Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study

DON BATORY, CLAY JOHNSON, BOB MACDONALD, and DALE VON HEEDER
University of Texas at Austin

This is a case study in the use of *product-line architectures* (*PLAs*) and *domain-specific languages* (*DSLs*) to design an extensible command-and-control simulator for Army fire support. The reusable components of our PLA are layers or "aspects" whose addition or removal simultaneously impacts the source code of multiple objects in multiple, distributed programs. The complexity of our component specifications is substantially reduced by using a DSL for defining and refining state machines, abstractions that are fundamental to simulators. We present preliminary results that show how our PLA and DSL synergistically produce a more flexible way of implementing state-machine-based simulators than is possible with a pure Java implementation.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies* (e.g., object-oriented, structured); D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Evolutionary prototyping*; *State diagrams*; D.2.10 [**Software Engineering**]: Design—*Methodologies and representations*; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specfic architectures*; *Languages*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*; D.2.m [**Software Engineering**]: Miscellaneous—*Rapid prototyping*

General Term: Design

Additional Key Words and Phrases: GenVoca, domain-specific languages, simulation, aspects, refinements

1. INTRODUCTION

Software evolution is a costly yet unavoidable consequence of a successful application. Evolution occurs when new features are added and existing capabilities

This work was supported in part by the University of Texas Applied Research Labs and the U.S. Department of Defense Advanced Research Projects Agency in cooperation with the U.S. Wright Laboratory Avionics Directorate under contract F33615-91C-1788.
Preliminary version presented at the International Conference on Software Reuse, Vienna, Austria, July 2000. Updated for ACM TOSEM September 2001.
Authors' addresses: D. Batory, Department of Computer Sciences, University of Texas at Austin, Austin, TX 78712; email: batory@cs.utexas.edu; C. Johnson, B. MacDonald, and D. von Herder, Applied Research Labs, University of Texas at Austin, Austin Texas 78713; email: {clay, bob, drv}@arlut.utexas.edu.

are enhanced. Unfortunately, many applications suffer *design fatigue*—when further evolution is difficult and costly because of issues not addressed in the initial design [Graves 2001]. Creating software that is easily evolvable is a central problem today in software engineering.

Three of several proposed complementary technologies address software evolution: object-oriented design patterns, domain-specific languages, and product-line architectures. *Design patterns* are techniques for restructuring and generalizing object-oriented software [Gamma et al. 1995]. Evolution occurs by applying design patterns to an existing design; the effects of these changes are borne by programmers who must manually transform an existing code base to match the updated design. Recent advances indicate that tool support for automating the applications of patterns is possible [Tokuda and Batory 1999]. *Domain-specific languages* (*DSLs*) raise the level of programming to allow customized applications to be specified compactly in terms of domain concepts; compilers translate DSL specifications into source code. Evolution is achieved by modifying DSL specifications [Van Deursen and Klint 1997]. *Product-line architectures* (*PLAs*) are designs for families of related applications; application construction is accomplished by composing reusable components. Evolution occurs by plugging and unplugging components that encapsulate new and enhanced features [Batory 1998; Bosch 1999; Czarnecki and Eisenecker 1999; Software Engineering Institute 2001; Weiss and Lai 1999]. Among PLA models, the GenVoca model is distinguished by an integration of ideas from aspect-oriented programming [Kiczales et al. 1997], parameterized programming [Goguen 1986], and program-construction by refinement [Baxter 1992].

This paper presents a case study in the use of GenVoca PLAs and DSLs to create an extensible command-and-control simulator for Army fire support. (Design patterns were also used, but they played a minor role.) We discovered that components of distributed simulations are *not* conventional DCOM and CORBA components, but rather are layers or "aspects" whose addition or removal simultaneously impacts the source code of *multiple*, distributed programs. Further, we found that writing our components in a general-purpose programming language (Java) resulted in complex code that obscured a relatively simple, state-machine-based design. By extending Java with domain-specific abstractions (in our case, state-machines), our component specifications were more readily understood by domain experts, knowledge engineers, and application programmers. Thus, this case study is interesting not only because of the novelties introduced by PLAs and DSLs, but also because of their integration: using only one technology would have been inadequate.

We begin by explaining the ideas and terminology of fire support. We review an existing simulator, called FSATS, and motivate its redesign. We present a GenVoca PLA for creating extensible fire-support simulators and introduce an extension to the Java language for defining and refining state-machines. Finally, based on simple measures of program complexity, we show how PLAs and DSLs individually simplify simulators, but only their combination provides practical extensibility.

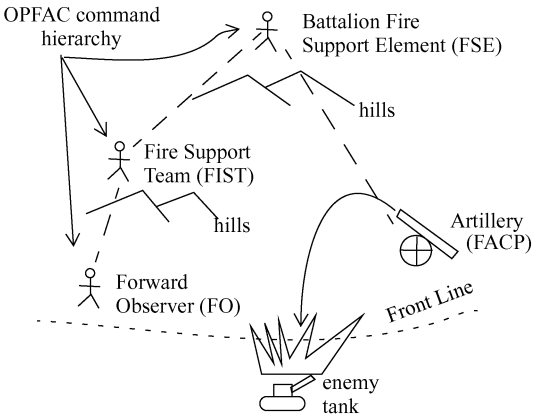


Fig. 1. OPFAC command hierarchy.

2. BACKGROUND

2.1 The Domain of Fire Support

Fire support is a command-and-control application that includes the detection of targets, assignment of weapons to attack the target, and coordination of the actual attack. The entities engaged in this process, called *operational facilities* (*OPFACs*), are soldier-operated (not machine-automated) command posts that exchange tactical (theater-of-war) messages.

Forward observers (*FO*) are OPFACs that are stationed at intervals across the frontline of a battlefield (Figure 1). They are one of several kinds of sensors responsible for detecting potential targets. A hierarchy of *fire support elements* (*FSE*) is responsible for directing requests from FOs to the most appropriate weapon system to handle the attack. FOs report to their *fire support team* (*FIST*); a FIST reports to a battalion FSE, a battalion FSE reports to a brigade FSE, and so on. Each FSE typically has one or more supporting *command posts* (*CPs*) with different weapon systems. For example, a battalion FSE might be supported by a *field artillery command post* (*FACP*); a FIST might be supported a mortar command post, and so on. In general, higher echelon FSEs are supported by higher echelon CPs with more powerful and/or longer range weapon systems.

FOs, FISTs, and other FSEs are responsible for evaluating a target. An evaluation may result in (a) assigning the target to be attacked by a supporting weapon, (b) elevating the target to the next higher echelon FSE for evaluation, or (c) denial—choosing not to attack the target. CPs are responsible for assigning targets to the best weapon or combination of weapons under their command. Once weapon(s) are assigned, messages are exchanged with the mission originator (usually an FO) to coordinate the completion of the mission. The particular message sequence depends on the target and weapon. It is still generally the case that all messages are relayed along the chain of CPs and FSEs that were involved in initiating the mission, although newer systems permit

messages to be exchanged directly between the weapon and observer. The message sequence for a particular mission is referred to as the *mission thread*. In general, an OPFAC can participate in any number of mission threads at one time.

A mission thread is an instance of a *mission type*. There are well over twenty mission types, including:

- *when-ready-fire-for-effect-mortars* (*WRFFE-mortars*)—a mortar CP is assigned to shoot at a target as soon as possible,
- *when-ready-fire-for-effect-artillery* (*WRFFE-artillery*)—one or more artillery CPs are assigned to shoot at a target as soon as possible,
- *time-on-target-artillery* (*TOT-artillery*)—field artillery are requested to fire at a target so that all rounds land at the specified location at the specified time, and
- *when-ready-adjust-mortars* (*WRAdjust-mortars*)—a forward observer knows only approximately the location of the enemy and requests single rounds to be fired with the observer sending corrections between rounds until the target is hit, at which point it becomes a WRFFE-mortar mission.

