual("Place object on scale");
4. scale.Read();
5. Manual("Place object on MRM");
6. camera.Setup(1);
7. vacuum_machine.Start();
8. If vacuum_machine.status = on, then
    (a) MRM.Release();
    (b) camera.ReadPhotos();
    (c) vacuum_machine.Stop();
9. Goto step 1

Figure 7: Galileo's free fall experiment, expressed in a variant of the MML programming language. "MRM" stands for "mechanical release machine".

Section 4.1 explains the abstraction that models an experiment and the language in which users write experiments. Section 4.2 presents the high-level requirements and the relevant design decisions used to meet those requirements. Section 4.3 discusses user interaction with the system. Finally, Section 4.4 compares the system to a programming language implementation.

4.1 Experiments and the MML language

An experiment consists of a set of devices and a sequence of operations of those devices. Examples of devices are thermometers and drill presses. Examples of operations are measuring the temperature and drilling a hole, in addition to setting up the devices with initial parameters and rudimentary data analysis.

The operations of an experiment are written in a language called MML, or Mission Modeling Language. (The term "mission" is a synonym for "experiment".) MML includes statements that are executed automatically, statements that must be executed manually (in which case MML prints a message and waits for a human operator to confirm that the operation has been performed), and statements that are executed semi-automatically. MML's control structures include sequential composition, parallel composition, procedure calls, conditional (`if`) statements, and limited support for iteration (loops), which is rarely necessary in our context. The C# implementation of the MML programming language uses predicate fields — though the MML language itself does not support predicate fields.

To give a flavor of the MML language, Figure 7 shows how Galileo's free fall experiment might be expressed in MML. The purpose of this experiment is to show that the rules of gravity apply equally to all objects. The experiment is performed in a vacuum and consists of dropping different objects from different locations and measuring their trajectories and fall time. The weight, initial height, and fall time of each object is measured and stored.

The list of devices (not shown) includes all the equipment that participates in the experiment: a vacuum machine, a mechanical release machine (MRM), a time-stamped camera, and a scale. Consumable hardware (such as the objects that are dropped) is not considered part of the experiment definition nor included in the list of devices.

4.2 Requirements and design

The key requirement for the experimental control system is adaptability to physical hardware changes and replacements. New devices are developed constantly; incorporating those devices into the system should require no software changes in the experimental control system (but may require changes to experiments). In addition, the system should give the users the ability to conduct an experiment, repeat it, and run simultaneous experiments, as well as repeat the same experiment steps with a different set of hardware. These requirements were met by adopting two major design decisions.

The first design decision is the definition and development of a Mission Modeling Language (MML) for writing experiments. The system software controls and manages various experiments ("missions") that take place simultaneously. Experiments expressed in MML specify both system behavior and user interaction with the system. MML was described in Section 4.1.

The second design decision is use of a highly configurable two-level system architecture. The two levels of the system are a knowledge level and an operational level. The operational level describes the concrete model of the system, derived from the functional requirements. The knowledge level contains the meta-model of the system and defines the legal configurations of operational level objects. The knowledge level describes the physical world that should be known to the system, including devices, device types, commands, protocols, cables, and IO channels.

4.3 Users

The system serves three different user groups: Experiment Designers, Experiment Operators, and Knowledge Level Editors.

The Experiment Designer creates an experiment from requirements supplied by a domain expert. The Experiment Designer also performs rudimentary analysis on the results to verify that the experiment was executed correctly.

The Experiment Operator runs the experiment. His editing capabilities are limited to skipping or re-executing statements and making minor changes to the information sent to the devices. He cannot make permanent changes to the experiment.

The Knowledge Level Editor is an electrical engineer responsible for developing new hardware and drivers for the system. He can edit the knowledge level in order to incorporate the new devices into the system.

The user interface for the Experiment Designer and the Experiment Operator is tree-based. The Experiment Designer adds new statements to the statements tree and new devices to the devices tree. Each element in the tree can be configured by a related properties form. The Experiment Operator can observe the properties of the various items in the trees as well as the progress of the experiment shown on the statement tree.

4.4 Programming language metaphor

The experimental control system can be viewed not only as a command and control software for defining and controlling experiments, but also as a programming language implementation supporting persistent objects. Because none of the system's users is trained in computer science or programming, a typical editor-compiler interface to MML would not be appropriate.

In the programming language metaphor, the Experiment Designer is a programmer, and the Experiment Operator executes an experiment in debug mode. The operational level of the architecture (Section 4.2) is the analog of the development environment, editor, compiler, linker, and executor. The knowledge level of the architecture includes the type definitions, and the Knowledge Level Editor defines new types. The developers used predicate fields to add the MML statements definitions to the knowledge level.

Ordinarily, a programmer can change the type of a variable by changing a declaration in a program. The Experiment Designer can change types by using the tree-structured GUI to set the type of an object. This change of type affects what fields exist in the object. In the system, all objects are persistent (they are stored in a database, for example), so the change of type must be reflected in all accesses to the existing object rather than by exiting the program and re-running it to create a new object.

