DSL Design in an Industrial Setting

Pacific Software Research Center
March 28, 2000

CONTRACT NO. F19628-96-C-0161
CDRL SEQUENCE NO. [CDRL 0002.25]

Prepared for:
USAF
Electronic Systems Center/AVK

Prepared for:
Pacific Software Research Center
Oregon Graduate Institute of Science and Technology
PO Box 91000
Portland, OR 97291

Formal Language Design in the Context of Domain Engineering

Tanya Widen

B.S.,Western Washington University, 1995

A thesis presented to the faculty of the

Oregon Graduate Institute of Science and Technology

in partial fulfillment of the

requirements for the degree

Master of Science

in

Computer Science and Engineering

June 1998

The thesis "Formal Language Design in the Context of Domain Engineering" by Tanya Widen has been examined and approved by the following Examination Commitee:

James Hook, Thesis Advisor
Associate Professor

Lois M. L. Delcambre
Associate Professor

John Launchbury
Assistant Professor

Acknowledgements

I would like to thank the following people for their support and encouragement: my family, Julie Wilson, Karen Ward, and Dresden Skees. I am also very grateful to the many people who provided excellent feedback on my writing. Special thanks go out to Jim Hook for his continuous guidance, and to Lisa Walton and Walid Taha for their comments.

Table of Contents

Abstract

Even though many domain specific languages have been implemented, there is a lack of well defined methods available for both analyzing domains with the intent of creating a domain specific lan guage and developing the domains specific languages. The research presented in this thesis aims at this problem by capturing, analyzing and merging two methods that define domain engineering processes. Ini tial analysis of the methods indicated that they have complementary strengths. Therefore, merging the two methods promises to produce a more robust method.

Will the integration of two domain engineering methods - one which has a mature domain analysis phase, and one which has a well studied method for developing domain specific languages given an appro priate domain analysis - produce such a complete formal domain engineering method? It is the thesis of this research that integrating the key features of each method will indeed produce a more robust method. Capturing the two methods involves studying documents describing the processes as well as obtaining additional information from experts on the techniques of the methods.

Capturing the definition of the methods and validating these definitions with experts is essential to ensure that the process is cor rectly understood. For one of the methods much additional work has to be done to synthesize information from the method experts, as the definition of the method is not well docu- mented or agreed upon.

Analyzing the methods involves analyzing the general characteristics of each method as well as analyzing each process step and its related work products. The analysis looks at the strengths and weak nesses of each method based on principles of software engineering and observations made about the appli cation of the methods. This analysis is used to compare the two methods and to propose an integrated method based on this comparison.

The results of this research are the definitions of the individual methods along with their analyses as well as a proposal for an integrated method that appears to offset some of the weaknesses of the original methods. Further work must still be done to validate the proposed method.

1. Introduction

1.1. Overview

The demands of the software industry on software providers to produce and maintain more complex products faster, better and cheaper has necessitated the need for methods and techniques to support the software production process. The field of software engineering research focuses on such solutions. Reuse is one area being studied within this field. Reuse of assets such as code or documentation reduces the amount of original work needed on each product, thereby reducing the production time and cost of development and increasing the quality of each product through the reuse of quality ensured assets.

Domain engineering research is a subset of reuse research. Domain engineering aims at achieving high levels of reuse in a specific domain as opposed to achieving general reuse across many assorted applications. Domain-specific languages are one approach for achieving high levels of reuse in a domain. A domain-specific language provides high level domain-specific abstractions, for specifying domain instances. These abstractions can be much more specialized than general purpose programming constructs. Domain-specific languages encapsulate the lower level implementation of products from the developer, much like a high level language encapsulates the assembly code that will be produced to run on a specific machine.

However, even though many domain-specific languages have been developed, there are no well defined methods available for both analyzing domains for an appropriate domain-specific language and for developing the domain-specific languages. The research presented in this thesis begins to solve this problem by capturing, analyzing and merging two domain engineering processes, for which initial analysis indicated complementary strengths [65]. Therefore, merging the two methods promises to produce a more robust method.

Will the integration of two domain engineering methods — one which has a mature domain analysis phase, and one which has a well studied method for developing domain-specific languages — produce such a complete domain engineering method? It is the thesis of this research that integrating the key features of each method will indeed produce a more robust method.

The rest of this chapter provides an introduction to the process of domain engineering followed by a description of how the methods were captured, analyzed and merged to create a combined method which appears to offset some of the weaknesses of the original methods.

1.2. Domain Engineering

Domain engineering is a process for defining, creating, and evolving reusable assets for a family of applications in a domain [2]. Figure 1 illustrates that domain engineering is a meta-level process for application engineering. Domain engineering creates products that can be used to create multiple different products during application engineering.

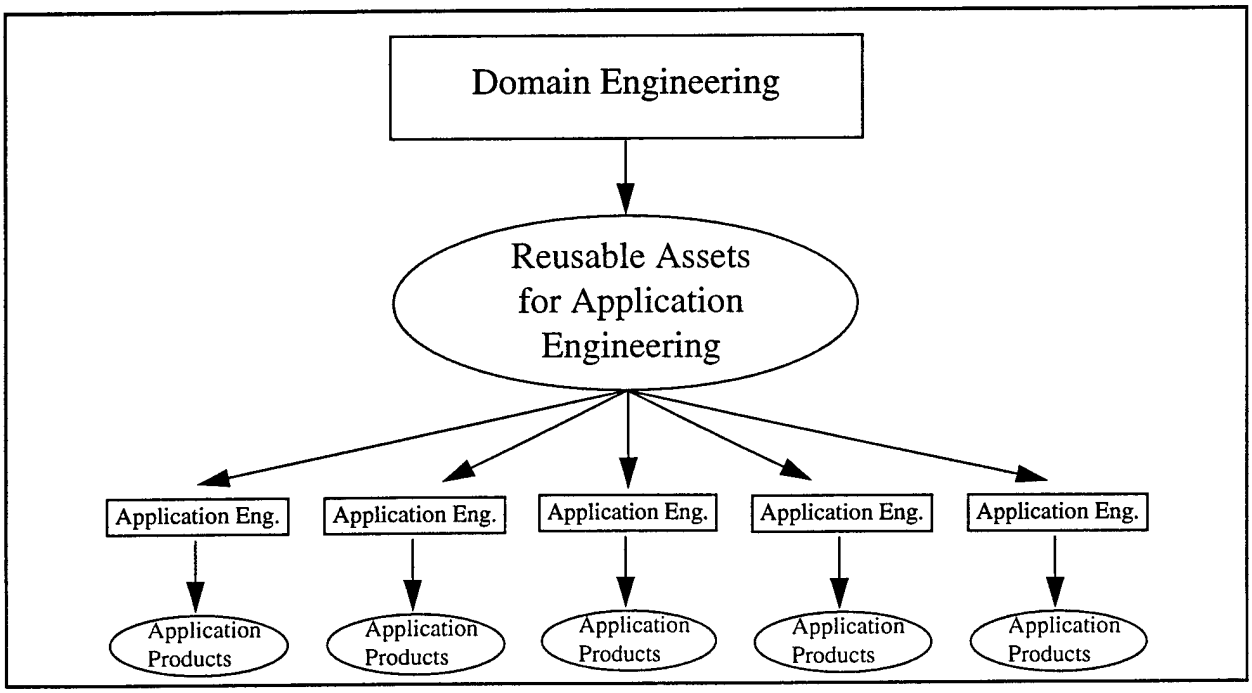

Figure 1: Domain Engineering

Reusable assets include processes, products, and tools that are intended to assist application engineers in developing and maintaining product instances, or applications, in the domain. A domain is a problem area where there may be many products — past, present, and future — that provide solutions to specific problems. A domain has many core concepts that define the family and the problem instances have variabilities that distinguish them from one another. Therefore it is worthwhile to study the problems from the set by first studying the common properties of the set and determining the special properties of the individual family members [33]. Example domains are word processing, grammars for languages, and elevator control systems. Each instance of a word processor, grammar and elevator control system is similar in some respects to other instances in the same domain.

Similar to the application engineering phases of requirements analysis, system specification and design and system implementation, domain engineering can be broken down into phases; domain analysis, domain design and domain implementation (Table 1). However, while application engineering focuses on

producing only one instance, domain engineering focuses on producing assets that can be used to produce multiple instances.

Table 1: Application Engineering vs. Domain Engineering

Application Engineering	Domain Engineering
Requirements analysis	Domain analysis
System specification and design	Domain design
System implementation	Domain implementation

Arango and Prieto-Diaz describe the high level phases of domain engineering as follows [2]. In the domain analysis phase of domain engineering information about the problem domain is gathered and analyzed to support the description and solution of those problems. A domain model is defined which identifies the commonalities and variabilities in the domain. In the domain design phase the aspects of the problem domain that will be supported by solutions are identified and specified. Finally in the domain implementation phase the solution products are created and tested for use.

Some domain engineering products are reusable components, architectures and domain-specific languages (DSL). These products are described in more detail in Section 2. Of interest here are executable DSLs. Figure 2 shows that an application generator takes a specification of a family instance in a DSL and, much like a traditional language compiler, produces a working solution in a target language of the domain.

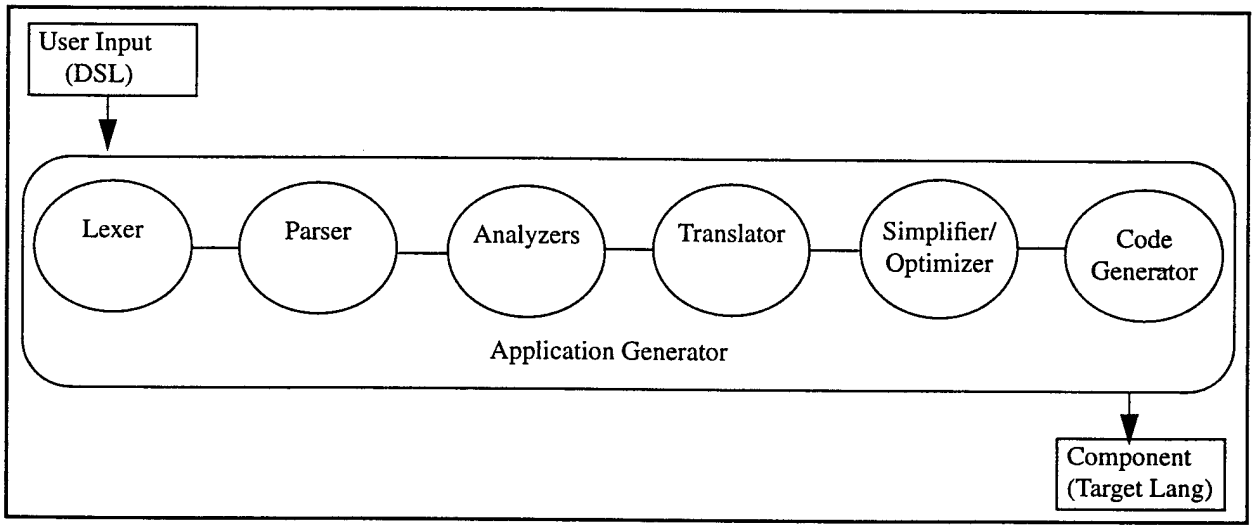

Figure 2: Application Generator

In the section on "Terminology" at the end of the thesis, terms are defined that are used throughout this thesis.

1.3. Integration Project Description

Two specific domain engineering methods, Lucent Technology Inc.'s (Lucent) Family-oriented Abstraction, Specification, and Translation (FAST) method and the Pacific Software Research Center's (PacSoft) Software Design Automation (SDA) method, are being studied for this integration project. The reasons that these two particular methods were chosen follow. Firstly, both methods have the ultimate goal of producing a domain-specific language for specifying and generating components in the domain. This differs from other domain engineering approaches which typically target components or architectures. Secondly, preliminary analysis shows that the methods are complementary; FAST has a well defined yet flexible domain analysis process while SDA is being defined to facilitate language design and implementation.

FAST's domain analysis process is the Commonality Analysis, which is a mature, well defined domain analysis method. However, the rest of the method, which includes language design and implementation, is not as well defined and hence is not as mature as the Commonality Analysis. SDA, on the other hand, is a method focusing on the development and implementation of domain-specific languages. However, it lacks steps for gathering and analyzing information. The intention is to combine the front end of FAST with the back end of SDA to come up with a complete domain engineering process that incorporates the strengths of both methods.

To incorporate the important features from each method, they must both be captured and analyzed in-depth to determine their strengths and weaknesses. The FAST method is captured and analyzed from information available on the FAST process, as well as from information available on the Software Productivity Consortium's Synthesis process, which is the basis of FAST. The information examined includes papers, course notes, and a process definition for FAST [4,67,68], as well as the Synthesis process guidebook [49]. These documents as well as discussions with FAST process advocates at Lucent provide the information used to document the method.