Each OPFAC (FO, FIST, FSE, etc.) performs different actions for each mission type. For example, the actions taken by an FO for a TOT-mortar mission are different than those for a WRFFE-artillery mission.

Clearly, the above description of fire support is highly simplified, for example, the actions taken by specific OPFACs in a mission thread and the translation of messages into formats for tactical transmission were omitted. These details, however, are not needed to understand the contributions of this paper.

2.1.1 *FSATS.* Simulation plays a key role in U.S. Army testing and training. It avoids costs of mobilizing live forces, provides repeatability in testing, and allows force-on-force combat training without the liability. Simulation has been used to model virtual environments, weapons effects, outcome adjudication, and as computational resources increase, the fidelity has been refined to entity-level simulators.

Fire support is one of a number of domains that has been modernized by digital *Command*, *Control*, *Communications*, *Computer*, *and Intelligence* (*C4I*) systems that automate battlefield mission processing. *AFATDS* (*Advanced Field Artillery Tactical Data System*) is arguably the most sophisticated C4I system in existence, and provides the software backbone (message transmission, processing, etc.) for fire support for the Army [Magnavox 1999]. *FSATS* (*Fire Support Automated Test System*) is a system for testing AFATDS and other fire-support C4I systems. FSATS collects digital message traffic from command and control communication networks, interprets these messages, and stores them in a database for later analysis. FSATS can simulate any or all OPFACs used in AFATDS [FSATS 1999]. The subject of a test can be overall system performance, individual OPFAC performance, or system operator performance. Thus, FSATS is used both in training Army personnel in fire support and debugging/testing AFATDS.

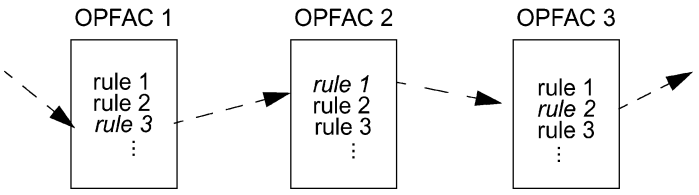


Fig. 2. Rule sets vs. mission threads.

2.1.2 *The Current FSATS Implementation.* FSATS has been under development for almost ten years. It is typical of the systems mentioned in our introductory paragraph: it began with a clean design but as its capabilities were extended, limitations of that design became increasingly troublesome.

The implementation is a combination of decision rules encoded in database tables, a set of "common actions" written as Ada procedures, and a decision rule interpreter, also in Ada. One set of rules is associated with each pair of an OPFAC type and a message type. When a tactical message is received by an OPFAC, the appropriate rule set is selected by the interpreter and each rule in the set is sequentially evaluated until one succeeds, at which point the action for that rule is executed and processing of that message terminates. There are from 200 to 1000 rules associated with each OPFAC type, divided among the various input message types. Each rule consists of a predicate, which is a conjunction of guards, and an action which is an index to a sequence of state and message common actions. Predicates typically contain five to ten guards (terms). The processing of rule sets is optimized, so that predicates can assume the failure of all previous predicates. Common actions range from simple (copy the target number field from the input to the output message) to complex (determining whether there exists a supporting OPFAC of type mortar which is capable of shooting the target indicated by the current message).

There are now obvious drawbacks to this design/implementation. While rule sets are used to express OPFAC behavior, OPFAC behavior is routinely understood and analyzed in terms of mission threads. Figure 2 illustrates a mission thread, the horizontal execution path, that associates various rules spanning multiple OPFAC programs. This complicates the knowledge acquisition and engineering process to derive from an analysis of multiple mission threads the rules as they apply at each OPFAC. Conversely, it obfuscates analyzing and debugging system behavior where rules for multiple mission threads are merged into monolithic sets within each OPFAC program.

The contrast of the vertical nature of rule sets versus the horizontal or "crosscutting" nature of mission threads in Figure 2 illustrates an encapsulation dichotomy that is not unique to FSATS [Batory and O'Malley 1992; Kiczales et al. 1997; Reenskaug et al. 1992]. In general, conventional OO approaches explore *use cases* (threads) for specification and analysis of system behavior. However, the concept of a use case is transient in a design process that identifies behavior (rules) with the actors (OPFACs) rather than the actions (missions). This trade-off is seemingly unavoidable given the need to produce objects that combine behaviors to react to a variety of situations. In FSATS, the transformation

of mission threads into rule sets yields autonomous OPFACs at an increased cost to analysis and maintenance.

As FSATS evolved, rule sets quickly became large and unwieldy. Moreover, different missions might use the same message type at an OPFAC for slightly different purposes. Simpler rules that once sufficed often had to be factored to disambiguate their applicability to newer, more specialized missions. In worse cases, large subsets of rules had to be duplicated, resulting in a dramatic increase in rules and interactions. Moreover, the relationship between rules of different OPFACs, and the missions to which they applied, was lost. Modifying OPFAC rules became perilous without laborious analysis to rediscover and reassess those dependencies. The combinatorial effect of rule set interactions made analysis increasingly difficult and time-consuming.

FSATS management realized that the current implementation was not sustainable in the long term, and a new approach was sought. FSATS would continue to evolve through the addition of new mission types and by varying the behavior of an OPFAC or mission to accommodate doctrinal differences over time or between different branches of the military. Thus, the clear need for extensible simulators was envisioned. The primary goals of a redesign were to:

- disentangle the logic implementing different mission types to make implementation and testing of a mission independent of existing missions,
- reduce the "conceptual distance" from logic specification to its implementation so that implementations are easily traced back to requirements and verified, and
- allow convenient switching of mission implementations to accommodate requirements from different users and to experiment with new approaches.

2.2 GenVoca

The technology chosen to address problems identified in the first-generation FSATS simulator was a GenVoca PLA implemented using the *Jakarta Tool Suite* (*JTS*) [Batory et al. 1998]. In this section, we motivate and explain basic ideas of GenVoca and one of its implementation techniques. It is beyond the scope of this paper to review design methodologies (i.e., how to apply GenVoca concepts) or to explain domains simpler than FSATS to elaborate the approach that we have taken. Interested readers should consult [Smaragdakis and Batory 2002], [Batory et al. 1995], and [Lopez-Herrejon and Batory 2001].

2.2.1 *Motivation.* Today's models of software are too low-level, exposing classes, methods, and objects as the focal point of discourse in software design and implementation. This makes it difficult, if not impossible, to reason about software architectures (a.k.a. component-based designs), to have simple, elegant, and easy to understand specifications of applications, and to be able to create and critique software designs automatically, given a set of high-level requirements.

Simple specifications that are amenable to automated reasoning, code generation, and analysis, are indeed possible provided that the focus of discourse can be shifted to components that encapsulate the implementation of individual and

largely orthogonal *features* that can be shared by multiple applications.[1] The intuitive rationale for this shift is evident in discussions about software products: architects don't speak about their products in terms of code modules, but instead explain their products in terms of features offered to clients. That is, the focus of discourse is on features and not on source code. GenVoca aims to raise the level of abstraction of understanding software from code modules (or code-encapsulation technologies) to features (or feature-encapsulation technologies).