5. IMPLEMENTATIONS

We built two implementations of the experimental control system that satisfy the requirements of Section 4. The first implementation is in daily use at one experimental facility. The second implementation is in incremental integration at a different facility and will eventually replace the first implementation. Re-implementation took advantage of knowledge gained from the first effort and permitted exploration of different design decisions, most notably the use of predicate fields.

5.1 Implementation 1

It took almost fifteen man years to develop Implementation 1. It was developed in Borland's Delphi IDE [29] using the Object Pascal programming language. Is was designed as a multi-tiered application [18] composed from many distributed components relying on Microsoft COM/DCOM technology [6, 10, 4]. This implementation contains approximately 100,000 lines of code.

The project was divided into four clients: experiment editing, experiment execution, knowledge level architecture, and results analyzer. The defined experiments, the knowledge level, and the results are stored in the database. Their corresponding objects, residing in the appropriate server components, are persistent objects [25]. The implementation contained the following servers: device engine, statement engine, result engine, experiment engine, and a message dispatcher. The clients communicated with the appropriate server to receive and send information about the persistent objects (statements, devices, experiments, etc). For example, the device engine component was responsible for supplying and editing device information as well as using the device when the experiment was activated.

The software system contained knowledge level capabilities that enable the dynamic addition of new types of devices — with their specific sets of protocols, commands, and I/O channels — as a plug-in operation, without software changes or re-compilation. Let us consider "oscilloscope" (in short, "scope") as an example. A scope is a measurement device; different scopes are manufactured by different companies, but since they are all made for the same purpose they all share a basic set of commands such as: turn on/off, BIT (built in test), read measurements. However, different scopes support different numbers of I/O channels, and some scopes offer more advanced commands. The system is flexible enough to enable the use of different kinds of scopes, the addition of new modern scopes to replace old ones, and the utilization of the new options provided by the modern equipment.

5.1.1 Deficiencies of Implementation 1

Implementation 1 is considered highly successful. It has been used daily since 2001 to design and run many experiments. It transformed the way that work was performed at the facility where it is installed, and other facilities are eager to obtain it. The development team won several company prizes for outstanding contribution to goals of a core division project.

While Implementation 1's adaptability to anticipated changes is a success, in retrospect its design unduly burdens users when adapting to other, unanticipated changes. An example for such a change is adding a new device types that require system awareness of their setups. This section discusses three design deficiencies that led to the decision to use predicate fields for the development of Implementation 2.

First, each object in the user interface (statement, device, device type, group, command, etc.) had a custom-made property page for editing its information. User interface development took a significant share of the development time, and it was difficult to integrate new devices with unfamiliar editing information. For example, a new form had to be developed (possibly inheriting and extending an existing form) when adding a new statement type (a statement is one step in an experiment, as described in Section 4.1), a new field to an existing statement type, or a new device type with different setup fields.

Second, changes to the structure of the statements required cross cutting layer changes. As an example, consider adding a `max_repeat` field that limits the number of times the Experiment Operator can repeat a statement. This modification results in changes to the database, database connectivity layer, business logic, client to business logic facade server side, client to business logic facade client side, statement view object client side, and the statement abstract form. This seemingly simple change required extensive developer effort and expensive process before releasing a new version or even a patch.

Third, the type of an object could not be changed. Suppose the Experiment Designer added an automatic statement, then wanted to change it to a manual statement. He would have to delete the automatic statement and create a new manual statement in its place. All the information in that statement is deleted and similar fields must be set again. Other tools used for experiment design suffer from the same problem [15].

5.2 Implementation 2

Implementation 2 was developed using Microsoft Visual Studio .NET and the C# language [16, 20, 5]. The DCOM underlying distribution layer was replaced with .NET Remoting [27, 21]. Although the fundamental requirements were similar to the previous project, Implementation 2 controls more, and more complicated, hardware than Implementation 1, and it will eventually replace Implementation 1. Management decided to develop Implementation 2 from scratch in the new language and environment, based on perceived longevity of the development environment, availability of libraries, and opportunities to improve the system's design. Most notably, Implementation 2 uses predicate fields.

The schedule of the project, which was dictated by external sources, required starting development while most of the requirements (controlled devices and experiments) were unknown or vague. A team of five developers (two of whom, including the team leader, also worked on Implementation 1) produced functionally equal to Implementation 1 in less then two years. This decrease in development time can be attributed to reusing knowledge learned from Implementation 1, using a better development framework, and using predicate fields to address some of the problems of Implementation 1. One design change is that whereas Implementation 1 is a package of four client programs, Implementation 2 consists of one client with all the necessary functionality, in order to create a unified look and feel.

5.2.1 Motivation for predicate fields

This section gives three motivations for the use of predicate fields.