The task of capturing and analyzing the SDA method requires more work than the capture of FAST. This is because the SDA method was neither well defined nor well understood by the members of PacSoft. Before this thesis was written, the SDA method was only partially documented. Most of the knowledge was implicitly understood by some member or members of PacSoft. However, the information was not shared or validated among the group. In fact, it was discovered during the method capture that some members of Pacsoft disagreed on significant parts of the definition and others didn't believe there was a complete method to document. These problems added to the difficulty of defining the SDA method. A major contribution of this thesis is the capture, refinement, and expansion of the SDA method.

There are several tasks that must be carried out to accomplish this. First of all the current state of the method must be understood. This provides a basis for refinement and improvement. This task is accom-

plished by reading available literature on the method as well as by talking with members of PacSoft who helped develop the method.

Once an initial understanding of the method is achieved, the underlying concepts of the method are explored to deepen the understanding of the principles of the method. The concepts to study include software reuse, application generators, formal methods, functional languages, program language design, and semantics. This information is again obtained by reading available literature and talking to people within PacSoft.

When the process is documented it can be validated and improved as understanding grows and changes. In other words, the method is improved continuously. As the method is captured, it is validated by members of PacSoft to ensure that important concepts have not been left out or misunderstood. Based on this feedback, the method can be refined again.

To help illustrate the two methods a standard example has been adapted to highlight important features of the methods that are concerned with the integration. The example chosen is a simplification of the Floating Weather Stations (FWS) domain, which was first defined by the Naval Research Laboratories. This example is discussed in more detail in section 3.2.

As mentioned above, along with the documentation and illustration of the methods an analysis of the methods is performed. The analysis captures the strengths and weaknesses of each method. The framework for capturing the strengths and weaknesses is based on the structural similarity of the two processes. The methods have been divided into 9 high level process steps and a category for general process characteristics:

1. General
2. Domain selection / identification
3. Domain scoping
4. Information gathering / analysis
5. Domain modeling
6. Language design
7. Generator building
8. Application Engineering Environment creation
9. Application Engineering Process capture
10. Documentation

As the strengths and weakness of each method are determined, detailed integration of the two methods can proceed. However, it has already been determined that the methods will not fit together as easily as pieces of a puzzle [65]. There are gaps between the two methods that must be bridged in order for the

unification to occur. In this thesis the gaps will be studied and solutions for filling the gaps will be proposed. For example there are incompatibilities in the level of information captured by each method.

The next section provides the reader with a brief introduction to software reuse. Software reuse establishes the context for this research and for how domain-specific languages and application generators relate to other approaches in the software reuse paradigm.

2. Background

Software reuse has been actively studied since the 1968 NATO software engineering conference [23]. Seminal work includes papers by McIllroy, Dijkstra, and Parnas [23,11,33]. Over the years many software reuse approaches have developed. Examples are libraries of reusable components and languages. In this section the various reuse approaches are discussed and categorized providing a context for understanding the specific approach of domain-specific languages and application generators discussed in this thesis. Much of the information in this section has been adapted from papers by Kruegar, Mili et. al., and Neighbors [20,26,31].

Software reuse approaches can be separated into two categories:

- General Purpose
- Domain-specific

General purpose approaches include assets that are reusable across all or most problems, while domain-specific methods include assets that are specialized for a given family of software products. General purpose approaches reuse concepts that are general computer science abstractions such as control flow, abstract data types, and mathematical structures. Domain-specific approaches, on the other hand, benefit from using much narrower and deeper concepts to provide tailored artifacts for a particular domain.

Software reuse approaches can also be categorized by the assets reused. These different reuse approaches, in increasing level of reuse, are:

- Ad hoc
- Components
- Architectures
- Languages/Generators
- Expert systems

Ad hoc reuse is widely practiced and always has been. Ad hoc implies that there is no process for creating or selecting reusable assets and that the assets used can be basically anything that was used before, i.e., requirements, code, or processes. The assets were originally created without the intent of being reused. The approach consists of developers taking work products that they are familiar with and adapting them to their current project. This process requires intimate details about the products being reused as selecting products is based on previous use, and the products must typically be modified to ensure proper behavior.

Libraries of components or component specifications are also widely used in practice as illustrated by object hierarchies, and Fortran and C libraries. The library approach consists of reusing components or specifications for components that encapsulate a particular concept or functionality. To use the components

a developer must first select an appropriate component and then perhaps modify it in some way to have it fit into the application.

Components are typically referred to as black box and white box. Black box components require no modification; they can be used as is. White box components on the other hand require that the developer know intimate details about the implementation of the component to specialize it for the task at hand.

An architecture captures the structure of a system. It is a model of a system in terms of its components and connectors. When a developer reuses an architecture, they can also reuse the components, therefore architectures provide for higher reuse than just components. When a developer reuses an architecture, they must first select an appropriate architecture and then tailor it to the particular system being built. As with components, this may require intimate details of the implementation.

Languages and application generators provide for even more reuse than architectures. Languages abstract the low level implementation away from the application engineers and provide them with a new, higher level language with which to program or specify products. With languages as reusable assets, developers define a system in the high level language and then automatically transform the definition into a lower level working application. The developers must learn a new language, but the implementation details of the lower level code are no longer needed.

Expert systems are the final level of reuse approaches. An expert system abstracts the problem to such a level that the client can now specify a system and the application engineer is taken out of the process.

These two classification schemes for reuse approaches, general versus domain-specific, and the levels of reusable assets, are used in this section to provide a framework for understanding the problems and research in the software reuse field.

2.1. Ad Hoc

There is not a big distinction between general purpose and domain-specific ad-hoc reuse. In both cases reusable assets are not created with the intent of being reused. Therefore methods do not exist that describe how to create and use ad hoc reusable assets. If there were, then they wouldn't be ad hoc.

In general-purpose ad-hoc reuse, the developer selects artifacts to reuse from any area that they are familiar with. On the other hand, with domain-specific ad-hoc reuse, the developer chooses artifacts from projects similar to the current project they are working on.

One problem with the ad hoc reuse approach is that errors are typically introduced into the new systems being built because of inadequate understanding of the assets being reused. Also there is no support for selecting which assets to reuse.

2.2. Components

Implementing or specifying components to be reused has been prevalent in the reuse community since McIllroy's paper "Mass produced software components" [23]. Functions, abstract data types, and objects are examples of the components captured. General purpose methods advise building libraries of components that are generally needed by the software industry. Examples of libraries include C run time libraries, FORTRAN math libraries, and general object hierarchies as provided with Smalltalk or C++.

Domain-specific reuse approaches for building components advocate building specialized libraries of components for an application domain. These libraries of components are then accessible by the application engineers working in the specific domain. Prieto-Diaz has developed a method for analyzing a domain with the intent of creating a library of domain-specific components [38].

The problems associated with general purpose libraries of components are typically classification and search problems, as the libraries are typically very large. Once the components are created, how do developers know that they exist and where to find them. The developers can't reuse a component if they don't know about it, or can't find it. Hierarchies and graphs are being used in general to group components to aid developers with locating appropriate components. Examples of these grouping mechanisms can be found in the object browsers of object-oriented languages such as Smalltalk and C++. There are also searching schemes for locating appropriate components based on keywords [20].

Even though domain-specific libraries of components aren't typically as large there are still problems of selecting which components to use. Domain-specific architectures are one means being studied to alleviate this problem. The developers only need to know about the architectures. These assist the developers with which components to gather.

For both general purpose and domain-specific components there are also problems with selecting an appropriate level of abstraction for components. If the components contain low level abstractions and are specialized then there must be many components and the benefits of reuse are not being taken advantage of. Many components are then either inapplicable or they have to be modified extensively to be used. However if they are very abstract, then they may be hard to understand as applicable to the problem at hand, or they may not do very much [31].

2.3. Architectures

Garlan and Shaw describe an architecture as a system's structure or organization [45]. The architecture defines the relationships among the major components of the system with system specific details abstracted away. The abstractions of the system specific details is what makes architectures reusable. For

each system to be developed from the same architecture, the architecture can be used as a starting point for the low-level design.

General purpose architecture research aims at classifying styles of designs used to build systems. Examples of general purpose architectures are pipe and filter systems, event-based or implicit invocation, and layered architecture styles. These high level models of systems can be used for many domains that use similar breakdowns of their components[45].

Domain-specific architectures aim to create system designs that are more detailed than general purpose architectures. Both the components and the connections are domain-specific instead of general purpose, therefore providing more details. Many domain engineering methods aim at analyzing a domain to come up with an architecture. Feature-oriented domain analysis at the Software Engineering Institute and the government's domain-specific software architectures program are just two examples [47,17,5].

A problem with architectures is representing an architecture so that it is easy to understand and use. There are currently many architecture description languages available that not only capture the architecture, but generate systems from architectural descriptions [9].

2.4. Languages/Generators

Languages and generators are conceptually above architectures and prevent the user of the language from accessing the implementation. This categorization illustrates that High Level Languages (HLL), such as C or Ada, are really a general purpose, generation approach to reuse. The user of the HLL can develop programs for many domains yet they can't get into the compiler and change how the constructs were implemented in assembly.

Very High Level Languages (VHLL), or sometimes called 4th Generation Languages (4GL), incorporate high level general abstractions of programing concepts explicitly in the language. They are based on application-independent mathematical and computational abstractions [26]. This makes them difficult for the average developer to use, as the mathematical background is required. In the past VHLLs have produced inefficient code. Kruegar states that "the primary concern in VHLLs is not efficiency in program execution but rather efficiency in implementing and modifying programs" [20]. They are very good for prototyping but they have problems if you want to produce real software.

Transformation systems are built on top of VHLLs. They are a domain-specific approach to reuse. They require domain-specific as well as general purpose re-write rules, and component libraries that help improve the performance of VHLL generated code. These are typically compiled with human interaction. The humans provide decision support when the transformation system is not sure which transformation to choose.

Application generators, also known as component generators, are also a domain-specific reuse approach to languages. They are like programming language compilers because they take specifications as inputs and automatically translate them into executable programs [8]. However, they "differ from traditional compilers in that the input specifications are typically very high-level, special-purpose, abstractions from a very narrow application domain" [20]. "Using specialized languages is an alternative to using program libraries. The languages serve as a general description that limits how the software components of the domain may be combined" [31].

Application generators are built on top of a software architecture or any other appropriate combination of domain models. The application developers don't see the constructs of the domain, they only see the application generators domain-specific language. The constructs are automatically connected by the application generator. This leaves the application developers more time to concentrate on 'what' the system should do rather that 'how' the system should do it because the algorithms and data structures are automatically selected [20]. Kruegar states that "Application generators generalize and embody the commonalities so they are implemented once when the application generator is built and then reused each time a software system is built using the generator" [20].

As documented by Cleaveland, application generators have many advantages [8]. They can:

- reduce programming errors because problems are specified at a higher level and trusted components can be reused
- be used by non-programmers who are familiar with the domain because they raise the level of abstraction to specification in terms and notations of the problem domain
- be used to prototype applications because the time needed to produce application is reduced
- be used to implement standards so that they are adhered to

However, Kruegar states that "a difficult challenge for implementors of application generators is defining the optimal domain coverage and the optimal distribution of domain concepts into the fixed and variable parts of the abstractions" [20].

Even though the benefits are high for using application generators, the cost is also high. To benefit from the work used in building an application generator it must be clear that the domain selected will have lots of use for an application generator. Kruegar outlines three characteristics necessary for a application generator to pay off.

- many similar software systems are written
- one software system is modified or rewritten many times during its lifetime, or
- many prototypes of a system are necessary to converge on a usable product.

This is where the research of this thesis fits in. Both SDA and FAST focus on DSLs.

Application generator generators are being studied by some groups. These are general purpose generators that take a domain-specification and produce a domain-specific application generator. Draco and GenVoca are examples of application generators generators.

Mili et al say that application generator generators can be built, but they say "it is difficult to design a development methodology for application generators that is appropriate for all application generators because it depends very heavily on the application domain" [26]. They say that when viewed as translators they have a standard architecture. "Designs and implementations of translators are so well understood and standardized that application generators can be built using application generators" [26].

2.5. Expert Systems

Expert systems are similar to application generators and languages in that they abstract away from the low level implementation of the problems. However, expert systems aim to replace the application engineer instead of provide tools for the application engineer to use. In other words they must capture all knowledge that the application engineer has, not only certain parts. Where application generators still aim to encapsulate the common features of a product line and provide higher level means of capturing the variability.