2.2.2 *Features and Refinements.* At its core, GenVoca is a design methodology for creating product-lines and building architecturally-extensible software—software that is extensible via component additions and removals. GenVoca is a scalable outgrowth of an old and practitioner-ignored methodology called *step-wise refinement*, which advocates that efficient programs can be created by revealing implementation details in a progressive manner. Traditional work on step-wise refinement focussed on microscopic program refinements (e.g., `x + 0` $\Rightarrow$ `x`), for which one had to apply hundreds or thousands of refinements to yield admittedly small programs. While the approach is fundamental, and industrial infrastructures are on the horizon [Baxter 1992; Simonyi 1995], GenVoca extends step-wise refinement by scaling refinements to a component or layer (i.e., multi-class-modularization) granularity, so that each refinement adds a feature to a program, and composing a few refinements yields an entire application.

The critical shift to understand software in this manner is to recognize that programs are *values*, and that refinements are *functions* that add features to programs. Consider the following constants (parameterless functions) that represent programs with different features:

```
f ()    //program with feature f
g ()    //program with feature g
```

A *refinement* is a function that takes a program as input and produces a refined (or feature-augmented) program as output:

```
i(x)    //adds feature i to program x
j(x)    //adds feature j to program x
```

It follows that a multi-featured application is specified by an *equation* that is a named composition of functions, and that different equations define a family of applications, such as:

```
app1 = i(f());        //app1 has features i and f
app2 = j(g());        //app2 has features j and g
app3 = i(j(f()));     //app3 has features i, j, and f
```

Thus, by casually inspecting an equation, one can readily determine features of an application.

[1]Griss [2000] defines a *feature* as a product characteristic that users and customers view as important in describing and distinguishing members of a product-line.

Note that there is a subtle but important confluence of ideas in this model: a function represents both a feature *and* its implementation. Thus, there can be different functions that offer different implementations of the *same* feature:

```
k1(x)    //adds feature k (with implementation1) to x
k2(x)    //adds feature k (with implementation2) to x
```

So when an application requires the use of feature k, it becomes a problem of *equation optimization* to determine which implementation of k would be the best (e.g., provide the best performance).[2] It is possible to automatically design software (i.e., produce an equation that optimizes some qualitative criteria) given a set of declarative constraints for a target application. An example of this kind of automated reasoning is presented in Batory et al. [2000].

As a practical matter, refinements typically cannot transform arbitrary programs. Rather, the input to refinements (functions) must satisfy a *type*—a set of constraints that are both syntactic and semantic in nature. A typical syntactic constraint is that a program must implement a set of well-defined Java interfaces; a typical semantic constraint is that the implementation of these interfaces satisfy certain behavioral properties. Thus, it is common that not all combinations of features (or their implementations) are correct [Kang et al. 1990]. A model for expressing program types and algorithms that can automatically and efficiently validate equations has been developed and is part of the Jakarta Tool Suite [Batory and Geraci 1997].

2.2.3 *Mixin-Layer Implementation.* There are many ways in which to implement refinements, ranging from dynamically composing objects to statically-composed meta-programs (i.e., programs that generate other programs) [Batory et al. 1998] and rule-sets of program transformation systems [Neighbors 1997]. One of the simplest is to use templates called *mixin-layers*. In the following, we use the term *component* to denote a mixin-layer implementation of a refinement.

A GenVoca component typically encapsulates multiple classes. Figure 3a depicts component X with four classes A–D. Any number of relationships can exist among these classes; Figure 3a shows only inheritance relationships. That is, B and C are subclasses of A, while D has no inheritance relationship with A–C.

The concept of refinement is an integral part of object-orientation. In particular, a subclass is a *refinement* of its superclass: it adds new data members, methods, and/or overrides existing methods. A *GenVoca refinement* scales inheritance to simultaneously refine multiple classes.[3] Figure 3b depicts a component Y that encapsulates three refining classes (A, B, and D) and an additional class (E). Note that the refining classes (A, B, D) do not have their superclasses

[2]Technically, different equations represent different programs. Equation optimization is over the space of semantically equivalent programs. This is identical to relational query optimization: a query is initially represented by a relational algebra expression, and this expression is optimized. Each expression represents a different, but semantically equivalent, query-evaluation program as the original expression.

[3]There are other kinds of refinements beyond those discussed in this paper. An example is an optimizing refinement, which maps an inefficient program to an efficient program [Neighbors 1997].

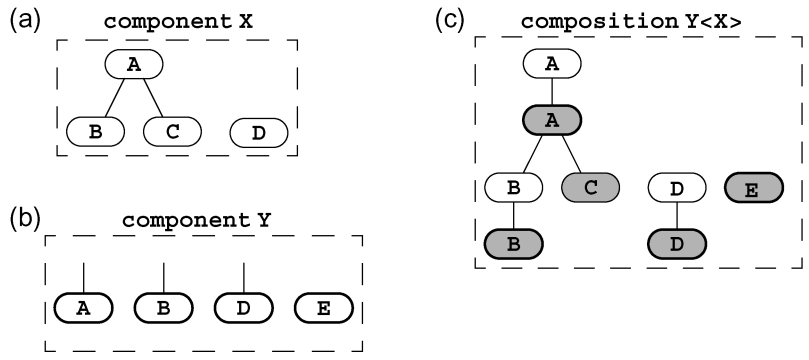


Fig. 3. GenVoca components and their composition.

specified; this enables them to be "plugged" underneath their yet-to-be-determined superclasses.[4]

In our model, where refinements are functions, we would write the composition of `Y` with `X` as `Y(X)`. When dealing with template implementations, however, the convention is to use a slightly different syntax, `Y<X>`. Thus, there is a trivial correspondence between model equations and their implementing mixin-layer template expressions.

Given this correspondence, Figure 3c shows the result of `Y<X>`. (The classes of `Y` are outlined in darker ovals to distinguish them from classes of `X`). Note that the obvious thing happens to classes `A`, `B`, and `D` of component `X`—they are refined by classes in `Y`, as expected. That is, a linear inheritance *refinement chain* is created, with the original definition (from `X`) at the top of the chain, and the most recent refinement (from `Y`) at the bottom. As more components are composed, the inheritance hierarchies that are produced get progressively broader (as new classes are added) and deeper (as existing classes are refined). As a rule, only the bottom-most class of a refinement chain is instantiated and subclassed to form other distinct chains. (These are indicated by the shaded classes of Figure 3c). The reason is that these classes contain all of the "features" or "aspects" that were added by higher classes in the chain. These higher classes simply represent intermediate derivations of the bottom class [Batory et al. 1998; Findler and Flatt 1998; Smaragdakis and Batory 1998]. A consequence of instantiating the "bottom-most" class of a chain is that refinement relationships take precedence over typical subclassing relationships. That is, if class A in component `X` is refined, it is the *most refined* version of A that is the superclass of B. This precedence relationship can be seen in Figure 3c.