First, we realized that all objects share the same characteristics while being edited. Implementation 1 connected the user interface tree's nodes (statement, device, ...) with viewers of the appropriate objects. However, it caused tight coupling between the objects

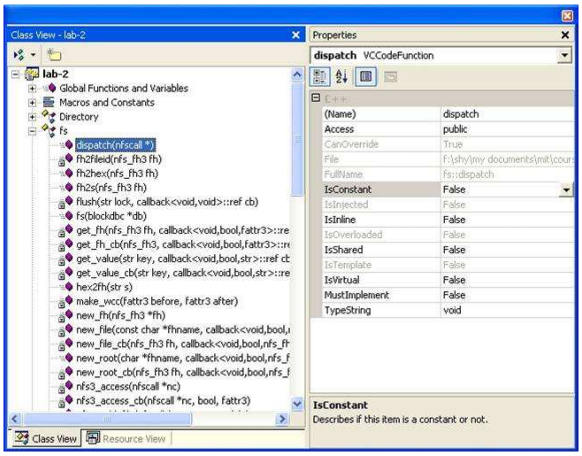

Figure 8: Example of the Visual Studio .NET editing concept. The user is editing or browsing in the left frame. On the right hand side is a property frame that shows the contents of the currently selected object (or the mutual properties of several selected objects). Each editable property provides a way to change its value (selecting from a list, opening a new window, writing text directly, ...).

in the user interface layer and the objects in the business logic layer resulting in the cross cutting modification problem mentioned in Section *5.1.1*. The information required when editing an object consists of four parts: the object data structure, the possible values for each field in this structure, the storage location of each field, and the connection between the different fields. This led us to design a dynamic object, whose structure is defined by predicates on its state. The fields carry all the information that is used to edit, save, and load them. We have also decided to adopt the Visual Studio .NET editing concept (see Figure 8). Every object that is selected has the same property page created using a PropertyGrid gadget. The PropertyGrid gadget queries the object about its fields using reflection and presents them in a readable way to the user. Having objects with similar interface greatly simplified the task of wrapping those objects with an object that can be queried by a PropertyGrid. The outcome is a very homogeneous look and feel for the user interface and greatly reduced user interface development effort.

Second, the level of uncertainty in which the development team had to work pushed us into generalizing beyond Implementation 1, which was itself highly flexible. Implementation 1's knowledge level included devices, device types, and commands. Using predicates to emulate dynamic reclassification into subclasses allowed us to move the statement hierarchy from the operational level into the knowledge level. Predicate fields also enabled us to insert device types' user interface information into the knowledge level. Moreover, controlling the entire knowledge level with predicates allows finer control over the operational level. For example, the developer can use a predicate to force the Experiment Designer to supply additional information when using a specific Scope X and a different piece of information when using Scope Y. Therefore, the augmented knowledge level is highly configurable and the level of re-usability is increased. Finally, by having each field contain all the knowledge needed to edit, load, and save it, the Knowledge Level Editor can easily use the predicates to support new devices which are combinations of the existing fields.

Third, using dynamic classification permitted us to improve the user interface by allowing the designer to change the type of edited objects without removing the old one and creating the new one. The changed objects retain all their mutual information, saving the Experiment Designer the trouble of supplying the same information repeatedly.

5.2.2 Predicate Use

In Implementation 2, dynamic types, fields, field types, and predicates, described in Section 3, were defined as part of the knowledge level in the database. When the Experiment Designer is designing the experiment, the Experiment Operator is viewing a running experiment, or the Knowledge Level Editor is modifying the system's knowledge of the physical world (Cables, Devices, ...), the system instantiates objects with the appropriate dynamic type. An object's visible state is modified (or viewed) by the appropriate user and when a change in the state causes predicates to be fulfilled or unfulfilled, the object's structure is updated.

The developers created dynamic types for all the different objects used in viewing and editing the experiments and physical structure (Devices, Devices Types, Statements, Signals, Setups, Groups, ...).

The system developers, not users, obtain the benefits of predicate fields. The Knowledge Level Editor interface was created using predicate fields and allows controlled changes to them. However, the users are oblivious to the existence of predicate fields.

6. EXAMPLE USES OF PREDICATE FIELDS

We will examine the use and effects of the predicate field's library through several examples. In Section 6.1, an Experiment Designer modifies an experiment. In Section 6.2, a developer augments the statement hierarchy with an additional field and exercises fine-grained control over the data supplied by the Experiment Designer.

6.1 Experiment Designer interaction

Using the partial definitions in Figure 9, we will observe two modifications to an existing experiment: adding a statement whose purpose is to send the automatic start command to vacuum machine #2, and replacing a manual statement with an automatic statement.

6.1.1 Creating and specializing a new object

The Experiment Designer starts the process by selecting "add new statement" from a pop-up generated from the selected statement. The newly created object satisfies only the predicate for being a statement, PStatement. Since the predicate is satisfied, two fields, `StatementType` (used to imitate sub-typing) and `name`, are added (exposed) to the dynamic object. Those fields are visible to the Experiment Designer in the user interface.

When the Experiment Designer sets the statement type to automatic statement, a new predicate `PAutomaticStatement` is satisfied and two new fields, device type and number of iterations, are added to the dynamic object (and to the statement properties form). The Experiment Designer can now set the device type to vacuum machine (chosen from a list of device types) and since a new predicate is fulfilled the `command` field is added to the statement. The designer can select the start command from the list of vacuum machine commands and vacuum machine #2 from the list of available vacuum machines, completing the construction of the statement.