Expert systems have been studied for years by the artificial intelligence community in an attempt to emulate human expertise in well defined problem domains. No general purpose systems have been built for creating applications due to the complexity and flexibility in the field. Also domain-specific expert systems may be built in limited domains where the rules for creating a system are well known and definable. However due to the amount of application engineering knowledge present in most domains they are few and far between. For this reason, this thesis does not study expert systems in detail.

2.6. Summary

All these methods are represented in Table 2 categorized by the two classifications for software reuse methods.

Table 2: Software Reuse Methods

	Domain-specific	General Purpose
Ad Hoc	Design and Code Scavenging	Design and Code Scavenging
Components	Object-Oriented Methods	Object-Oriented Methods
	Libraries	Libraries
Architectures	Domain-specific architectures	Client/Server and Layered system architectures
Generators	Application Generators	High Level Languages
	Transformation Systems	Very High Level Languages
		Application Generator Gens.
Expert Systems	Domain-specific	General purpose

3. The FAST and SDA Methods

In this chapter, the two domain engineering methods: FAST and SDA are captured in detail. Before they are introduced, however, some assumptions that bound the scope of domain engineering as used here, are discussed. Also an introduction to the working example is included to familiarize the reader with the domain of Floating Weather Stations.

3.1. Assumptions

Currently the domain engineering methods discussed in this paper are not applicable to every domain. They are only applicable to domains where expert knowledge already exists. The methods are not currently robust enough to handle new unexplored domains where no expert knowledge is present.

These domain engineering methods assume that the chosen domains meet certain criteria. First of all it is assumed that the domains are mature. Mature domains have been around long enough so that they are well understood. The problems in the domains are known as well as the solutions to the problems. If the domains aren't mature then the domain engineering methods must supply more structure to establishing a domain.

The second criterion is that the domains must be stable. While stable domains may change or evolve over time, their underlying structure and basic concepts do not change. If the domains are not stable then the domain engineering efforts may have to be completely redone whenever the domain changes. The benefit from domain engineering comes from the reuse of the artifacts. If the artifacts have to be restructured often then the return on investment of the domain engineering effort may not be worthwhile.

The existence and availability of domain experts is another criteria for domains to possess before a domain engineering effort is undertaken. Domain experts have worked on problems in the domain and they understand all or part of the domain in-depth. Without domain experts the domain engineers must learn the domain inside and out so that they know how to build artifacts that are valid for all domain instances.

All of the domain experts may not agree exactly about every notation and concept in the domain. This is because most domains are well established scientific domains where objective knowledge exists independent of individuals. The domain experts may use different notations for the same concepts or the same notations for different concepts. They may even use different notations and different concepts [43].

These differences are most likely to occur in domains where the domain experts are spread out and don't communicate. When the domains are within one organization the domain experts are more likely to have learned the same notations and concepts. Especially if they work together and need to be able to com-

municate.

This thesis will focus on domains where there is only one set of notations and concepts that the domain experts agree on. It is not difficult to extend the methods for multiple views of the domain however it might make the method more complex and harder to understand.

Finally it is assumed that the social problems of introducing new methods and technologies into the domain are overcome. The companies where the domain engineering efforts will take place must support the activities. The necessary resources must be allocated for the domain engineering effort to succeed. Also the people within the organization must be willing to accept the new technologies otherwise they may not be used once they are produced. It is beyond the scope of this thesis to discuss how to bring about change in an organization, but these are important issues that must be resolved.

3.2. Example

The example domain used for illustration of certain parts of the methods throughout this paper is that of the Floating Weather Stations (FWS). The FWS systems are a family of free floating buoys that provide weather data to air and ship traffic at sea (Figure 3). The buoys collect data on air and water temperature, and wind speed through a variety of sensors. The readings are stored and averaged by the FWS. Each buoy has a radio transmitter for broadcasting weather information to satellites which pass the information along to passing air and sea traffic [48].

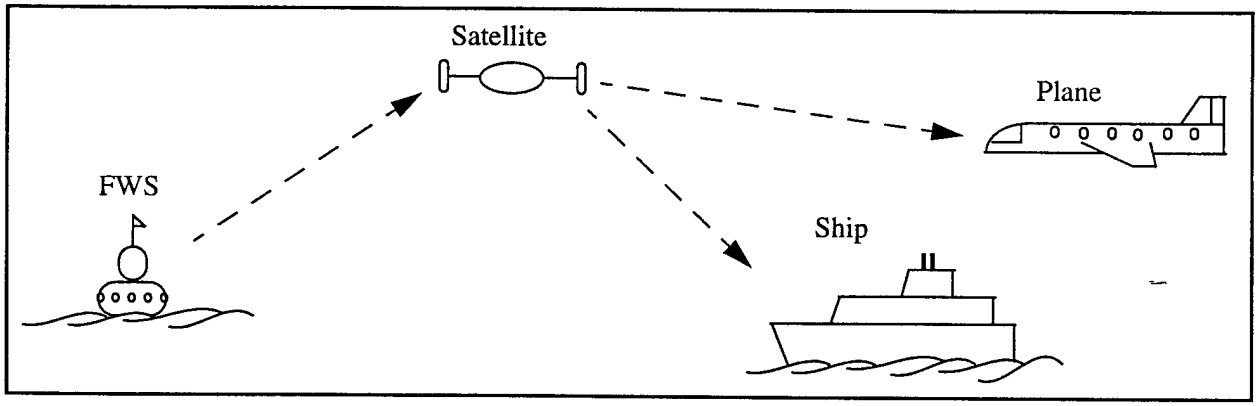

Figure 3: Floating Weather Station in Context

This example was developed at the Naval Research Laboratories (NRL). It has been studied and used for many years in classes teaching structured design. There are several example implementations [cite].

The FWS domain is a good candidate for domain engineering because there are many possible systems to be built and they are very similar. It is also a mature and stable domain with much information

available about it. Unfortunately since it is not a real domain there are no domain experts.

Currently Lucent, and PacSoft are working to refine the FWS example into a pedagogic example for domain engineering. This example is significantly scaled down from the original problem, however, it is still sufficient for illustrating the approaches. The information about the FWS systems in this paper is derived from the previous work at the NRL and the current work by Lucent and PacSoft [4].

3.3. Family-oriented Abstraction Specification and Translation (FAST)

3.3.1. Introduction

FAST is a systematic process for analyzing software families and for developing specialized languages, tools, templates, libraries and processes for a software family, or domain [4,67]. These assets are developed to aid application engineers in producing applications in the domain efficiently. FAST is based on the Synthesis method developed by Cambell, Faulk and Weiss at the Software Productivity Consortium (SPC) [7,49].

As mentioned previously, FAST includes a well documented and validated commonality analysis subprocess, which groups domain experts together to gather, analyze and capture domain information. Other steps in the FAST process aim to build on this analysis to define and develop an application modeling language, an application engineering environment, and a standard application engineering process. These steps are not as well defined in the FAST literature as the commonality analysis.

A goal of this research is to build on the strengths of FAST, particularly the commonality analysis, by providing more support for language design through the definition of the SDA method, and the integration of the two methods.

3.3.2. Fundamental concepts

David Weiss, who worked on the Synthesis method at the SPC, has been working to evolve and mature the FAST method at Lucent. One of the goals of the domain engineering research at Lucent is to develop a domain engineering method that is accessible to engineers. Accessible technology can be transitioned readily to the engineers and they can apply it without the assistance of researchers or experts in the technology. A conscious effort has been made to keep the introduction of formality and new concepts to a minimum so that it is straightforward for the engineers to learn and apply the method on their own. However, the method is also intended to be adaptable in order to provide room for tailoring based on the skill set of the domain engineering team.

The FAST method has been applied to over a dozen domains at Lucent, and even more projects are being planned. The engineers who have used the FAST process are relaying its success to other engineers. At Lucent, this is the most effective way to get a method adopted. If the engineers have faith in a product then others will trust them and follow suit.

3.3.3. The FAST Method

The information presented here is adopted from the Synthesis guidebook, as well as documents, course notes and the process definition of FAST in Pasta [4,67,68,49]. Additional information was gathered in informal discussions with FAST researchers and developers using FAST during a collaboration project between PacSoft and Lucent. A high level view of the method is shown in Figure 4. A detailed description follows.

3.3.4. Overview of FAST Method

An overview of the FAST process and the products created during FAST activities are outlined below.

3.3.4.1. FAST Process

- Analyze family
  - Qualify family
  - Analyze commonality
  - Define decision model
  - Design application modeling language
    + Design family
    + Define composition mapping
- Implement family
  - Design the application engineering environment
  - Create a standard application engineering process
  - Implement the application engineering environment
  - Document the application engineering environment

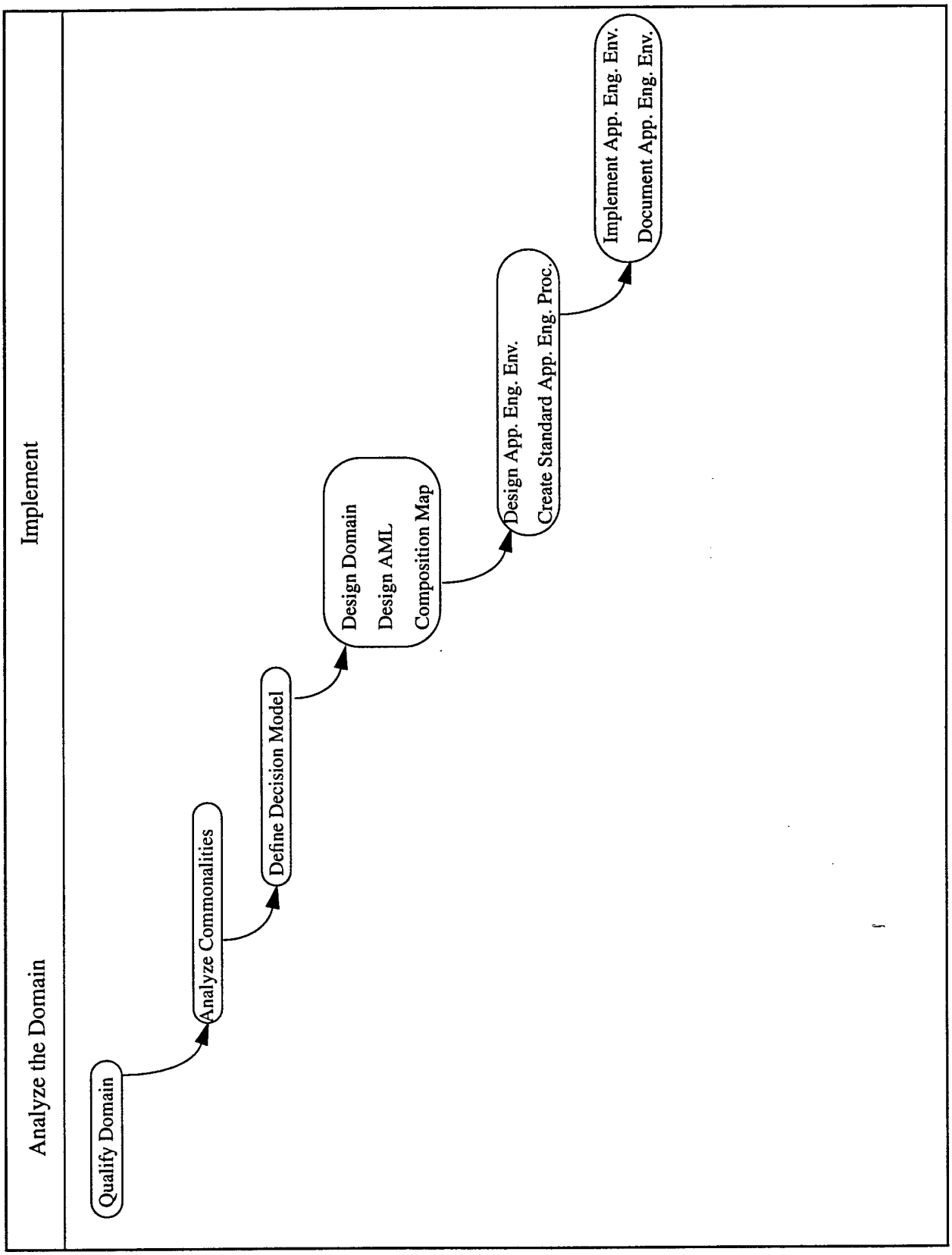

Figure 4: FAST Process

3.3.4.2. FAST Products

- Domain model
  - Economic model
  - Commonality analysis
  - Decision model
  - Application modeling language
  - Family Design
  - Composition mapping
- Tool set design
- Standard application engineering process
- Library
  - Documentation template
  - Code template
- Generation tools
- Analysis tools
- Documentation
  - User's guide
  - Reference manual
  - Training material