Representation. A GenVoca component/refinement is encoded in JTS as a class with nested classes. A representation of component `X` of Figure 3a is shown below, where `$TEqn.A` denotes the most refined version of class `A` (e.g.,

[4]More accurately, a *refinement* of class `A` is a subclass of `A` with name `A`. Normally, subclasses must have names distinct from their superclass, but not so here. The idea is to graft on as many refinements to a class as necessary—forming a linear "refinement" chain—to synthesize the actual version of `A` that is to be used. Subclasses with names distinct from their superclass define entirely new classes (such as `B` and `C` above), which can subsequently be refined.

classes X.B and X.C in Figure 3a have $TEqn.A as their superclass). We use the Java technique of defining properties via empty interfaces; interface F is used to indicate the type of component X:

```
interface F { }    // empty

class X implements F {
  class A {...}
  class B extends $TEqn.A {...}
  class C extends $TEqn.A {...}
  class D {...}
}
```

Components like Y that encapsulate refinements are expressed as mixins—classes whose superclass is specified via a parameter. A representation of Y is a *mixin-layer* [Findler and Flatt 1998; Smaragdakis and Batory 1998, 2002], where Y's parameter s can be instantiated by any component that is of type F:

```
class Y < F s > extends s implements F {
  class A extends s.A {...}
  class B extends s.B {...}
  class D extends s.D {...}
  class E {...}
}
```

In the parlance of the model of Section 2.2.2, X is a value of type F, and Y is a function with a parameter s of type F that returns a refined program of type F. The composition of Y with X, depicted in Figure 3c, is expressed by:

```
class MyExample extends Y<X>;
```

where $TEqn is replaced by MyExample in the instantiated bodies of X and Y. Readers familiar with earlier descriptions of the GenVoca model will recognize that F corresponds to a realm interface,[5] X and Y are components of realm F, and MyExample is a type equation [Batory and O'Malley 1992]. Extensibility is achieved by adding and removing mixin-layers from applications; product-line applications are defined by different compositions of mixin-layers.

2.2.4 *Perspective.* Stepwise refinement originated in the late-1960 writings of Wirth and Dijkstra. The key to its modernization lies in scaling the effects of individual refinements, to which there are many contributors. Neighbors [1989] first described the architectural organization of mapping from abstract to concrete languages in DRACO, where the mappings between

[5]Technically, a realm interface would not be empty, but would specify class interfaces and their methods. That is, a realm interface would include nested interfaces of the classes that a component of that realm should implement. Thus, nested class A of Y would extend s.A as above, but also might implement F.IA, a particular nested interface of F. Java (and current JTS extensions of Java) do not enforce that class interfaces be implemented when interface declarations are nested [Smaragdakis and Batory 1998]. On going research aims to correct this situation [Cardone and Lin 2001].

a higher (more abstract) language representation of a program to a lower (more implementation-oriented) representation can be seen as large-scale refinements. Parameterized programming, which provides the conceptual infrastructure for early models on parametric components, was advanced by Goguen [1986]. The earliest use of plug-compatible layers (i.e., large-scale refinements) for creating product families and extensible applications originated in the mid-to-late 1980s in the work of Batory and O'Malley [1992]. Feature descriptions of applications and product-lines originated in the early 1990s with Kang's [1990] FODA (Feature Oriented Domain Analysis) and Gomaa's EDLC (Evolutionary Domain Life Cycle) [Gomaa et al. 1992] models. Collaborations, as object-oriented representations of refinements, were discussed by Reenskaug in 1992 [Reenskaug et al. 1992]. Kiczales's notion of aspects with "cross-cutting" effects clarified the general need for feature encapsulations [Kiczales et al. 1997]. Recent work on multi-dimensional separation of concerns examines a more flexible way of identifying and composing features in existing software [Tarr et al. 1999].

It is also worth noting the trade-off between the large-scale refinements of GenVoca and generic small-scale (or microscopic) refinements ($x + 0 \Rightarrow x$) that are more commonly found in the literature (e.g., [Rich and Waters 1992]).

The traditional argument for small-scale refinements is that a relatively small number of generic small-scale refinements can generate a larger number of large-scale refinements. Additionally, large-scale refinements tend to be applicable less often, because they tend to make more assumptions about the application context. (That is, the refinement `Y` of the Figure 3 is applicable less often than a "sub-refinement" that only specializes `A`, because `Y` requires the presence of `B` and `C`.) Where the case for traditional small-scale refinements breaks down is precisely when doing domain-specific development; the generation argument fails because hardly any of the generic transforms are of interest in a restricted domain, and the contextual assumptions argument breaks down because the domain provides the required context.

Domain-specific small-scale refinements can indeed be used to address the above-cited deficiencies. But, as we mentioned earlier, enormous numbers of domain-specific small-scale refinements must be applied to produce admittedly small programs. Scaling refinements, as we are doing, provides a more practical way to develop complex, domain-specific software artifacts. The tools are simpler, and the concepts are closer to main-stream programming methodologies (e.g., OO collaborations, as we will see in the next section).

3. THE IMPLEMENTATION

The GenVoca-FSATS design was implemented using the *Jakarta Tool Suite* (*JTS*) [Batory et al. 1998], a set of Java-based tools for creating product-line architectures and compilers for extensible Java languages. The following sections outline the essential concepts of our JTS implementation.

3.1 A Design for an Extensible Fire-Support Simulator

The Design. The key idea behind the GenVoca-FSATS design is the encapsulation of individual mission types as components. That is, the central

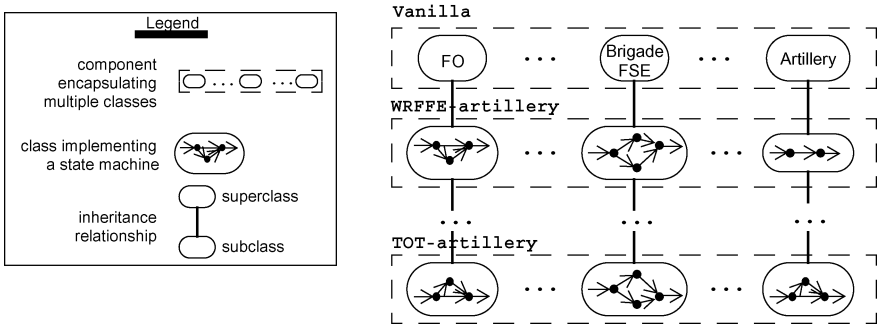


Fig. 4. OPFAC inheritance refinement hierarchy.

variabilities in FSATS throughout its history (and projected future) lie in the addition, enhancement, and removal of mission types. By encapsulating mission types as components, evolution of FSATS is greatly simplified.

We noted earlier, that every mission type has a "cross-cutting effect", because the addition or removal of a mission type impacts multiple OPFAC programs. A mission type is an example of a common kind of refinement called a *collaboration*—a set of objects that work collectively to achieve a certain goal [Reenskaug et al. 1992; Smaragdakis and Batory 1998; Van Hilst and Notkin 1996]. Collaborations have the desirable property that they can be defined largely in isolation from other collaborations, thereby simplifying application design. In the case of FSATS, a mission is a collaboration of objects (OPFACs) that work cooperatively to prosecute a particular mission. The actions taken by each OPFAC are defined by a protocol (state machine) that it follows to do its part in processing a mission thread. Different OPFACs follow different protocols for different mission types.

An extensible, component-based design for FSATS follows directly from these observations. One component (`Vanilla`) defines an initial OPFAC class hierarchy and routines for sending and receiving messages, routing messages to appropriate missions, reading simulation scripts, and so forth. Figure 4 depicts the `Vanilla` component encapsulating multiple classes, one per OPFAC type. The OPFACs that are defined in `Vanilla` do not know how to react to external stimuli. Such reactions are encapsulated in mission components.