6.1.2 Changing the dynamic type of an object

Suppose the Experiment Designer wishes to change a manual statement to an automatic statement in an experiment.[1] When the

```
enum StatementType {ManualStatement, AutomaticStatement};
pred PStatement dynamicType = Statement;
pred PManualStatement statementType = ManualStatement;
pred PAutomaticStatement statementType = AutomaticStatement;
pred PAnyDeviceType deviceType != null;

field StatementType statementType when@PStatement;
field string name when@Pstatement;
field DeviceType deviceType when@PAutomaticStatement;
field IntType numOfIterations when@PAutomaticStatement;
field Command command when@PAnyDeviceType;
field Device device when@PAnyDeviceType;
```

Figure 9: Definition of fields, field types, and predicates used to demonstrate specializing a statement and changing a statement's type.

Experiment Designer changes the type of the statement from manual to automatic, the system detects that the PManualStatement predicate is no longer satisfied and that the PAutomaticStatement predicate is satisfied. The system removes accessibility to all the fields connected with PManualStatement, possibly causing more predicates to be unsatisfied. Then, the system adds all the field controlled by the newly satisfied predicate PAutomaticStatement. It is important to realize that those fields which are shared by both statement types will retain their value over this exchange, even if they are eliminated and then revived. The reason is that they are saved in the same place in the database.

6.2 Developer interaction

As an example of the developer's interaction consider augmenting the statement hierarchy with the `max_repeat` field discussed in Section *5.1.1*.[2] The Experiment Designer uses this field to limit how many times a certain statement can be executed by the Experiment Operator. Three changes to the system are required.

First, the developer adds a `max_repeat` field to the knowledge level. The new field's type is `unsignedAndInfinity`(this field type allows the infinity value as well as any positive integer) and its predicate is the statement predicate. The default value for this field is infinity (for backward compatibility).

Second, the developer creates a storage location for the new value in the database. It can be an additional column in the statement table or in any table directly connected (by foreign key relation) to the statement table. Since the last two modifications are done to the knowledge level, they can even be done while an Experiment Designer is working. The Experiment Designer will see this new field and will have the option to change its value.

The third change is done to the code. The new field cannot be used in an automated way (presenting a description, sending setup to the device) but requires behavior changes. The developer modifies the execute method of the statement class (a real type) residing in the logic tier to count the number of times it has been executed, prohibiting execution after `max_repeat` times.

6.2.1 Fine-grained control

[1]We started integrating the system before all the hardware was developed. For hardware with known interface, we connected the device's commands to the null device driver until the device driver was developed. For devices with unknown interface, the Experiment Designer used manual operations as placeholders for the real operations. When the specific device type was incorporated into the system, the Experiment Designer replaced the manual statement with the appropriate automatic command.

[2]This was not real a modification in Implementation 2, as it was part of its initial design. However, this modification was a user-requested change to Implementation 1, which resulted in many cross-cutting modifications, and partly triggered the use of predicate fields.

Predicate fields support fine-grained control over objects. As an example, consider the case where developers were asked to help the users detect power failures in the facility. The request was simple: each read-results statement sent to scope #3 needed to be marked by a unique number set by the Experiment Designer. The solution was equally simple: the developers added a predicate for the specified device (device = scope #3) and a temporary field which appeared only when this predicate is satisfied (scope #3 is chosen). Avoiding setting a default value for this field forces the Experiment Designer to set a value in this field when it is part of the object.

7. EXPERIENCE

In this section, we discuss the experience gained by using predicate fields in our implementation. We offer insight into the technique's strengths and weaknesses as well as the process the developers and (affected) users went through from resentment to acceptance.

Section 7.1 discusses the effects of the library on developers using the dynamic objects. In Section 7.2 we observe the initial negative reaction of the developers to the library, and speculate on its origin. Section 7.3 demonstrates the acceptance of the library by the developers. Finally, Section 7.4 summarizes the limitations of our library and the disadvantages of using it.

7.1 Developers using the library

User interface developers had to interact with the dynamic objects. A dynamic object provides a simple interface to query the values of its fields. In addition, it exposed events for some of the fields. Those events are fired when the corresponding field's value is changed. The developer only had to instantiate the object (with the appropriate dynamic type), and possibly choose a storage location. Finally, the developer can pass the dynamic object to a wrapper object that can be used for reflection (for example passing it to PropertyGrid gadget for editing). Those developers reported that due to its simple interface the dynamic objects were easily used and the result was a significant decrease in development time. Some of the developers were even surprised by the ease and quickness of the development process.

7.2 Modifying the knowledge level

Developers adding support for new requirements and features had to modify the dynamic types (rarely), fields, field types, and predicates (more frequently). These modifications demanded understanding the interaction between the different data structures. Adding new predicates and fields and modifying existing ones was initially difficult for most of the developers. This initial difficulty can be attributed to five factors.

1. Using a declarative approach. Since using predicate fields in our library implies using a declarative approach rather than the usual procedural approach, developers had to change their way of thinking. Combining object-oriented development with the predicate-oriented approach is somewhat similar to switching from procedural languages to object-oriented ones. In addition, the declarative approach can make it harder to understand the behavior of specific components.

2. Far-reaching effects. Knowledge level modifications have far-reaching effects. This combined with the intrinsic type un-safety (objects changing their type dynamically) of the dynamic objects, occasionally caused unexpected system behavior following changes to the predicate fields. As examples, the developers observed that certain fields failed to appear where they should, or the system would report[3] many instances of the "field not found" run-time exception. The common practice was to put the blame on the predicate fields library until the actual problem was found.