FAST I. Analyze Family

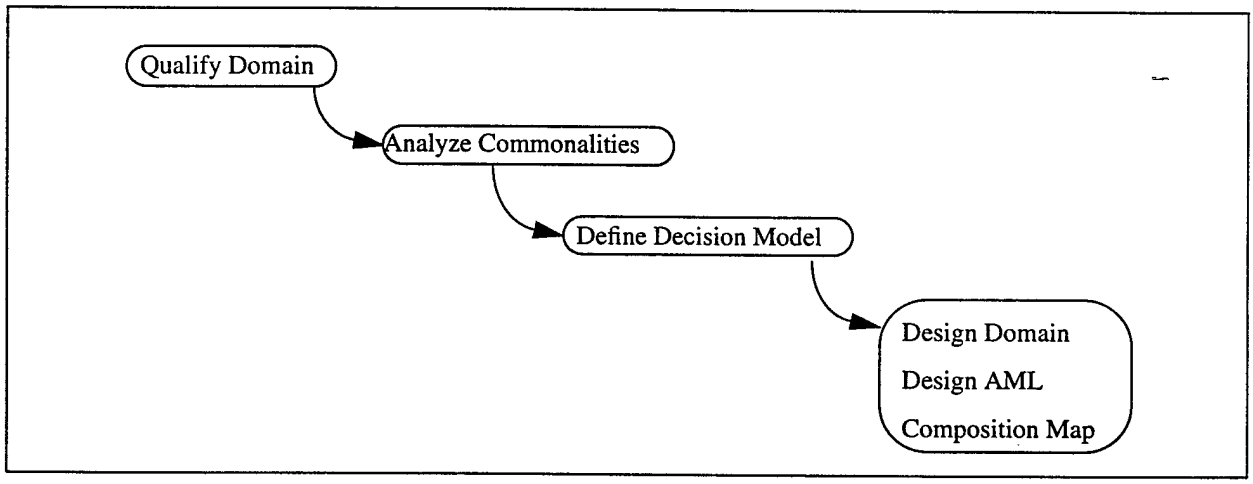

Figure 5: FAST Analyze Family Phase

The analyze family process includes activities needed to understand the domain and what constitutes solutions in the domain. The purpose of analyzing the domain is to produce or refine a specification for the application engineering environment. The output is called the domain model, which is an aggregation of all the products produced in these activities.

FAST I.I. Qualify Domain

Qualifying the domain determines whether a domain engineering project with the selected domain is likely to save time and money in the future. The domain is evaluated through the Economic Model. The economic analysis identifies if it is cost-effective to pursue the domain engineering project on the domain.

The Economic Model contains a rather simple formula for evaluating the domain. Four estimates are collected for the evaluation. One estimate is the cost of producing applications in the domain using the current technology. Another estimate is the cost of performing the domain engineering on the domain to produce an application engineering environment. The cost of producing applications using the application engineering environment is also estimated. Finally, the number of future applications in the domain is estimated.

These estimates are used to determine if the cost of producing the future applications with the application engineering environment along with producing the environment is less than the cost of producing the future applications using the current technology. If this is the case, then the domain engineering project is deemed profitable and should be pursued.

If a full scale domain engineering project is not considered valuable, then further estimates can be collected to evaluate if a partial effort would be worthwhile. These might indicate that a partial effort, say only the analysis, is still beneficial even without an application engineering environment. However, these estimates are not well defined.

FAST I.II. Analyze Commonality

Commonality analysis activities expand and share domain understanding, structure the domain information, and explicitly capture the knowledge within the domain. The commonality analysis is the basis for all other products and activities in FAST.

The commonality analysis process consists of moderated group discussions among domain experts. The discussions iterate over the sections of the commonality analysis document. The commonality analysis document is reviewed and refined by the group throughout the commonality analysis sessions.

The commonality document is partitioned into sections. The sections include:

- an introduction,

- an overview of the domain,
- a dictionary of terms,
- a list of commonalities,
- a list of variabilities,
- a list of the parameters of variation, and
- a list of all the issues that arise
- appendixes

The commonalities and variabilities capture the core of the domain model. Their structure provides a high-level view of the domain, and their content captures the domain expertise. The other products support domain modeling.

The introduction states the objectives and expected results of the CA project. It also identifies the intended audience of the complete CA document. This section really captures the motivation for performing the analysis. It's purpose is to inspire the domain analysis team during the analysis and justify the analysis to readers of the analysis.

The overview section captures a high level introduction to the domain being analyzed. The basic concepts of the domain are identified and described. The overview also specifies the scope of the domain. The scope bounds the domain and captures the connections to other neighboring domains. In addition, the overview section captures any specific areas or issues that are to be addressed during the analysis.

The dictionary of terms explicitly captures and thereby standardizes the definitions of the key terms in the domain. Technical terms as well as non-technical terms are defined in the dictionary. Abbreviations and acronyms are also captured. The dictionary serves as the reference to sort out any ambiguities that arise.

Commonalities are assumptions that are true for every product instance in the domain. Attributes that every instance must have and functions that every instance must perform are examples of commonalities. The list of commonalities explicitly captures the information that pertains to all instances in a domain. The list is structured according to the structure of the domain. The structure provides additional information about the domain. An appropriate structure is determined by the domain experts.

Variabilities, on the other hand, are assumptions about how instances of the domain differ. They define where and what choices application engineers have when creating domain instances. Optional attributes or functions are examples of the variabilities that are captured during the analysis. The variabilities are also structured. However, the structure can differ from the structure used for the commonalities.

The parameters of variation capture the range of values for the variabilities. For each variability, the alternative values of the variability are identified along with a default value (if any) and the binding

time, which is the time at which the value is known or can change. The binding time can be compile time, run time, or execution time.

The issues section is used to capture any issues that arise during the analysis of the domain. Issues include domain topics where there is disagreement about how to proceed, or where further investigation is needed to understand the implications of a decision. The possible alternatives of an issue are captured along with a description of the issue. As issues are resolved, the chosen alternative is identified and the rationale for the decision is documented.

The appendixes contain additional information that arises during the analysis that does not fit into the commonality analysis sections, but that the domain experts believe is pertinent and should be included.

An initial CA example is shown in Figure 6.

The commonalities and variabilities sections of the commonality analysis serve as a basic model of the domain. Therefore the structure of these sections is significant. The grouping of the commonalities and variabilities provides higher level understanding of the domain structure, and makes the commonality analysis easier to produce and review. Throughout the CA process the search for abstractions in the domain is pursued to find an appropriate structure for the lists of commonalities and variabilities.

The lists may first be structured according do some example categories. The FAST documentation suggests discussing the behavior, outputs, output conditions, and external interfaces to other domains, devices, and the platform. These are typical areas that must eventually be categorized somewhere in the commonalities and variabilities sections.

The FAST course notes provide tips on capturing commonalities and variabilities and on selecting an appropriate structure for the domain model. One of the tips is to compare the domain of focus to similar domains. Similar domains help the domain of study to be seen in different lights. They may help uncover things about the domain that would not have been thought about otherwise. Another one of the tips is to ask naive questions about the domain. Some example questions are: is this always true, and what if this changes. Testing each commonality for associated variabilities is another way commonalities and variabilities can be generated. For example if a commonality is that every member must have a sensor, then an associated variability would be the different types of sensors.

The CA document is internally reviewed by the team members once they have completed an iteration on it. The purpose of the review is to inspect both the content and structure of the document. The definitions in the dictionary of terms are checked for completeness and consistency of usage in the document. The commonalities, variabilities, and parameters of variation are inspected for correctness and completeness. These are informal checks that are done manually; there is no tool support for them at this time.

Commonality Analysis for Floating Weather Stations

Introduction

Floating Weather Station (FWS) buoys are deployed at sea and drift around, periodically reporting the current wind speed and temperature. Each member of the FWS family contains an onboard computer that controls the operation of the buoy while it is at sea.

The purpose of this analysis is to provide the following capabilities for the FWS family of buoys:

- A way to specify the configuration of a particular buoy.
- A way to generate, for a specified buoy configuration, the software that controls a buoy while it is at sea.

Overview

This commonality analysis is concerned with the following issues:

- What equipment configurations should be accommodated?
- What computing platforms would be used on buoys?
- What capabilities will be needed to make buoys sufficiently reliable to perform their missions?

Dictionary of Terms

Term	Meaning
Sensor period	The number of seconds between sensor readings
Transmission period	The number of seconds between message transmissions
Wind speed	The speed of the wind in knots: nautical miles per hour
Wind temperature	The temperature of the wind in degrees Celsius
Water temperature	The temperature of the water in degrees Celsius

Figure 6: FWS Commonality Analysis

Commonalities

The following statements are basic assumptions about the FWS domain, i.e., they are true of all FWS systems.

C1. At fixed intervals, the buoy monitors the *wind speed, water temperature and wind temperature* at its location.

C2. The buoy is equipped with one or more sensors that monitor *wind speed.*

C3. The buoy is equipped with one or more sensors that monitor *wind temperature.*

C4. The buoy is equipped with one or more sensors that monitor *water temperature.*

C5. The buoy is equipped with a radio transmitter that enables it to send messages.

C6. At fixed intervals, the buoy transmits messages containing an approximation of the current wind speed, and temperature. The values reported are the averages of all the sensor readings.

Variabilities

The following statements describe how FWS buoys may vary.

V1. The number of each type of sensor on a buoy may vary.

V2. The sensor period of the sensors on a buoy may vary.

V3. The transmission period of messages from the buoy may vary.

V4. The sensor hardware for each type of sensor on a buoy may vary.

V5. The transmitter hardware on a buoy may vary.

Parameters of Variation

Parameter	Meaning	Domain	Binding Time	Default Value
P1. Sensor Counts	Number of each type of sensor	[1..5]	Specification	1
P2. Sensor Period	Sensor period	[1..600] seconds	Specification	5
P3. Transmit Period	Transmit period	[1..600] seconds	Specification	10

Figure 6 (continued): FWS Commonality Analysis

The structure of the document is also inspected during the review. All of the sections must be accounted for. The structure of the commonalities and variabilities sections is also inspected. The abstractions chosen are once again reviewed to test if the information is grouped according to the abstractions of the domain.

Following the internal review, an external review is held which inspects the same issues as the internal review. The external review is done by engineers who are knowledgeable in the domain area, but who have not participated in the commonality analysis, so that they can inspect the content of the CA as well as the structure with a fresh eye.

FAST I.III. Define Decision Model

The decision model captures the set of requirements and engineering decisions that an application engineer must resolve to describe and construct a product. It is used for designing the application modeling language as well as determining the standard application engineering process.

The decision model is a description of the process that the application engineers follow to specify a new family member. It defines what variabilities must be specified by the application engineer, and what values they can have. The decision model also identifies any sequencing for the decisions. A complete set of decisions identifies a new family member. The decision model can be directly derived from the parameters of variation.

FAST I.IV. Design Application Modeling Language (AML)

The application modeling language will be used by application engineers to specify product instances. Family member descriptions in the application modeling language are then generated into working applications and their documentation. The application modeling language can be implemented using either a compositional or a compiler approach. The tasks followed depend on which approach is used.

An Application Modeling Language (AML) is designed which will be used by the application engineers to specify family members. Therefore the decisions that are identified by the decision model must be expressible in the AML.

Constructs of the language must be defined and the syntax for each construct of the language is captured by a set of BNF production rules, or some equivalent.

Behavior Hiding Module

Message Generation

- Service: Periodically retrieve data from the Data Banker and transmit it.
- Secret: The way in which the other modules are used.

Message Format

- Service: Support construction of an output message.
- Secret: The message format.

Environment Hiding Modules

Sensor Device Drivers

- Service: Provide access to the various sensors. There may be a submodule for each sensor type.
- Secret: The details of the sensor hardware.

Transmitter Device

- Service: Provide access to the transmitter
- Secret: The details of the transmitter hardware.

Software Decision Hiding Modules

Sensor Monitor

- Service: Periodically retrieve data from the various sensors and deposit in the Data Banker.
- Secret: The way other modules are used.

Data Banker

- Service: Store the most recent data.
- Secret: The algorithm and data structures used.

Averager

- Service: Process the current Data Banker data to produce a current wind speed and temperature estimate.
- Secret: The algorithm used.

Figure 7: Domain Design

When the compiler approach is used some form of semantics for the language constructs are captured. When compilation is used, a design is created where only the abstract modules and the parameters of variation used to generate the concrete modules are specified in the design.