Each mission component encapsulates protocols (expressed as state machines) that are added to each OPFAC that could participate in a thread of this mission type. Composing a mission component with `Vanilla` extends each OPFAC with knowledge of how to react to particular external stimuli and how to coordinate its response with other OPFACs. For example, when the `WRFFE-artillery` component is added, a forward observer now has a protocol that tells it how to react when it sees an enemy tank—it creates a `WRFFE-artillery` message which it relays to its FIST. The FIST commander, in turn, follows his `WRFFE-artillery` protocol to forward this message to his battalion FSE, and so on. Figure 4 depicts the `WRFFE-artillery` component encapsulating multiple classes, again one per OPFAC type. Each enclosed class encapsulates a protocol which is added to its appropriate OPFAC class. Component

composition is accomplished via inheritance, and is shown by dark vertical lines between class ovals in Figure 4. The same holds for other mission components (e.g., `TOT-artillery`). *Note that the classes that are instantiated are the bottom-most classes of these linear inheritance chains, because they embody all the protocols/features that have been grafted onto each OPFAC.* Readers will recognize this is an example of the GenVoca paradigm of Section 2.2, where components are mixin-layers.

The GenVoca-FSATS design has distinct advantages:

- it is mission-type *extensible* (i.e., it is comparatively easy to add new mission types to an existing GenVoca-FSATS simulator),[6]
- each mission type is defined largely independently of others, thereby reducing the difficulties of specification, coding, and debugging, and
- understandability is improved: OPFAC behavior is routinely understood and analyzed as mission threads. Mission-type components directly capture this simplicity, avoiding the complications of knowledge acquisition and engineering of rule sets.

Implementation. There are over twenty different mixin-layer components in GenVoca-FSATS, all of which are composed now to form a "fully-loaded" simulator. There are individual components for each mission type, just like Figure 4. However, there is no monolithic `Vanilla` component. We discovered that `Vanilla` could be decomposed into ten largely independent layers (totalling 97 classes) that deal with different aspects of the FSATS infrastructure. For example, there are distinct components for:

- OPFACs reading from simulation scripts,
- OPFAC communication with local and remote processes,
- OPFAC proxies (objects that are used to evaluate whether OPFAC commanders are supported by desired weapons platforms),
- different weapon OPFACs (e.g., distinct components for mortar, artillery, etc.), and
- GUI displays for graphical depiction of ongoing simulations.

Packaging these capabilities as distinct components, simplifies both specifications (because no extraneous details need to be included), and debugging (as components can largely be debugged in isolation). An important feature of our design is that all OPFACs are coded as threads executing within a single Java process. An "RMI adaptor" component transforms this design into a distributed program where each OPFAC thread executes in its own process at a different site [Batory et al. 1999]. The advantage here is that it is substantially easier to debug layers and mission threads within a single process than to debug remote

[6]Although a product-line of different FSATS simulators is possible; presently the emphasis of FSATS is on mission type extensibility. It is worth noting, however, that exponentially-large product-lines of FSATS simulators could be synthesized—i.e., if there are m mission components, there can be up to 2^m distinct compositions/simulators.

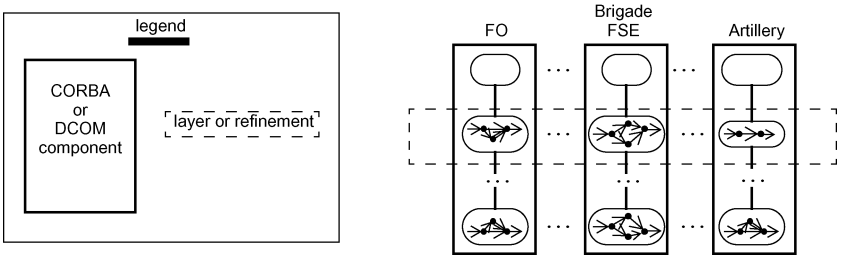


Fig. 5. CORBA and DCOM vs. layers (refinements).

executions. Furthermore, the adaptor is included in an FSATS design only when distributed simulations are needed.

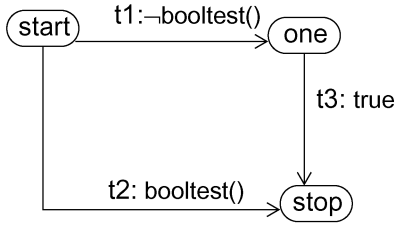
Perspective. It is worth comparing our notion of components with those that are common in today's software industry. Event-based distributed architectures, where DCOM and CORBA components communicate via message exchanges, is likely to be a dominant architectural paradigm of the future [Taylor 1999]. The original design of FSATS is a classic example: OPFAC programs are distributed DCOM/CORBA "components" that exchange messages. Yet the "components" common to distributed architectures are *orthogonal* to the components in the GenVoca-FSATS design. (This is depicted below in Figure 5 where each vertical inheritance chain corresponds to an OPFAC that is a CORBA or DCOM class, whereas an FSATS mission type is depicted by a horizontal slice through all OPFACs). That is, our components (layers) encapsulate fragments of *many* OPFACs, instead of encapsulating an *individual* OPFAC. (This is typical of approaches based on collaboration-based or "aspect-based" designs).

Event-based architectures are clearly extensible by their ability to add and remove component instances (e.g., adding and removing OPFACs from a simulation). This is (*OPFAC*) *object population* extensibility, which FSATS definitely requires. But FSATS also needs *feature* extensibility—OPFAC programs must be mission-type extensible. While these distinctions seem obvious in hind-sight, they were not so, prior to our work. FSATS clearly differentiates them.

3.2 A Domain-Specific Language for State Machines

We discovered that OPFAC rule sets were largely representations of state machines. We found that expressing OPFAC actions as state machines was a *substantial* improvement over rules; they are much easier to explain and understand, and require very little background to comprehend. A major goal of the redesign was to minimize the "conceptual distance" between architectural abstractions and their implementation. The problem we faced is that encodings of state machines are obscure, and given the situation that our specifications often *refined* previously created machines, expressing state machines in pure Java code was unattractive. To eliminate these problems, we used JTS to extend Java with a domain-specific language for declaring and refining state machines, so that our informal state machines (nodes, edges, etc.) had a direct expression as

(a)



(b)

```
state_machine exampleJavaSM {

    event_delivery receive_message(M m);          (1)
    on_error { error(-1,m); }                     (2)
    otherwise_default { ignore_message(m); }      (3)

    states start, one, stop;                      (4)

    edge t1 : start -> one                        (5)
    conditions !booltest()
    do { /* t1 action */ }

    edge t2 : start -> stop
    conditions booltest()
    do { /* t2 action */ }

    edge t3 : one -> stop
    conditions true
    do { /* t3 action */ }

    // methods and data members from here on... (6)

    boolean booltest() { ... }
    exampleJavaSM() { current_state = start; }
    ...
}
```

Fig. 6. JavaSM state machine specification.

a formal, compilable document. This extended version of Java is called *JavaSM*, and took us a bit more than a week to code into JTS.

Initial Declarations. A central idea of JavaSM is that a state machine specification translates into the definition of a single class. There is a generated variable (`current_state`) whose value indicates the current state of the protocol (i.e., state-machine-class instance). When a message is received by an OPFAC, a designated method is invoked with this message as an argument; depending on the state of the protocol, different transitions occur. Figure 6a shows a simple state machine with three states and three transitions. When a message arrives in the `start` state, if method `booltest()` is true, the state advances to `stop`; otherwise the next state is `one`.

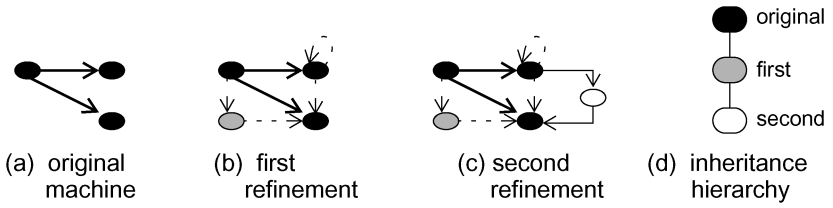


Fig. 7. Refining state machines.