3. Modifications done directly in the database. Until proper tools had been developed, knowledge level modifications were done directly into the database. A developer making the modifications had to visualize in his head the connections between the different fields, field types, and predicates. Most developers found it to be very annoying.

4. Predicate fields are implemented as a library, rather than as a language extension. Defining fields and predicates is more cumbersome, compared to regular classes. The software was more verbose and thus less readable. Finally, performance was affected by calculating predicates and accessing fields.

5. Type safety problems. Objects appear to change their type dynamically. Combined with the library implementation, static type checking is very hard. Consider the following example for type unsafety. Observe Figure 11. In our system the `name` field of `o` is accessed with the `s["name"]` command. The underlying assumption is that `o` has a field named `name`. Since this assumption can't be verified until run time, static type checking cannot be done. We propose switching to the Smalltalk solution of dynamic type safety verification by writing many tests cases. We could create a framework to allow the developer to specify assumptions about the system and objects' interaction. For example, as long as a statement has a device type field, it should also have a command field.

7.3 Developer reactions

Whereas the developers initially disliked the predicate fields library, all of them reported that once they became familiar with defining predicates and fields (usually after the first two or three predicates they defined), they were able to proceed with ease. They also found out that their perspective toward designing the user interface has changed and noted that one result of using the predicate fields was that almost no custom forms had to be built.

For example, the development team had to support a requirement for changing objects' visibility and modifiability in the different contexts — for example, prohibiting the user from making command changes when running an experiment while allowing him to make setup changes. We added contexts and information about normal and unusual field behavior in the different contexts. Using this solution it took only a couple of days to support the requirement.

Another developer had a task to design a client for the knowledge level modifications done by the Knowledge Level Editor. This client is used to add new device types into the system, and other amendments in the system's physical world knowledge. The developer suggested that using custom-made wizards would be easily implemented and the resulting user interface would be clearer. After some bitter arguments, the development team decided to implement the wizards using the predicate fields library with a slightly different version of the PropertyGrid gadget. Some developers found it to be an unexpected use of the predicate fields library. However, the implementing developer, who had initially opposed the proposal, reported that in retrospect it was much easier and more general to implement it with the library.

[3]The system contained most thrown exceptions, allowing the users to recover and continue their work in the face of certain types of errors.

Since our development methodology is incremental, we postponed the signals package development to late in the development process. Signals is a rather large package that took extensive efforts to implement in Implementation 1. A signal is the output of a measurement device and constitutes the basis for the gathering, presentation, and analysis of the results. The complexity of signals lies in automatic switching (the system should deduce switching from the user's chosen hardware) and results processing (measurement-devices results should be separated into the signals granularity, stored, analyzed, and retrieved in this level). The signals package was inserted in a relatively small effort (compared to the previous implementation) using predicate fields to control the amount of signals each measurement device supports in addition to being an integral part of the result analyzer.

7.4 Limitations

While using predicate fields, we have observed some limitations and pitfalls. Most problems result from the library syntax and would have had less impact had predicate fields been implemented as a language extension.

First, since system behavior heavily depends on meta data, changes to the meta data can have far-reaching affects. It was often the case that a careless developer made a seemingly simple change to the predicate fields definitions, only to discover that the system had become unusable. This requires careful modification, well-documented knowledge level structure, and unit testing.

Secondly, part of the resulting software can be harder to understand. Since we are using declarative definitions, it is very hard to grasp a component's full structure (even if we did not take into account dynamic structure modifications).

Thirdly, implementing predicate fields as a library rather than as a language extension (as is the case with all previously predicate-oriented programming proposals) can cause the software to be less readable. It can also incur some performance overhead when modifying the object structure and accessing fields within the object, but this overhead is negligible since dynamic objects are used primarily in the user interface.

Lastly, since types are changed dynamically, type safety cannot be guaranteed. However, developers can use the proposed testing scheme to transform their beliefs about the correlation between the object's state and its set of fields, into tests.

8. LIBRARY LOW-LEVEL IMPLEMENTATION

We now summarize the key capabilities that are provided by the predicate fields library. The dynamic object abstract class is DynamicObject. Its interface allows adding and removing fields as well as registering for the field's value change event. The extending classes include DatabaseDynamicObject, MemoryDynamicObject, and XMLDynamicObject, each loading and storing the information from a different location. The FieldsAdapter class is responsible for adding and removing fields in the dynamic object as the predicates become satisfied or unsatisfied. ObjectPresenter is a wrapper class that serves as the mediator between the dynamic object and the PropertyGrid (a .NET user interface gadget that uses reflection to provide a user interface for browsing the properties of an object). The ObjectPresenter appears to the PropertyGrid to be an object corresponding to the current contents of the relevant dynamic object. The ObjectsBinder allows the user to simultaneously edit the similar structure of a group of objects.

8.1 Library code sample

Figure 10 contains a part of the dynamic object interface. A dynamic object has the C# type `DynamicObject`. The `DynamicObject` constructor takes as an argument the dynamic type, and sets the dynamic object's type to its argument. The `DatabaseDynamicObject` implements a dynamic object that is loaded from and saved to the database. Its constructor has an additional argument, the ID of the object. This ID may be an ID of an existing object (which will be loaded from the database) or a fresh ID created by the system (indicating where to store the object). The dynamic object interface `IDynamicObject` contains a notification mechanism, which allows a using container to be notified when a field value is changed.