If the composition approach is used then a domain design is created. The domain design is an architecture for the family. The architecture is created by organizing the software into information hiding modules. The commonalities are used to organize the information that is included in the design. Things are grouped together into modules to make the design easily changeable. An example domain design is featured in Figure .

If multiple components interact, then a composition mapping is needed. It specifies how values in an application model provide values to the parameters of variation used to instantiate abstract modules. It is the mapping between the AML and the components specified in the design. The composition mapping cannot be finished until the Application Modeling Language is specified.

The outputs of this activity are an application modeling language design, a domain design and a composition mapping if required.

FAST II. Implement Family

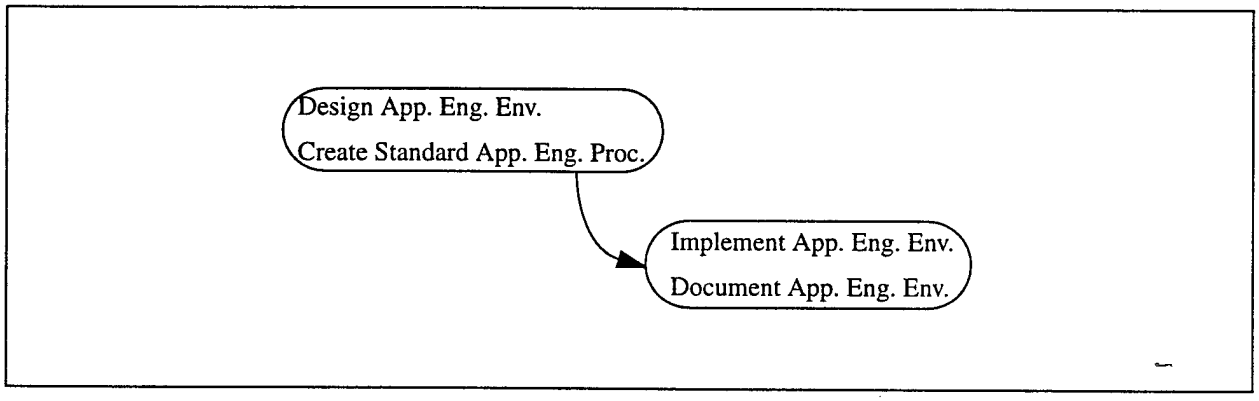

Figure 8: FAST Implement Family Phase

Implementing the domain consists of developing or refining an environment that meets the specification given by the domain model.

FAST II.I. Design Application Engineering Environment

Once a language is defined, artifacts can be built to help support the application engineers in using the language to specify new family members. These artifacts are brought together in a new application engineering environment. The purpose of the environment is to make rapid specification of new software

products. Therefore the environments must be designed for the application engineers who will use them. The environments are not just for generating code. They provide facilities for integrating the reusable code with the variable specification.

Many different tools may comprise the application engineering environment. Both generation tools and analysis tools can be designed. Generation tools take the application engineers specifications in the AML and transform them into source code and documentation. Analysis tools, on the other hand, help the application engineers by analyzing specifications and providing useful feedback. Analysis tools may include consistency and completeness checks, simulations, performance estimates, comparators of different models, and other forms of analysis. The tool set also includes supporting tools, such as editors and window managers.

During tool set design, libraries of common assets are also specified. The libraries may include both code templates and documentation templates. As the names suggest, code templates are used in producing the component(s), and the documentation templates are used for producing the manuals for the generated component(s).

The outputs of these activities include the generation and analysis tool designs and the code and documentation templates

FAST II.II. Create Standard Application Engineering Process

In this step a standard application engineering process is developed and documented, which guides the application engineers through the steps of developing an application in the new environment.

The application engineering process for a domain is the process whereby an application engineer uses the application engineering environment to make the decisions identified in the decision model. Because the decisions and tools are different for different domains, the engineering process varies from domain to domain.

The standard application engineering process is created from the decision model, which captures the engineering decisions to be made, and the application modeling language design, which captures the abstractions used to specify family members.

FAST II.III. Implement Application Engineering Environment

After the design is finished, implementation can begin. Implementing assets for a domain consists of constructing and integrating all the assets in the application engineering environment. The environment is built to the specifications determined during design.

Implementing the environment includes building or finding the tools that are part of the environ-

ment and the data on which those tools operate, e.g., the code and documentation templates that are used by the tools to generate an application. Implementing the templates consists of writing the templates in a suitable language to meet the specifications produced as part of family design.

When using the compilation option to transform a specification into a working component, a compiler must be written for the application modeling language. In the compositional approach the system composition mapping that composes applications from templates in the library must be implemented. For each decision in the application modeling language the mapping identifies the templates in the library needed when the decision is made. The composer implements the mapping by parsing the program to get the values of the decisions, retrieving the corresponding templates from the library, instantiating the templates with the values supplied for the decisions, and integrating the instantiated templates.

Analysis tools also need to be implemented. These analysis tools should be implemented for the specifications developed during the design phase.

The outputs of this step include the library of documentation and code templates, the generation tools, and the analysis tools.

FAST II.IV. Document Application Engineering Environment

This purpose of this step is to produce the documentation needed by the application engineers who will use the application engineering environment. It includes users guide and reference material, training material, as well any other necessary information.

Documentation of the application engineering environment including a users guide, reference material, and training material are created from the designs of the environment, and the language as well as the standard application engineering process.

The outputs of this activity include a users guide, reference material, and training material.

3.4. Software Design Automation (SDA)

3.4.1. Introduction

SDA is a principled, semantics-based approach to language definition and implementation, being developed at the Pacific Software Research Center (PacSoft). SDA integrates concepts from formal semantics, functional programming, and type theory to produce domain-specific languages with provably correct implementations, and statically checkable global properties.

SDA is a refinement and extension of the Software Design for Reliability and Reuse (SDRR) method previously studied by PacSoft. Related work by PacSoft includes the SDRR papers by Keiburtz, Hook, Walton and others [6,18,63].

SDA has been designed to augment existing domain engineering methods, such as, FAST and Feature-oriented Domain Analysis, with language design and implementation support. Therefore, SDA is not itself a complete domain engineering method. SDA assumes a domain engineering method will be used in conjunction that provides support for selecting a domain and capturing the knowledge in a structured form. SDA provides techniques to model the knowledge needed for language design, to design the language, to specify the precise semantics of the language, and to implement the language.

3.4.2. Overview of SDA Method

An overview of the SDA process steps and the SDA products are outlined below.

3.4.2.1. SDA Process

- Analyze the domain
  - Capture a written definition
  - Formulate formal domain model
  - Define solution model
  - Capture the interface to the legacy environment
  - Validate
- Define the language
  - Capture the initial language definition
  - Formalize the semantics
- Implement the generator
  - Design the generator and support products

- Incrementally develop the generator and support products

3.4.2.2. SDA Products

- Written domain definition
  - Problem statement
  - Architecture and interfaces
  - Requirements
  - Workflow analysis
  - Issues, alternative solutions, decisions, and rationale
- Formal domain model
- Solution model
- Environment interface document
- Language definition
  - Initial language definition
  - Language semantics
- Generator and support product's designs and plans for building
- Generator and support products

3.4.3. Fundamental concepts

After decades of research, language design remains a difficult activity requiring diverse skills, experience and judgement. SDA advocates a particular approach to language design that produces typeful, formally defined languages. In addition, the language should be principled, that is it should incorporate the language design principles articulated by Hoare, Tennent [54], and others. It should be typed, support modularity and reuse, have internal regularity, and, as suggested by Einstein, be as simple as possible (but no simpler).

In addition to the general principles of language design, SDA recommends the following principles for DSL design:

- Focus on a declarative description of the problem, not an imperative description of the solution. Language design driven by people with intimate knowledge of a particular solution is often overly biased toward that solution. An excellent example of a radically declarative language is NASA's Amphion/NAIF system developed by Lowry and others [22].
- Imitate "good" languages. Whenever appropriate imitate the lambda calculus, Algol, Pascal, Prolog, Haskell, ML, or other well studied "good" examples.

- Never invent what you can steal. SDA advocates two primary approaches to reuse in language design. The first is to prototype domain-specific languages as embedded languages in higher-order, typed languages such as Haskell or ML. This provides a rich, well-understood type structure, basic mechanisms for abstraction, control, and modularity, and gives a flexible environment for prototyping. The second reuse strategy is to use monadic building blocks to construct the semantics of the language out of reusable pieces [27,60] (a reasonable alternative would be to use Mosses's action semantics [28]).
- Avoid becoming general purpose by accident. Be aware of the expressive power of the language you are defining. If it can express arbitrary computation make sure it does it well.

SDA uses functional compilation technology as a tool to precisely and abstractly capture the definition of a domain-specific language.

The SDA method is pictured in Figure 9. A more detailed description of the method follows.

3.4.4. SDA Method

Before the SDA method is applied to a domain, preliminary organizational and managerial work is conducted. The domain engineering team must be assembled. The domain engineering team includes a team of language design experts knowledgeable in formal methods, as well as a team of domain experts who will serve as the primary sources of information and validation. Plans and schedules are captured and periodically revisited and refined during the course of the domain engineering activities. Preliminary work must also be undertaken to allocate resources to activities. However, beyond these few guidelines, how the process is managed is beyond the scope of this thesis. It is assumed, however, that the activities are planned and tracked to ensure their completion. Without proper management and team focus, any project is in danger of failing.

The SDA method describes how to design and implement DSLs within a domain engineering project. The method consists of three phases: analyze the domain, define the language, and implement the generator.

In the analysis phase domain information from the domain analysis and domain experts is formalized in models that are used for the language design. In the language definition phase the complete specification of the language is defined based on the models collected in the first phase. This definition is captured as an interpreter expressed in a functional language, which is a prototype of the language that can be used and validated. In the implementation phase a generator is built from the validated language definition. Additional support products are also developed.

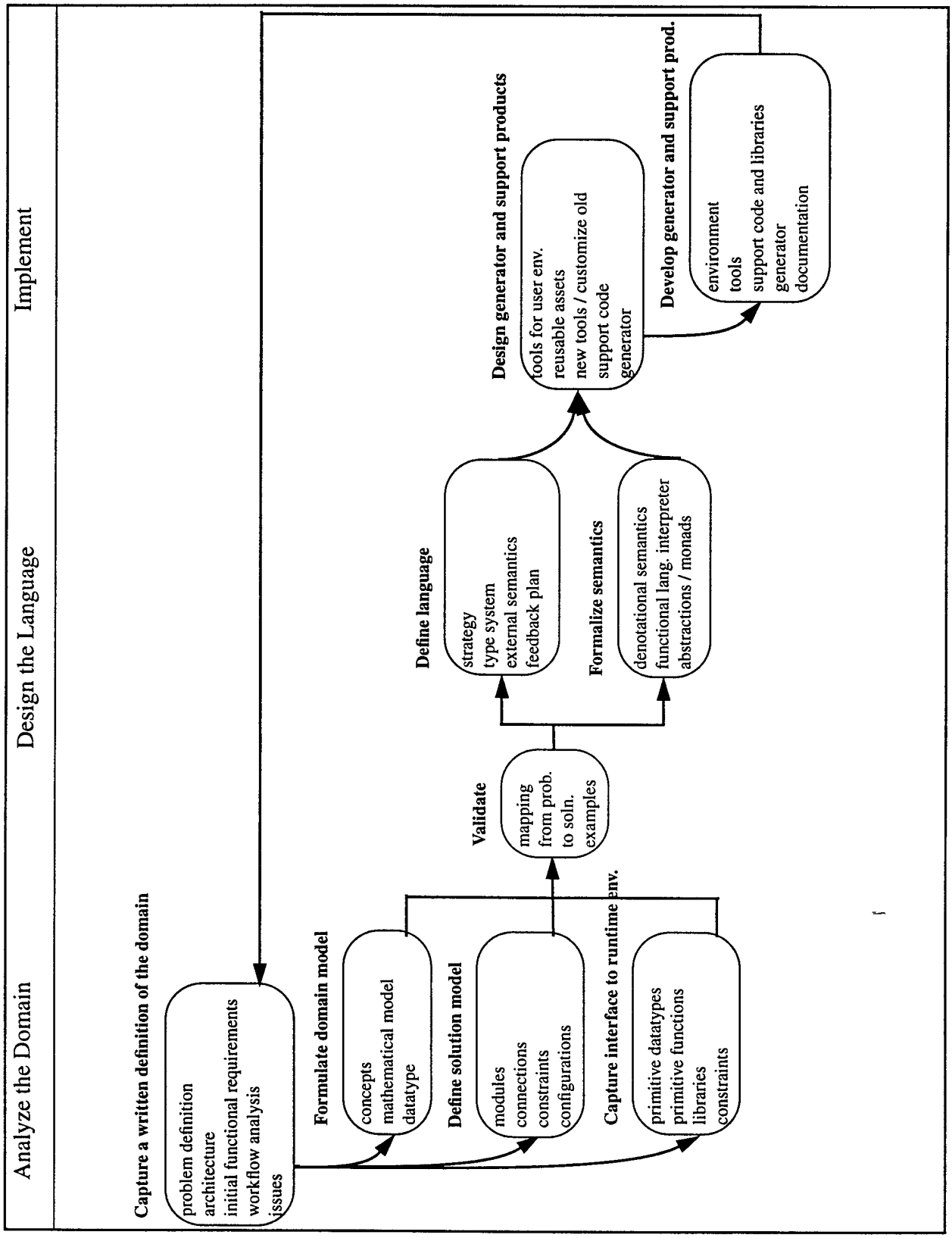

Figure 9: High-level View of SDA Process

The conceptual architecture of a component generator produced by the SDA method is given in Figure 10. The DSL is the user-visible language for specifying components. The domain and solution models are representations in a functional language of abstract models of the problem-view of the domain and the functional behavior of solutions, respectively. The concrete component is a code component that meets the requirements of the run-time environment.