Our model of FSATS required boolean conditions that triggered a transition to be arbitrary Java expressions with no side-effects, and the actions performed by a transition, to be arbitrary Java statements. Figure 6b shows a JavaSM specification of Figure 6a. (1) defines the name and formal parameters of the void method that delivers a message to the state machine. In the case that actions have corrupted the current state, (2) defines the code that is to be executed upon error discovery. When a message is received and no transition is activated, (3) defines the code that is to be executed (in this case, ignore the message). The three states in Figure 6a are declared in (4). Edges are declared in (5): each edge has a name, start state, end state, transition condition, and transition action. Java data member declarations and methods are introduced after edge declarations (6). When the specification of Figure 6b is translated, the class `exampleJavaSM` is generated. Additional capabilities of JavaSM are discussed in Batory et al. [1998].

Refinement Declarations. State machines can be progressively refined in a layered manner. A *refinement* is the addition of states, edges, data members, and methods to an existing machine. A common situation in FSATS is illustrated in Figure 7. Protocols for missions of the same general type (e.g., WRFFE) share the same protocol fragment for initialization (Figure 7a). A particular mission type (e.g, WRFFE-artillery) grafts on states and edges that are specific to it (Figure 7b). Additional missions contribute their own states and edges (Figure 7c), thus allowing complex state machines to be built in a stepwise manner.

The original state machine and each refinement are expressed as separate JavaSM specifications that are encapsulated in distinct layers. When these layers are composed, their JavaSM specifications are translated into a Java class hierarchy. Figure 7d shows this hierarchy: the root class was generated from the JavaSM specification of Figure 7a; its immediate subclass was generated from the JavaSM refinement specification of Figure 7b; the terminal subclass was generated from the JavaSM refinement specification of Figure 7c. Figure 8 sketches a JavaSM specification of this refinement chain.

Inheritance (i.e., class refinement) plays a central role in this implementation. All the states and edges in Figure 7a are inherited by the machine refinements of Figure 7b, and these states, edges, and so forth, are inherited by the machine refinements of Figure 7c. The machine that is executed, is created by instantiating the bottom-most class of the refinement chain of Figure 7d. Readers will again recognize this an example of the GenVoca paradigm of Section 2.2.

```
state_machine original {
   states one, two, three;

   edge a : one -> two ...
   edge b : one -> three ...
}

state_machine first refines original {
   states four;

   edge c : one -> four ...
   edge d : four -> three...
   edge e : two -> three ...
   edge f : two -> two ...
}

state_machine second refines first {
   states five;

   edge g : two-> five ...
   edge h : five -> three ...
}
```

Fig. 8. A JavaSM refinement hierarchy.

Perspective. Domain-specific languages for state machines are common (e.g., Berry and Gonthier 1992; Ellsberger et al. 1997; Harel 1987; Harel and Gery 1996; Neighbors 1997). Our way of expressing state machines—as states with enter and exit methods, edges with conditions and actions—is an elementary subset of Harel's Statecharts [Harel 1987; Harel and Gery 1996] and SDL extended finite state machines [Ellsberger et al. 1997]. The notion of refinement in Statecharts is the ability to explode individual nodes into complex state machines. This is very different than the notion of refinement explored in this paper. Our work is closer to the refinement of extended finite state machines in SDL where a process class (which encodes a state machine) can be refined via subclassing (i.e., new states and edges are added to extend the parent machine's capabilities). While the idea of state machine refinements is not new, it is new in the context of a DSL-addition to a general-purpose programming language (Java), and it is fundamental in the context of component-based development of FSATS simulators.

4. PRELIMINARY RESULTS

Our preliminary findings are encouraging—the objectives of the redesign were met by the GenVoca-FSATs design:

- it is now possible to specify, add, verify, and test a mission type independent of other mission types (because a version of FSATS can be created with a single mission),

- it is now possible to remove and replace mission types to accommodate varying user requirements, and
- JavaSM allows a direct implementation of a specification, thereby reducing the "conceptual distance" between specification and implementation.

As is common in re-engineering projects, detailed statistics on the effort involved in the original implementation are not available. However, we can make some rough comparisons. From our experience with the original FSATS simulator, we estimate the time to add a mission to be about 1 month. A similar addition to GenVoca-FSATS, including one iteration to identify and correct an initial misunderstanding of the protocols for that mission, was accomplished in about 3 days.

To evaluate the redesign in a less anecdotal fashion, we collected statistics on program complexity. We used simple measures of class complexity: the number of methods (**nmeth**), the number of lines of code (**nloc**), and the number of tokens/symbols (**nsymb**) per class. (We originally used other metrics [Chidamber and Kemerer 1991], but found they provided no further insights.) Because of our use of JTS, we have access to both component-specification code (i.e., layered JavaSM code written by FSATS engineers), and generated non-layered pure-Java code (which approximates code that would have been written by hand). By using metrics to compare pure-Java code vs. JavaSM code, and layered vs. non-layered code, we can quantitatively evaluate the impact of layering and JavaSM on reducing program complexity, a key goal of our redesign.

Complexity of Non-Layered Java Code. Consider a non-layered design of FSATS. Suppose all of our class refinement chains were "squashed" into single classes; these would be the classes that would be written by hand if a non-layered design were used. Consider the FSATS class hierarchy that is rooted by class `MissionImpl`; this class encapsulates methods and an encoding of a state machine that is shared by all OPFACS. (In our prototype, we implemented different variants of WRFFE missions.) Class `FoMission`, a subclass of `MissionImpl`, encapsulates the additional methods and the Java-equivalent of state machine edges/states that define the actions that are specific to a Forward Observer. Other subclasses of `MissionImpl` encapsulate additions that are specific to other OPFACs. The "Pure Java" columns of Table I present complexity statistics of the `FoMission` and `MissionImpl` classes. *Note that our statistics for subclasses*, *by definition*, *must be no less than those of their superclasses* (*because the complexity of superclasses is inherited*).

One observation is immediately apparent: the number of methods (117) in `MissionImpl` is huge. Different encoding techniques for state machines might reduce the number, but the complexity would be shifted elsewhere (e.g., methods would become more complicated). Because our prototype only deals with WRFFE missions, we must expect that the number of methods in `MissionImpl` will increase, as more mission types are added. Consider the following: there are 30 methods in class `MissionImpl` alone that are used in WRFFE missions. When we add a WRFFE mission that is specialized for a particular weapon system (e.g., mortar), another 10 methods are added. Since WRFFE is representative of mission complexity, as more mission types are added with their

Table I. Statistics for Non-Layered Implementation of Class FoMission

	Pure Java			JavaSM		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
`MissionImpl`	117	461	3452	54	133	1445
`FoMission`	119	490	3737	56	143	1615

weapon specializations, it is not inconceivable that `MissionImpl` will have several hundred methods. Clearly, such a class would be both incomprehensible and unmaintainable.[7]