Figure 11 demonstrates using a dynamic object in a tree of statements. The statements are statements in the MML programming language (Mission Modeling Language, see Section 4.1), and the tree is the user interface to that programming language (see Section 4.3).

Since the MML statements are persistent in a database, the statement ID (either an existing statement to be loaded, or a new ID created by the system) is passed to the constructor of DatabaseDynamicObject. When the new dynamic object is created the predicate (PStatement) is satisfied, causing the `name` field to be added to the object. Also, when the user selects a node (in the statements tree), the tree `OnSelect` method wraps the contained dynamic object with an object of type `ObjectsPresenter`. The `OnSelect` method then passes the `ObjectsPresented` to a PropertyGrid gadget that uses reflection to present the object's fields to the user.

8.2 Field storage location

This section summarizes how our library determines the storage location for fields in a dynamic object. The dynamic objects can be loaded from and saved to persistent storage. For an object stored in the database, the table used to store it is found in the dynamic type definition. When a field is stored to or loaded from the database, the object's ID is used to find the appropriate row, and the field's column is used to find the correct column for the field's value. However, some fields have to be saved in tables other than the object's table. For example, the Knowledge Level Editor (Section 4.3) can define commands on groups of devices. Since the correlation between devices and groups is many-to-many, this correlation is saved in its own table. Therefore, when a dynamic object's device-group field needs to be loaded, the previous suggested method fails.

We solved this problem by adding information to determine storage in a table connected (by a foreign key relation) to the object table. (This seems to be enough for our purposes. In the future, it may be valuable to allow storing and loading object's fields from a table connected with a path of arbitrary length to the object table.) This information consists of two column names and a table name added to the field's definition. As shown in Figure 12, the library locates the additional row (the row in the second table, which contains the object's data) by searching the second table (found in the field's definition) for a value found in the object row (a row containing the object data in the object's main table, taken from the dynamic type).

9. RELATED WORK

Our work is the first, to our knowledge, to evaluate predicate fields or other predicate-oriented programming techniques in a substantial real-world application. However, the concepts are not new and have been developed by previous researchers.

```
// This delegate define the
// signature of an event which
// is thrown will a dynamic field
// changes its value.
public delegate void FieldChangedEventHandler
  (object sender, string fieldName,
   object oldValue, object newValue);

// The root of the Dynamic Objects inheritance tree.
// Instances of IDynamicObject may contain
// predicate fields.
public interface IDynamicObject
{
  event FieldChangedEventHandler
    FieldValueChanged;
  void AddField(string fieldName,
                object defaultValue);
  void RemoveField(string fieldName);
  bool FieldExists(string fieldName);

  object this[string fieldName] {
    get;   // use default getter
    set;   // use default setter
  }
  DynamicType dynamicType {
    get;
  }
}

// Provides implementation for common methods
// for the DynamicObjects Hierarchy
public class DynamicObject: IDynamicObject
{
  public DynamicObject(DynamicType dynamicType) {
    this.dynamicTypeId = dynamicType
  }
}

// A persistent dynamic object that is loaded
// from and saved to a database.
public class DatabaseDynamicObject : DynamicObject
{
  public DatabaseDynamicObject(int id,
    DynamicType dynamicType): base(dynamicType) {
    this.id = id;
  }
  ...
}
```

Figure 10: C# code for the dynamic object interface of the predicate fields library.

Most work on enhancing object-oriented languages with predicates has focused on the dispatch mechanism, which gives a way for different objects to respond differently to a particular method invocation; examples include predicate dispatching and predicate classes. By contrast, predicate fields give a way for different objects to have different fields at run time: the structure, not the behavior, of objects is dictated by run-time values. Predicate fields can be used to simulate aspects of predicate classes and predicate dispatch, and that has been a common use for them (including in our case study).

Predicate classes [8] support a form of automatic, dynamic classification of objects. They provide the functionality of predicate fields; additionally, they permit methods to specialize on the identity or state of an argument, in addition to the receiver's type. An object has the type of a predicate class if the object's state satisfies the class's predicate. A predicate is an arbitrary boolean expression, including subtype checks that permit emulating single or multiple dynamic dispatch. As in our implementation, objects reserve space for any fields that might be inherited from a predicate object, and the value in such a field persists even when the controlling predicate evaluates to false and the field is inaccessible.

```
// A tree of MML programming language statements
// is displayed in the MML user interface.
Class StatementsTreeView : TreeView
{
  // Create a dynamic object for a statement
  // and connect it to the appropriate node.
  public newStatement(TreeNode node,
                      int statementId){
    // In the stylized syntax of figures 1-6, as
    // would be used in a language-based
    // implementation of predicate fields, the
    // body of this method would be:
    //   IDynamicObject o =
    //     new Statement(statementId);
    //   node.tag = o;
    //   o.name = "new statement";
    //   o.FieldValueChanged +=
    //     new FieldChangedEventHandler(Refresh);
    IDynamicObject o =
      new DatabaseDynamicObject(statementId,
                                Statement);
    // Connect the object to its node in the UI.
    node.tag = o;
    o["name"] = "new statement";
    o.FieldValueChanged +=
      new FieldChangedEventHandler(Refresh);
  }
  public event Refresh(object sender,
                       string fieldName,
                       object oldValue,
                       object newValue){
    ... // Refresh the tree view in the UI.
  }
  // This method is called when the
  // user selects one of the tree nodes.
  public OnSelect(TreeNode n){
    // Present the selected statement to the user.
    propertyGrid.selected =
      new ObjectsPresenter(n.tag);
  }
  PropertyGrid properyGrid;
}
```