The simple domain semantics map the DSL, which is only required to be expressive over distinguishing characteristics of domain instances, on to the complete domain model. The solver maps the problem level domain model on to the lower level solution model. The emit function is not required by all domains; it translates the implementation independent functional specification of the solution into a concrete artifact in the language technology of the target environment.

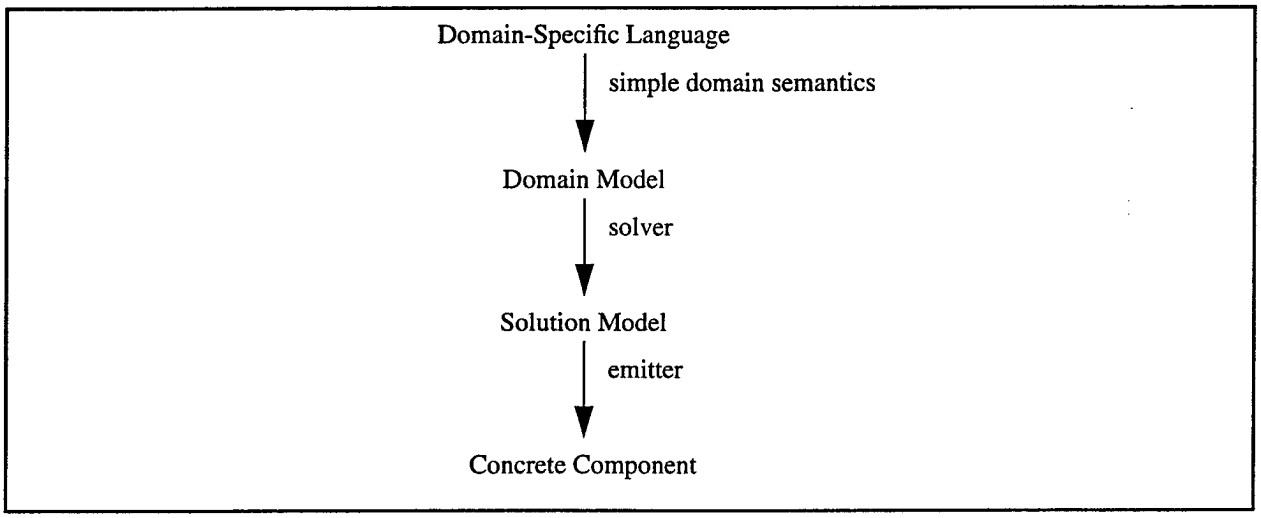

Figure 10: Conceptual Architecture of SDA generator

The simple domain semantics and solver functions are initially prototyped in SDA as semantics-based interpreters in a functional language. They may subsequently be refined into more sophisticated translators.

SDA I. Analyze the domain

Analyzing the domain is the first phase of the SDA method. SDA relies on the domain experts to provide a domain analysis document that provides most of the information needed for the language and generator design. However domain analyses vary. The following sections describe what information must be extracted from the domain analysis, or produced by the formal method experts in conjunction with the domain experts. The goal of this phase is to capture domain information from the domain analysis and domain experts as formal models that can be used for the language design.

This phase is divided into four parts, which are shown in Figure 11. In the first part information is gathered and captured as an informal written description of the domain. In the second part the domain model is defined, which captures a problem level model of the domain. In the third part lower level solution information is captured in the solution model. In the fourth part the environment interface specification is defined, which captures the runtime environment of generated code. These activities are described in more detail in the following sections.

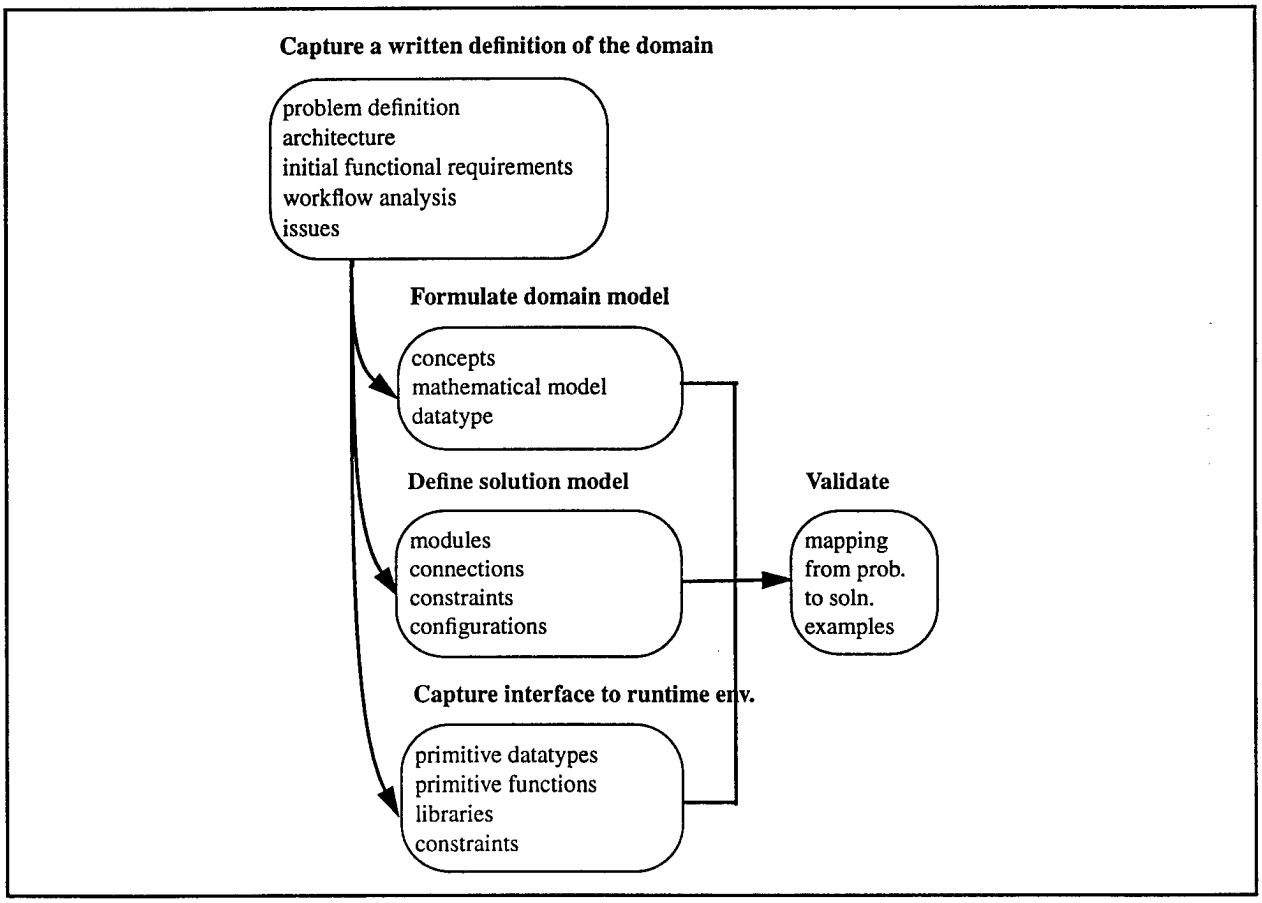

Figure 11: SDA: Analyze the Domain Phase

The activities in the analyze domain phase are primarily carried out by the team of language design experts. The language design team, however, requires access to domain experts for gathering additional information and for validating the language design team's understanding of the information captured and the models produced.

SDA I.I. Capture a written definition of the domain

The purpose of this step is to capture the key concepts of the domain and the organization. The key concepts of the domain include high level entities, functionality and constraints. The key concepts of the

organization include people, processes, documentation, and tools. This is helpful for gathering more information, establishing domain boundaries, as well as determining who to approach for providing feedback on the developing language and user environment. The domain information is refined and formally modeled in subsequent steps.

The domain being studied is a Floating Weather Station (FWS). See Figure 3.The FWS systems are a family of free floating buoys that provide weather data to air and ship traffic at sea. The buoys collect data on air and water temperature, and wind speed through a variety of sensors. Each buoy has a radio transmitter to broadcast weather information to satellites which pass the information along to the air and sea traffic.

For each buoy a system is created that controls the operation of the buoy. It must coordinate the collection of sensor information and the distribution of reports to the satellites. The system is required to be inexpensive and easy to replace because the FWS systems are considered to be disposable equipment because of the conditions in which they will be deployed. It is assumed that many buoys will disappear because of equipment deterioration, bad weather conditions, accidents, or hostile action.

Each FWS buoy will contain one or more small computers, a set of wind and temperature sensors, and a radio transmitter. Eventually, a variety of special purpose buoys may be configured with different types of sensors, such as wave spectra sensors. Although these will not be covered by the initial procurement, provisions for future expansion is required.

Figure 12: Problem Statement

There are generally many sources of information available for each domain: documentation, designs, and code for previously built systems, technical reports and books on well studied systems, users, domain experts, application engineers, managers, and other people in the community. In this step information is gathered from the domain analysis documents provided as well as from other sources as needed. The information gathered is captured in five major parts:

1. Problem statement
2. Architecture
3. Initial requirements
4. Workflow analysis
5. Issues list

The problem statement describes, at a high level, the problem area addressed by the domain. The problem statement explains the purpose of software components that are to be implemented in the domain.

The basic concepts in the domain are established along with the external functions that domain instances provide. The statement depicts how the domain interacts with neighboring domains. Neighboring domains are any entities that may interact with domain instances. They could be hardware or software. For the FWS domain an example problem statement is illustrated in Figure 12.