Now consider the effects of using JavaSM. The "JavaSM" columns of Table I show corresponding statistics, where state exit and enter declarations and edge declarations are treated as (equivalent in complexity to) method declarations. We call such declarations *method-equivalents*. Comparing the corresponding columns in Table I, we see that coding in JavaSM reduces software complexity by a factor of 2. That is, the number of method-equivalents is reduced by a factor of 2 (from 119 to 56), the number of lines of code is reduced by a factor of 3 (from 490 to 143), and the number of symbols is reduced by a factor of 2 (from 3737 to 1615). However, the problem that we noted in the pure-Java implementation remains. Namely, the generic WRFFE mission contributes over 10 method-equivalents to `MissionImpl` alone; when WRFFE is specialized for a particular weapon system (e.g., mortar), another 3 method-equivalents are added. While this is substantially better than its non-layered pure-Java equivalent, it is not inconceivable that `MissionImpl` will have over a hundred method-equivalents in the future. *While the JavaSM DSL indeed simplifies specifications, it only delays the onset of design fatigue. Non-layered designs of FSATS may be difficult to scale and ultimately hard to maintain, even if the JavaSM DSL is used.*

Complexity of Layered Java Code. Now consider a layered design implemented in pure Java. The "Inherited Complexity" columns of Table II show the inheritance-cumulative statistics for each class of the `MissionImpl` and `FoMission` refinement chains. The rows where `MissionImpl` and `FoMission` data are listed in **bold** represent classes that are the terminals of their respective refinement chains. These rows correspond to the rows in Table I. The "Isolated Complexity" columns of Table II show complexity statistics for individual classes of Table II (i.e., we are measuring class complexity, and not including the complexity of superclasses). Note that most classes are rather simple. The `MissionAnyL.MissionImpl` class, for example, is the most complex, with 43 methods. (This class encapsulates "infrastructure" methods used by all missions.) Table II indicates that layering disentangles the logic of different features of the `FoMission` and `MissionImpl` classes into units that are small enough to be comprehensible and manageable by programmers. For example,

[7]It would be expected that programmers would introduce some other modularity, thereby decomposing a class with hundreds of methods into multiple classes with smaller numbers of methods. While this would indeed work, it would complicate the "white-board"-to-implementation mapping (which is what we want to avoid) and there would be no guarantee that the resulting design would be mission-type extensible.

Table II. Statistics for a Layered Java Implementation of Class FoMission

	Inherited Complexity			Isolated Complexity		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
MissionL.MissionImpl	9	25	209	9	25	209
ProxyL.MissionImpl	11	30	261	2	5	52
MissionAnyL.MissionImpl	51	179	1431	43	149	1170
MissionWrffeL.MissionImpl	83	314	2342	35	135	911
MissionWrffeMortarL.MissionImpl	93	358	2677	13	44	335
MissionWrffeArtyL.MissionImpl	109	425	3187	19	67	510
MissionWrffeMlrsL.MissionImpl	**117**	**461**	**3452**	**11**	**36**	**265**
BasicL.FoMission	117	461	3468	0	0	16
MissionWrffeMortarL.FoMission	117	468	3547	4	7	79
MissionWrffeArtyL.FoMission	119	484	3687	7	16	140
MissionWrffeMlrs.FoMission	**119**	**490**	**3737**	**3**	**6**	**50**

Table III. Statistics on a Layered JavaSM Implementation of Class FoMission

	Inherited Complexity			Isolated Complexity		
Class Name	**nmeth**	**nloc**	**nsymb**	**nmeth**	**nloc**	**nsymb**
MissionL.MissionImpl	8	20	169	8	20	169
ProxyL.MissionImpl	10	25	221	2	5	52
MissionAnyL.MissionImpl	34	90	877	24	65	656
MissionWrffeL.MissionImpl	45	115	1132	11	25	255
MissionWrffeMortarL.MissionImpl	48	121	1231	3	6	99
MissionWrffeArtyL.MissionImpl	52	129	1383	4	8	152
MissionWrffeMlrsL.MissionImpl	**54**	**133**	**1445**	**2**	**4**	**62**
BasicL.FoMission	54	133	1461	0	0	16
MissionWrffeMortarL.FoMission	54	136	1518	2	3	57
MissionWrffeArtyL.FoMission	55	140	1586	3	4	68
MissionWrffeMlrs.FoMission	**56**	**143**	**1615**	**2**	**3**	**29**

instead of having to understand a class with 117 methods, the largest layered subclass has 43 methods; instead of 461 lines of code there are 149 lines, and so on.

To gauge the impact of a layered design in JavaSM, consider the "Inherited Complexity" columns of Table III that show statistics for `MissionImpl` and `FoMission` refinement chains written in JavaSM. The "Isolated Complexity" columns of Table III show corresponding statistics for individual classes. They show that layered JavaSM specifications are indeed compact: instead of a class with 43 methods there are 24 method-equivalents, instead of 149 lines of code there are 65 lines, and so on. Thus, a combination of domain-specific languages and layered designs greatly reduces program complexity.

Our use of the "Isolated Complexity" metric as the indicator of class complexity requires some discussion. It is indeed the case that the "true" complexity of a class is somehow related to the total complexity of its superclasses plus the additional complexity of the class itself. So it could be argued that the "Inherited Complexity" metric might be a better measure of the actual difficulty of understanding a given layer. This is not the case for FSATS. Typically FSATS layers simply invoke methods of their superclass, much in the same

way that COM and CORBA components invoke methods of server interfaces. Implementation details are hidden behind such interfaces, thereby making it easy for programmers to invoke server methods without having to know how servers are implemented. The same holds for layers in FSATS. The only difference here, is that a *few* methods of each FSATS class override (i.e., extend) previously defined methods, thereby requiring programmers to know more of the "guts" of superclass implementation. But for FSATs (and other generators that we have built), this additional implementation knowledge is minimal. Further, there may be layers in superclass implementations that provide infrastructure that programmers of mission-layers do not need to be aware of at all; they are simply methods that are private to that layer. For these reasons, we believe that "Isolated Complexity" is closer to the true complexity of a class than "Inherited Complexity."

The reduction in program complexity is a key goal of our project; these tables support the observations of FSATS engineers: the mapping between a "white-board" design of FSATS protocols and an implementation, is both direct and invertible with layered JavaSM specifications. That is, writing components in JavaSM matches the informal designs that domain experts use; it requires fewer mental transformations from design to implementation, which simplifies maintenance and extensibility, and makes for a much less error-prone product. In contrast, mapping from the original FSATS implementation back to the design was not possible due to the lack of an association of any particular rule or set of rules with a specific mission.

5. CONCLUSIONS

Extensibility is the property that simple changes to the design of a software artifact require a proportionally simple effort to modify its source code. Extensibility is a result of premeditated engineering, whereby anticipated variabilities in a domain are made simple by design. Two complementary technologies are emerging that make extensibility possible: *product-line architectures* (*PLAs*) and *domain-specific languages* (*DSLs*). Product-lines rely on components to encapsulate the implementation of basic features or "aspects" that are common to applications in a domain; applications are extensible through the addition and removal of components. Domain-specific languages enable applications to be programmed in domain abstractions, thereby allowing compact, clear, and machine-processable specifications to replace detailed and abstruse code. Extensibility is achieved through the evolution of specifications.

FSATS is a simulator for Army fire support and is representative of a complex domain of distributed command-and-control applications. The original implementation of FSATS had reached a state of design fatigue, where anticipated changes/enhancements to its capabilities would be expensive to realize. We undertook the task of redesigning FSATS so that its inherent and projected variabilities—that of adding new mission types—would be easy to introduce. Another important goal was to minimize the "conceptual distance" from "white-board" designs of domain experts to actual program specifications; because of the complexity of fire-support, the specifications had to closely match

these designs to make the next-generation FSATS source understandable and maintainable.