Figure 11: Using a dynamic object in the user interface development. This code demonstrate the creation of the statements dynamic objects in a tree of statements.

Classifiers in Kea [17, 19, 23] are a previous, more limited mechanism. Kea classifiers automatically determine the effective type of an object, which can affect dispatching. However, Kea is a functional language, so no dynamic reclassification ever occurs. (The same is true of Views [32], which offer different observer methods on a single unchanging object.) Furthermore, Kea classifiers are determined by the value of a single enumerated-type attribute, or by explicit instantiation of a particular subclass. Our implementation is similar, but permits classification based on the value of a field of arbitrary primitive or string type. CLOS [2] and Dylan [1] are even more limited: dispatching can depend on the identity of an argument, but cannot easily depend on a more general condition of an argument. Objects in the Self language [31] have an explicit parent pointer that can be reassigned at run time, permitting the effective superclass of an object to change.

Other research focuses on making dispatching more efficient without affecting the object's representation. Multiple dispatch (also called multi-method dispatch) [3, 7, 12, 11] permits dispatching to depend on the run-time classes of multiple arguments, not just the

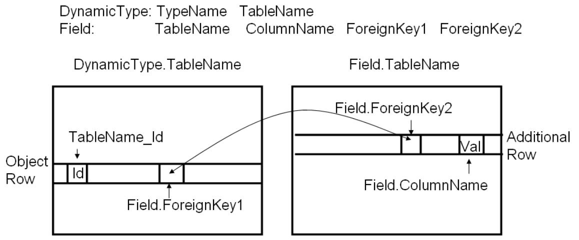

Figure 12: Finding a field's database location, using the table information from the dynamic type and the ID of the object, in addition to the foreign keys and the column name in the field's definition.

receiver. Predicate dispatching [13, 9, 30, 26, 22] permits arbitrary boolean predicates to control dynamic dispatch; logical implication between predicates is the overriding relationship. Predicate dispatching generalizes object-oriented single and multiple dispatch, ML-style pattern matching, predicate classes, and classifiers; however, it does not directly support predicate fields. Millstein [22] gives the most extensive previous evaluation of predicate-oriented programming, showing that predicate dispatching can reduce an 800-line program to 700 lines. The work of Millstein et al. [11, 22] also gives a modular static type system for an expressive language; other approaches, including our own, rely on dynamic type checking, which has the disadvantage of revealing errors too late.

10. CONCLUSIONS

We have presented an implementation of predicate fields as a library, and a case study in which the predicate fields where used pervasively in a 100,000-line system. Developers found predicate fields useful in practice. The library is used successfully and intensively in a project with very tight deadlines. This case study provides concrete evidence that other practitioners should try predicate fields, and that researchers should continue to refine designs and implementations.

In the case study, developers used predicate fields to support greater software flexibility, both for themselves and for power users (Knowledge Level Editors). Predicate fields gave developers greater control over objects, allowed fined-grained modifications to specific fields, and provided a declarative approach to defining object structure, which permitted changes to be easily reflected throughout the system.

Predicate fields were particularly useful in providing a uniform interface to an arbitrary (and dynamically changing) number of possible object structures, which was achieved through a combination of reflection and treating dynamic objects (created by the library) as real C# objects. Our experience suggests that predicate fields may be particularly useful in projects with an extensive user interface which contains many objects organized hierarchically. Furthermore, predicate fields are a good match for projects in which requirements (and persistent objects) change frequently, because they permit both the code and the objects themselves to be easily adapted to new circumstances.

Using predicate fields, developers were able to model real-world objects (which appear to have different attributes in different circumstances, or whose structure changes over time) in a simple and natural fashion. The alternative — employing complex design patterns to capture this information — is less attractive.

Our predicate fields implementation has a number of limitations: it is implemented as a library rather than integrated with the programming language; it does not support predicate dispatching or predicate classes; the syntax for predicates is very limited; and it uses dynamic type-checking, giving the possibility of run-time type errors. Despite these limitations, the implementation was sufficiently powerful to solve the problems encountered in building the industrial control system, and the developers did not feel that (for example) lack of support for dynamic behavior modification of objects (predicate dispatch) was a significant hindrance. This result is suggestive regarding how much linguistic power and complexity is desirable in practice.