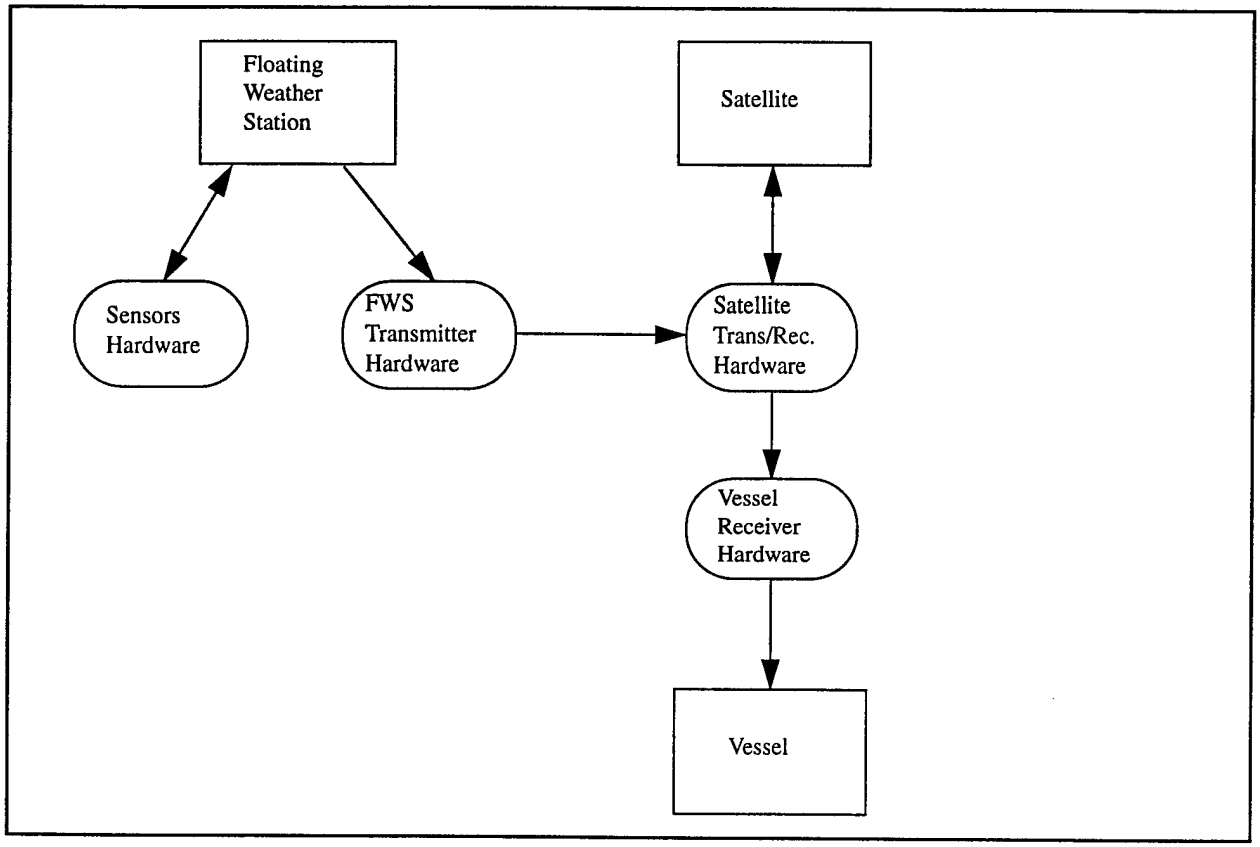

Figure 13: System Architecture

The system architecture captures the context of the domain. It defines at a high level the boundaries or scope of the domain. Scoping the domain requires that all neighboring domains agree on the boundaries and the interfaces between the boundaries. In the FWS domain there are only two neighboring domains: the sensor drivers and the transmitter drivers. The architecture description captures these boundaries and interfaces. The example system's architecture is shown in Figure 13, and the agreed upon interface between the FWS and these domains is defined in Figure 14.

Along with the problem statement, architecture, and interface information, an initial set of domain requirements are captured. The initial requirements describe the possible inputs and outputs of the domain as well as the features or actions domain instances may offer. The requirements specification also describes behavioral constraints on domain instances. The expected behavior can be captured as a high level description of how inputs become outputs. The requirements include behavioral as well as performance con-

straints. Figure 15 shows an example set of initial requirements for the FWS example.

- Sensor Interface:
  Stimuli: Sensor value
  Response: Request Sensor Data
- Transmitter Interface:
  Stimuli:
  Response: Report Type

Where the Report Type consists of: Buoy #, Wind Avg. and Temp. Avg.
or Buoy#, Last 10 Wind Avg., and Last 10 Temp. Avg.

NOTE:

Even though there are many different types of sensors they all have the same interface. The FWS can request information from a sensor, and the sensor can respond with its value. Notice that the FWS must keep track of which sensors it is requesting information from. This is a boundary decision, the responsibility could have been on the sensor to send its information to the FWS system and indicate what the type of the sensor is.

The FWS transmitter interface is very simple. The transmitter receives different reports from the buoy which it is supposed to transmit to a satellite. According to the interface specification, it is up to the transmitter to know where and how to transmit the information it receives.

Figure 14: Neighboring Domain Interface Specs.

The workflow analysis captures the terminology of the domain as well as organizational information that can be used for gathering more information about the domain. It includes descriptions of how a problem is specified to the application engineers, how they talk about the problem, what documents they receive and use to gather more information, what tools they use to solve the problem, and what high-level approach they take to solving the problem. It also details who they communicate with and how.

The information captured in the workflow analysis is useful for learning about the domain and for designing the language and user environment. The workflow analysis captures the notations and concepts used by the engineers. These are incorporated into the domain-specific language. The workflow analysis also captures what tools are used or needed by application engineers during specification of domain

instances. These can then be incorporated into the user environment.

The workflow analysis additionally provides information on who to contact to get specific information on the domain and neighboring domains. This is helpful for gathering more information, establishing domain boundaries, as well as determining who to approach for providing feedback on the developing language and user environment. The FWS example workflow analysis is captured in Figure 16.

Inputs to the generated components are various sensor values from the sensor hardware.

Outputs of the generated components are various reports that are broadcasted to the radio transmitter hardware.

The various sensor values are collected regularly. Periodically both the average temperature and average wind speed values are sent out in a report. Averages calculation may vary depending on the type and combination of sensors. The FWS also sends out a scheduled history report of the last ten sets of averaged values.

- All sensors values must be collected regularly at a specified interval.
- The averages must be reported regularly at a specified interval.
- The FWS system must be able to store up to 10 wind and temperature averages.

Figure 15: Functional Requirements

As the domain definition activities are carried out, any issues or questions that arise are documented in the issues list. For each issue the alternative solutions, the decision made, and the reasoning behind the decision are also captured so that the same issues do not arise again later.

In summary, the domain definition contains products from each of the tasks performed. This includes the problem statement, architecture and interfaces, requirements, workflow analysis, and issues and their alternative solutions, final decisions and rationales

When building a new FWS the application engineers receive a specification that states what are the number of sensors to be installed and the sensor types, as well as the transmitter type. They also find out how often the sensor should be read and how often the reports should be generated and sent.

With this specification they go out and build the system that correctly implements reading the sensors and generating the reports.

Figure 16: Workflow Analysis

SDA I.II. Formulate formal domain model

The domain model captures the common and varying concepts from a problem view of the domain as opposed to the solution view which is captured in the following step. The domain model captures how the users see and talk about the domain. Users talk about real objects such as floating weather stations, sensors, and reports, as opposed to procedures and controls and character strings. The model provides a framework for describing domain instances based on their distinguishing characteristics. The domain model evolves into the language for expressing problem instances in the domain.

Deriving such a model proceeds in three parts:

1. Identification / abstraction of domain concepts
2. Searching for similar/mathematical counterparts
3. Formalization of the model

Identification and abstraction of domain concepts involves eliciting, describing, and modeling the common and varying concepts of the domain. The entities in the domain are captured as compositions of their parts. The relationships among concepts are also captured. These are integrated into the composition of the entities. Common and varying operations are also modeled.

An initial domain model should be captured early on. It is important that the first model be considered as soon as possible, even if it is naive, because the model helps focus the data gathering and synthesis activities. The initial model can be corrected and refined in later iterations, but without a model, the data gathering activities can degrade into a never ending process of learning every detail about a domain [12].

Abstractions must be actively pursued. Aggregation and generalization are mechanisms for deriving abstractions from concrete data. For example aggregating air temperature sensor and water temperature sensor, into a aggregate sensor of type temperature sensor, raises the level of abstraction. However, aggregation and generalization may not lead to an overall model of the domain. They are considered bottom up abstraction mechanisms.

Top down models must also be sought out during model building, to deduce a high level model of the problem domain that covers the entire domain. For example the FWS can be reduced to a system that reads data, stores some of this data, and transmits reports of the data periodically.

However, neither of these models is sufficient for describing problem instances. Both top down and bottom up modeling are used to derive a model of the domain that defines the minimum amount of information required to specify individual problem instances.

Many levels of the model are produced. They are created in both a top down and bottom up fashion

depending on the concepts being modeled. The high level models are refined and the low level models are abstracted as domain understanding grows, until there is a complete model that covers all the levels. The commonalities and variabilities are captured in these models. Refinements that are common to all domain instances are part of the common structure of the domain while optional or variable refinements depict variabilities.

```
data FWS =
        FWS of (sensor list) * sensor-period * transmitter-type * transmitter-period *
         msg-format;
data msg-format = short-msg | long-msg;
data transmitter-period=
        transmitter-period of int;
data transmitter-type = am | fm;
data sensor-period =
        sensor-period of int;
data sensor =
        sensor of sensor-type * sensor-address;
data sensor-type = AirTemp | WindTemp | WindSpeed;
data sensor-address =
        sensor-address of int;
```

Figure 17: Formal Domain Model

Searching for similar problems and comparing the current domain to these, helps in finding a model for the domain. Abstractions from other domains may be transferable to the current domain. Additional information about these abstractions may then also be useful in understanding the current domain. For example, the solutions or certain properties of the other domain may also be transferable to the domain of study.

Of particular interest are mathematical models that may be similar to the structure of the domain. Examples of these are: graphs, grammars, or finite automata. Such problems have been well studied and that knowledge may be transferable to the domain being studied. For example the mathematical models may provide additional properties that can be rigorously analyzed.

Searching for mathematical counterparts includes determining if there are mathematical abstractions that closely model the domain. If so, then it is seen if their known solutions can be used to help build a solution for the domain. Additional interests include what properties do the mathematical models hold,

and how do these relate to the domain.

When exploring abstractions and formalisms for a domain, there will be much interplay between the natural domain abstractions and the mathematical abstractions. When natural abstractions in the domain are understood try to formalize them by coming up with mathematical counterparts. Similarly, if a mathematical abstraction seems appropriate, test out various aspects of it to see if it leads to a better understanding of the domain in terms of the natural domain entities. The two views work together to specify the domain problem.

Formalizing the domain model involves precisely capturing the relationships among the entities. Once a mathematical model has been selected, SDA advocates building an executable representation of the model in a typed functional language, such as Haskell. It is then illuminating to build instances corresponding to particular examples in the domain in the functional language. This step both tests the adequacy of the model and provides a foundation for subsequent validation and exploratory development at later stages of the method.

Getting the model right is the most difficult and critical step of the process. Executable representations in typed languages prove useful because: (1) naive models often contain type errors, (2) type checking enforces all structural constraints on model entities, providing assurance that the examples really are covered by the model, (3) the process of developing examples tends to illuminate regularities in the domain that may lead to more abstract models, and (4) the executable model representation will be the basis of subsequent prototype development.

The composition of entities is captured as a parameterized datatype in a functional language such as Haskell. The parameters define variabilities to be filled in for specific instances of the domain. Other relations are captured in the structure of the datatype. An example of a formal domain model is captured in Figure 17.

Many models of the domain will be developed during model building. They will vary in formality and abstraction level. Each model being considered must be examined to see how well it applies to the domain. The final model should be: adequate, relevant, consistent, precise, and simple [12].

A precise model is externally correct or unambiguous. A simple model breaks down and structures the domain to ease its understanding. An adequate model conveys all of the information needed. A relevant model contains only the information needed and nothing else. Finally, a consistent model contains no conflicting information.

The output of the domain modeling activities is a formalized model of the domain that has been validated by domain experts and application engineers.

SDA I.III. Define solution model

The solution model is similar to the architecture of the entire system that was captured earlier. However, this model is a refinement of the architecture and captures more details about the particular domain being studied. The solution model captures the design of domain instances. The solution model distinguishes between common and varying aspects of the design of the domain so that the design can be suited to the individual instances.

The solution model partitions the domain into components and connectors that provide the required functionality of problem instances. This model is used in conjunction with the domain model for designing the domain-specific language. This solution structure is needed to understand what is to be generated from the domain model.

The solution model divides the domain into the modules of related functionality that are required for solutions. The interfaces to the modules, sometimes known as their signatures, are captured in the target language of the domain. The target language is the language to be generated during domain instance creation.

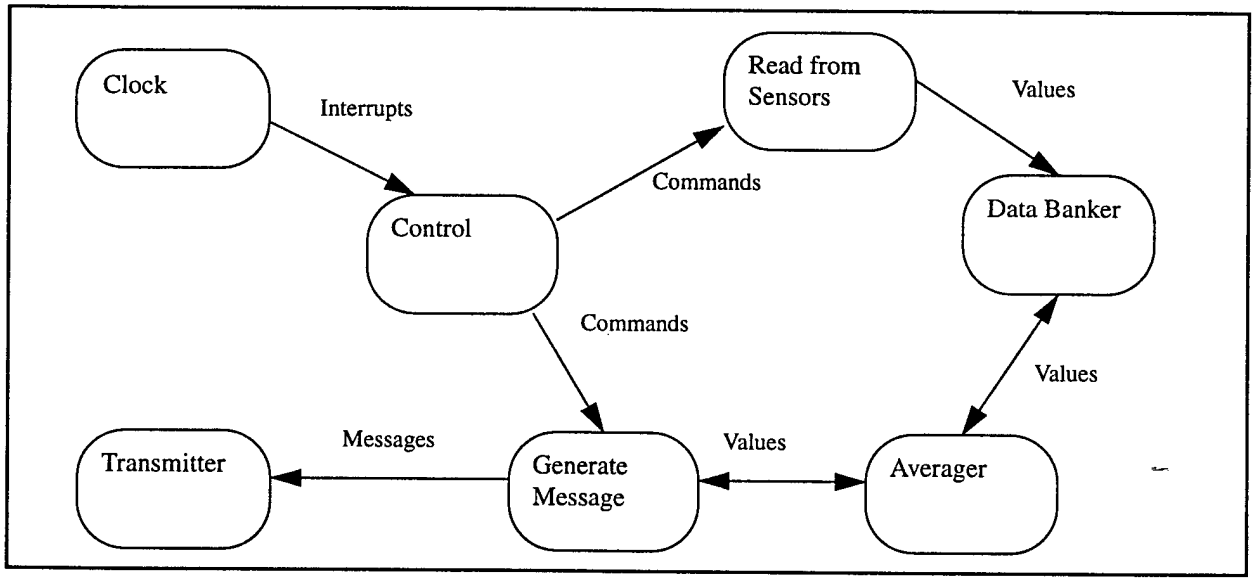

Figure 18: Solution Model

On top of the module signatures, the connections, or interactions between modules are captured. These indicate the flow between modules. Constraints on the solution are also captured, as well as configurations of the modules. These provide additional information on how the modules can be combined, or how the modules interact.