We achieved the goals of extensibility and understandability through an integration of PLA and DSL technologies. We used a GenVoca PLA to express the building blocks of fire support simulators as layers or refinements, whose addition or removal simultaneously impacts the source code of multiple, distributed programs. But a layered design was insufficient, because our components could not be easily written in pure Java. The reason is that the code expressing state machine abstractions was so low-level that it would be difficult to read and maintain. We addressed this problem by extending the Java language with a domain-specific language, to express state machines and their refinements, and wrote our components in this extended language. Preliminary findings confirm that our component specifications are substantially simplified; "white-board" designs of domain experts have direct and invertible expressions in our specifications. Thus, we believe that the combination of PLAs and DSLs is essential in creating extensible fire support simulators.

While fire support is admittedly a domain with specific and unusual requirements, there is nothing domain-specific about the need for PLAs, DSLs, and their benefits. In this regard, FSATS is not unusual; it is a classical example of the many domains where both technologies naturally complement each other to produce a result that is better than either technology could deliver in isolation. Research on PLA and DSL technologies should focus on infrastructures (such as IP [Simonyi 1995] and JTS [Batory et al. 1998]) that support their integration; research on PLA and DSL methodologies must be more cognizant that synergy is not only possible, but desirable.

ACKNOWLEDGMENTS

We thank Dewayne Perry (UTexas) for his insightful comments on an earlier draft and bringing SDL to our attention, and Frank Weil (Motorola) for clarifying discussions on SDL state machines. We also thank the referees for their helpful suggestions that improved the final draft of this paper.

REFERENCES

FSATS 1999. "System Segment Specification (SSS) for the Fire Support Automated Test System", Applied Research Laboratories, The University of Texas. See URL `http://www.arlut.utexas.edu/~fsatswww/fsats.shtml`.

BATORY, D. AND O'MALLEY, S. 1992. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. Softw. Eng. Methodol.* (Oct.).

BATORY, D., COGLIANESE, L., GOODWILL, M., AND SHAFER, S. 1995. Creating Reference Architectures: An Example from Avionics. *Symposium on Software Reusability*, Seattle, WA. (Apr.).

BATORY, D., LOFASO, B., AND SMARAGDAKIS, Y. 1998. JTS: Tools for Implementing Domain-Specific Languages. *5th International Conference on Software Reuse*, Victoria, Canada (June). `http://www.cs.utexas.edu/users/schwartz/JTS30Beta2.htm`.

BATORY, D. AND GERACI, B. J. 1997. Composition Validation and Subjectivity in GenVoca Generators. *IEEE Trans. Softw. Eng.* (Feb.), 67–82.

BATORY, D. 1998. Product-Line Architectures. *Smalltalk and Java Conference*, Erfurt, Germany (Oct.).

BATORY, D., SMARAGDAKIS, Y., AND COGLIANESE, L. 1999. Architectural Styles as Adaptors. *Software Architecture*, Patrick Donohoe, ed., Kluwer Academic Publishers.

BATORY, D., CHEN, G., ROBERTSON, E., AND WANG, T. 2000. Design Wizards and Visual Programming Environments for GenVoca Generators. *IEEE Trans. Softw. Eng.* (May), 441–452.

BAXTER, I. 1992. Design Maintenance Systems. *CACM* (Apr.).

BERRY, G. AND GONTHIER, G 1992. The Esterel Synchronous Programming language: Design, Semantics, and Implementation. *Science of Computer Programming*. 87–152.

BOSCH, J. 1999. Product-Line Architectures in Industry: A Case Study. *ICSE*, Los Angeles, CA.

CARDONE, R. AND LIN, C. 2001. Comparing Frameworks and Layered Refinement. ICSE Toronto.

CHIDAMBER S. R. AND KEMERER, C. F. 1991. Towards a Metrics Suite for Object Oriented Design. *OOPSLA*.

CZARNECKI, K. AND EISENECKER, U. W. 1999. Components and Generative Programming. *ACM SIGSOFT*.

VAN DEURSEN, A. AND KLINT, P. 1997. Little Languages: Little Maintenance? *SIGPLAN Workshop on Domain-Specific Languages*.

EICK, S. G., GRAVES, T. L., KARR, A. F., MARRON, J. S., AND MOCKUS, A. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. Softw. Eng.*, *27*:1 (January), 1–12.

ELLSBERGER, J., HOGREFE, D., AND SARMA, A. 1997. *Formal Object-Oriented Language for Communicating Systems*. Prentice-Hall, Englewood Cliffs, NJ.

FINDLER, R. B. AND FLATT, M. "Modular Object-Oriented Programming with Units and Mixins", *ICFP 98*.

GAMMA E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

GOGUEN, J. A. 1986. Reusing and Interconnecting Software Components. *IEEE Computer* (Feb.).

GOMAA, H., KERSCHBERG, L., AND SUGAMARAN, V. 1992. A Knowledge-Based Approach to Generating Target System Specifications from a Domain Model. *IFIP Congress 1*, 252–258.

GRISS, M. 2000. Implementing Product-Line Features by Composing Component Aspects. *First International Software Product-Line Conference*, Denver, CO. (Aug.).

HAREL, D. 1987. Statecharts: A Visual Formalism for Complex Systems, *Science of Computer Programming*, 231–274.

HAREL, D. AND GERY, E. 1996. Executable Object Modeling with Statecharts. *ICSE*.

KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. 1990. Feature-Oriented Domain Analysis Feasibility Study, SEI. Technical Report CMU/SEI-90-TR-21 (Nov.).

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-Oriented Programming, *ECOOP*. 220–242.

LOPEZ-HERREJON, R. E. AND BATORY, D. 2001. A Standard Problem for Evaluating Product-Line Methodologies. *Third International Conference on Generative and Component-Based Software Engineering* (*GCSE 2001*), (Sept. 9–13), Messe Erfurt, Erfurt, Germany.

MAGNAVOX, 1999. System Segment Specification (SSS) for the Advanced Field Artillery Tactical Data System (AFATDS).

NEIGHBORS, J. 1989. Draco: A Method for Engineering Reusable Software Components, in T. J. Biggerstaff and A. Perlis, eds., *Software Reusability*, Addison-Wesley/ACM Press.

NEIGHBORS, J. 1997. "DataXfer Protocol," BayFront Technologies. 1997, URL `http://bayfront-technologies.com`.

REENSKAUG, T., ET AL. 1992. "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", *Journal of Object-Oriented Programming*, 5(6) (Oct.), 27–41.

RICH, C. AND WATERS, R. C. 1990. *The Programmer's Apprentice*, ACM Press.

Software Engineering Institute 2001. The Product Line Practice (PLP) Initiative, URL `http://www.sei.cmu.edu/plp/plp_init.html`.

SIMONYI, C. 1995. The Death of Computer Languages, the Birth of Intentional Programming, *NATO Science Committee Conference*.

SMARAGDAKIS, Y. AND BATORY, D. 1998. Implementing Layered Designs with Mixin Layers, *ECOOP*.

SMARAGDAKIS, Y. AND BATORY, D. 2002. Mixin Layers: An Object-Orinted Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng.* Method.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, S. M. JR. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns, *ICSE*.

TAYLOR, R. 1999. Panel on Software Reuse. *Motorola Software Engineering Symposium*, Ft. Lauderdale.

TOKUDA, L. AND BATORY, D. 1999. Evolving Object-Oriented Architectures with Refactorings. *Conference on Automated Software Engineering*, Orlando, FL.

VAN HILST, M. AND NOTKIN, D. 1996. Using Role Components to Implement Collaboration-Based Designs, *OOPSLA*, 359–369.

WEISS, D. M. AND LAI, C. T. R 1999. *Software Product-Line Engineering*. Addison-Wesley.

Received February 2001; revised September 2001; accepted October 2001