The case study also revealed some potential downsides of using predicate fields, some of which are consequences of the implementation and some of which are inherent to the use of predicate fields. Language and tools support are important in reaping the potential benefits of predicate fields. For instance, it would have been better to generate most of the user interface automatically, and better syntax checking would have been welcome. The declarative nature of the predicates was useful in propagating changes throughout the system, but the same characteristic (non-local control) could make component behavior difficult to understand. The biggest remaining problem is lack of static type safety. In the absence of integration into a widely accepted programming language implementation (a distant prospect), we have proposed a testing mechanism that may mitigate this problem.

Acknowledgments

Our research would not have been possible without the assistance of Shoshana Avraham who contributed many design ideas for the industrial control system. We also thank Galia Shlezinger, Carmit Zadok, Meister Lior, and Dafna Folkman for the development of Implementation 2. We used Tony Allowatt's "Bending the .NET PropertyGrid to Your Will" (`http://www.codeproject.com/cs/miscctrl/bending_property.asp`) when creating wrapper objects.

11. REFERENCES

[1] Apple Computer. *Dylan, an Object-Oriented Dynamic Language*, 1992.

[2] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *ACM SIGPLAN Notices*, 23(SI):1–145, 1988. Special issue: X3J13 Document 88-002R.

[3] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. CommonLoops: Merging Lisp and object-oriented programming. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 17–29, Portland, OR, USA, June 1986.

[4] D. Box. *Essential COM*. Addison-Wesley Professional, Boston, MA, 1997.

[5] E. Brown. *Windows Forms Programming in C#*. Manning Publications, Greenwich, CT, 2002.

[6] K. Brown, T. Ewald, C. Sells, and D. Box. *Effective COM: 50 Ways to Improve Your COM and MTS-based Applications*. Addison-Wesley Professional, Boston, MA, 1999.

[7] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92, the 6th European Conference on Object-Oriented Programming*, Utrecht, The Netherlands, June 29–July 3, 1992.

[8] C. Chambers. Predicate classes. In *ECOOP '93, the 7th European Conference on Object-Oriented Programming*, pages 268–296, Kaiserslautern, Germany, July 28–30, 1993.

[9] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *Object-Oriented Programming Systems,*

Languages, and Applications (OOPSLA '99), pages 238–255, Denver, Colorado, Nov. 3–5, 1999.

[10] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Seattle, WA, 1996.

[11] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 130–145, Minneapolis, MN, USA, Oct. 15–19, 2000.

[12] E. Dujardin, E. Amiel, and E. Simon. Fast algorithms for compressed multimethod dispatch table generation. *ACM Transactions on Programming Languages and Systems*, 20(1):116–165, Jan. 1998.

[13] M. D. Ernst, C. S. Kaplan, and C. Chambers. Predicate dispatching: A unified theory of dispatch. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 186–211, Brussels, Belgium, July 20-24, 1998.

[14] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[15] Geotest – Marvin Test Systems, Inc, CA. *ATEasy 3.0 Reference Manual*, 2000.

[16] J. Glynn, C. Torok, R. Conway, W. Choudhury, Z. Greenvoss, S. Kulkarniq, and N. Whitlow. *Professional Windows GUI Programming Using C#*. Apress, Berkeley, CA, 2002.

[17] J. Hamer, J. Hosking, and W. Mugridge. A method for integrating classification within an object-oriented environment. Technical Report 48, Department of Computer Science, University of Auckland, Oct. 1990.

[18] E. Harmon. *Delphi COM Programming*. New Riders Publishing, Indianapolis, IN, 2000.

[19] J. Hosking, J. Hamer, and W. Mugridge. Integrating functional and object-oriented programming. In *Technology of Object-Oriented Languages and Systems TOOLS 3*, pages 345–355, Sydney, 1990.

[20] M. MacDonald. *User Interfaces in C#: Windows Forms and Custom Controls*. Apress, Berkeley, CA, 2002.

[21] M. MacDonald. *Microsoft .NET Distributed Applications: Integrating XML Web Services and .NET Remoting*. Microsoft Press, Seattle, WA, 2003.

[22] T. Millstein. Practical predicate dispatch. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 345–364, Vancouver, BC, Canada, Oct. 26–28, 2004.

[23] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In *ECOOP '91, the 5th European Conference on Object-Oriented Programming*, pages 307–324, Geneva, Switzerland, July 17–193, 1991.

[24] B. A. Myers and M. B. Rosson. Survey on user interface programming. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 195–202. ACM Press, 1992.

[25] C. Nock. *Data Access Patterns: Database Interactions in Object-Oriented Applications*. Addison-Wesley Professional, Boston, MA, 2003.

[26] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, pages 56–64, Enschede, The Netherlands, Apr. 24-26, 2002.

[27] I. Rammer. *Advanced .NET Remoting (C# Edition)*. Apress, Berkeley, CA, 2002.

[28] A. Taivalsaari. Object-oriented programming with modes. *Journal of Object-Oriented Programming*, pages 25–32, June 1993.

[29] S. Teixeira and X. Pacheco. *Delphi 5 Developer's Guide*. Sams, Scotts Valley, CA, 1999.

[30] A. Ucko. Predicate dispatching in the Common Lisp Object System. Technical Report 2001-006, MIT Artificial Intelligence Laboratory, Cambridge, MA, June 2001.

[31] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 227–242, Orlando, FL, USA, Oct. 1987.

[32] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 307–313, Munich, Germany, Jan. 1987.