The activities in this step focus on discovering components and connectors that will provide the

needed functionality for domain instances. The mathematical structures used in the domain model may lead to natural partitions for the solution model. Since the abstractions may be well studied, there may be many solution strategies to choose from.

To help understand the solution model, the functionality of a generated component can be captured. To achieve this, the functional requirements of a family member must first be refined. Using the high-level functional requirements from before, refine the requirements for each domain entity. Begin by capturing the expected behaviors of the entities. Capture the inputs and expected outputs of the generated component, as well as, the relation between them. Next separate the domain into sub-components that would be a solution to the problem. Partition the solution into objects and operations that are a part of every family member and also capture the objects and operations that are variable in family members. A partial solution model for the FWS example domain is illustrated in Figure 18.

Since the problem domain is required to be mature and stable, there should also be example solutions that can be analyzed to learn about the solution structure. The solutions can be analyzed to learn about the various components of a solution and how they fit together.

Once known, the elements of the solution model can be separated into functions or modules that specify the partitioning of the domain. Modules capture similar concepts and provide a consistent interface. The interfaces to the functions or modules specify the legal interactions with the elements.

The interfaces of the modules are captured in the target environments programming languages. A description of how the modules interact should also be captured to accompany the interface specifications. It should describe how the elements fit together to solve the problem. As for the system architecture, this may be captured with pictures or text.

The output of the solution modeling activities is a validated architecture of the domain, including the components or modules that will provide the necessary functionality, the connectors that will connect components, and a description of how the components interact to provide the needed system functionality,

SDA I.IV. Capture the interface to the legacy environment

In addition to the architecture of the solution, it is necessary to characterize the environment in which components execute. This somewhat ad hoc collection of requirements is called the environment model. It includes (1) the implementation technology of the target system, (2) significant reusable assets (libraries) provided in the target environment, and (3) non-functional constraints on the behavior of the target system (e.g. performance, stack and heap usage).

In capturing the environment, the primitive datatypes and operations are identified. All library functions from the legacy environment that the generated domain instances may interact with are also

defined. This enables generated applications to be composed of reusable components from the legacy system. This is beneficial if libraries already exist in the domain that implement required functionalities. This way the components don't have to be reimplemented for the generated applications.

Constraints on the generated domain instances are also captured. For example there may be constraints on the language that can be used in the target environment. Other examples of constraints include space and speed restrictions on the generated code.

The requirements of the environment influence the technology that may be used to implement generation. Currently the method assumes that the generated code will be tightly integrated with the legacy code at the source level. This is done by generating the actual domain instances to a language that the legacy code can directly interoperate with. This enables generated components to interact with the legacy code without having to worry about problematic foreign function interfaces [56]. In the initial phase of the SDA project, tool support was developed supporting source level integration of components (this component is the emitter in Figure 10). That is, PacSoft developed a retargetable compiler tool that produces Ada or C that can be linked directly with existing legacy code [56].

In the future, with the advent of COM and CORBA interactions, it may be possible to leave the generated components in a high level functional language, which provide powerful high level control mechanisms and symbolic datatypes that are not available in traditional languages. The connections to the legacy system could be carried out through COM or CORBA connections, which uncouples the tight integration of generated components and the legacy systems. In ongoing work, PacSoft is developing tool support for "object-level linking" using standard interfaces expressed in COM or CORBA.

SDA I.V. Preliminarily Validate Models

At this point it is possible to do a preliminary validation of the models by showing that everything in a solution can be calculated from an instance of the problem described in the model. This activity is done manually as the semantics, which map the problem into the solution, are not yet formally captured.

Specific activities that build confidence in coherence include (1) doing a preliminary design review in which it is argued informally that the solver can be implemented (that is, show that all variable components of the solution are uniquely specified by the domain model), (2) prototyping combinators that may be used to glue aspects of the solution together (ideally these combinators will be well behaved algebraically, frequently they will be expressible as a monad), and (3) reviewing worked out examples with domain experts. If additional information is needed to complete a solution then the domain model is not robust enough and does not contain all of the information necessary.

The outputs of this activity are examples of how problems map to solutions.

SDA II. Design the language

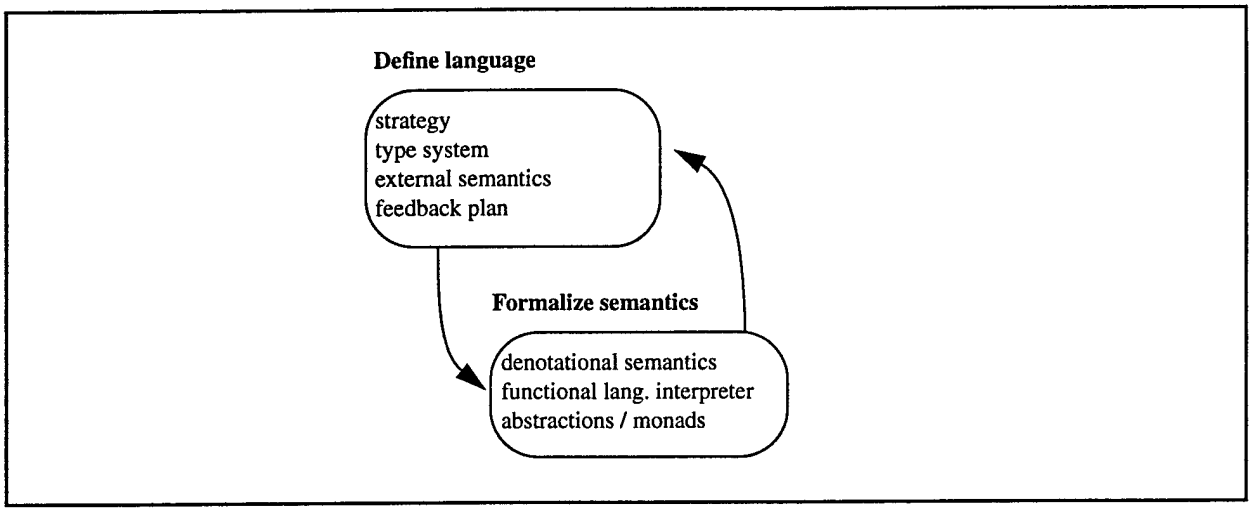

Figure 19: SDA: Design the Language Phase

The language definition phase produces a preliminary DSL implementation as an embedded language with simple domain semantics and solver expressed as interpreters in a functional language so that it can be used and evaluated. This phase is divided into two parts, which are shown in Figure 19:

1. Language definition
2. Semantics formalization.

SDA II.I. Define Language

The language definition captures the informal semantics and syntactic structure of the DSL. To ensure that the DSL meets the needs of its intended users, the domain's application engineers, a feedback plan is produced prior to the language definition, that outlines how to get continued structured feedback on the language so that it can be validated to determine if it is appropriate.

The feedback plan contains information on who to contact for feedback, what feedback to collect, when to collect feedback and what to do with it. Its purpose is to find out if a DSL is intuitive to application engineers since they will be the ones using the new language. Usability is a key concern so the language should be easily understood by the application engineers who will have to use the DSL. The users must approve of the notations used as well as the concepts. In subsequent iterations of the method the application engineers are also consulted to refine the language definition into the concrete syntax of the domain. It is also important to validate that all parameters of a solution are expressible in the language. The workflow analysis provides important information on who to include in the feedback process.

The DSL will be used to specify domain instances. The commonalities of the domain are reused for all instances so they don't need to be expressible. Only the distinguishing features of domain instances need to be included in the language. Generating instances from specifications unites the common and varying features into a complete solution.

The design proceeds in three parts:

1. design strategy
2. type system / gross syntactic structure
3. validate

The design strategy is an informal statement that identifies the key concepts, metaphors, and models to be pursued in the language definition. It identifies the concepts and notations from the models and workflow products that will organize the type system and motivate the gross syntactic structure of the language.

The design principles are refined into the DSL's concrete syntax as the language definition evolves. The workflow analysis and domain model suggest an initial outline for the design principles of the language. The language must be expressive over the varying domain entities and the parameters of the domain model, and it should be captured with the vocabulary and notations of the domain, which are captured in the workflow analysis. Using the same notations and concepts that the application engineers use will make the DSL much easier for them to learn.

Use the workflow analysis and the user view domain model to suggest a strategy for the language. The language must be expressive over the domain entities and the parameters of a solution. This will enable application engineers to specify new members of the domain by expressing only the parameters that make up the new member. However, the language design must take care to only allow the necessary capabilities. There should not be room for the solutions to be over specified by allowing the application engineer to provide implementation details.

The strategy should be discussed with the domain experts in detail. If they think things are missing that weren't captured in the model be sure to update the model as well as the strategy. Make sure the domain experts approve of the notations used as well as the concepts. Usability is a key concern so the strategy should be easily understood by the application engineers who will have to use the DSL.

The type system and gross syntactic structure of the language identify the atomic entities, or types, and entity composition operations appropriate for each type so that all composite types can also be created. At a minimum there must be methods in the language to construct entities of all meaningful types. Typically there will also be constructions that analyze entities of each type as well.

Once the domain information becomes stable, establish the syntax of the DSL. The DSL's syntax

should be based on the vocabulary and notations already in use by the application engineers. Using the same notations and concepts that the application engineers use will make the DSL much easier for them to learn. The language definition can be captured as a grammar for the language. The grammar will specify what the legal combinations of the syntax are.

While defining the syntax, also capture the problem semantics of the language. The problem semantics can be a text description of what the syntax constructs mean. The semantics will be formalized later.

Work with the application engineers, who will be users of the DSL, to refine the language definition. Construct and discuss examples of problems specified in the language so that the users can really see how the language will work. Validate the language definition and environment according to the user feedback plan. Keep the domain models consistent with the language definition so that they don't get stale and out of date.

The outputs of these activities are a complete language syntax and the type system that corresponds to the language.

SDA II.II. Formalize the semantics

The purpose of this step is to capture the precise meaning of the language generator through denotation style semantics. The classic denotational definition of a language is a set of equations that give a compositional definition with an inductive structure derived from the abstract syntax of the language [41, 42]. Semantics-based interpreters take on this structure, occasionally relaxing the strict requirement on compositional definitions and inheriting recursion from the host language. This style of definition is an excellent match for the style of programming supported by modern functional languages.

The semantics formally specify how a new family member is produced from the inputs. It is here that the parameters of family members are defined and mapped into the solution model.

A formal semantics makes explicit the relationship between the concepts introduced in the type system and syntax of the language and the abstractions identified in the domain model and solution model. To decompose the problem of giving language meaning, SDA recommends that the simple domain semantics and the solver be considered separately, and that the solver be further decomposed into separate solvers for each identified facet of the solution.

The first definition of the DSL to be formalized is the simple domain semantics. This is a (sometimes trivial) interpretation of the DSL into specific instances of the domain model.

The next step is to specify the solver, which maps problem specifications to solutions. This is done by giving each of the syntactic terms of the language semantic descriptions. The syntax referred to here is

the abstract syntax of the language. This is done by defining the language compositionaly in terms of phrases, each of which has an associated meaning function. The meaning of an expression is then obtained by composing the semantic functions determined by its syntactic structure. Recall that at an earlier stage the feasibility of this translation was asserted informally; this relationship is made explicit now by completing the explicit formal definition of the solver.

The structure of the solution model will be reflected in the structure of the solver. If the solution model can be easily decomposed, that decomposition can be exploited. If algebraic combinators for building solution components were developed earlier, they can be exploited in the compositional construction of solutions.

SDA suggests building solvers that are structured as semantics-based interpreters of the domain model. Other technologies may be supported in the future. Functional languages are used to implement the interpreters, as they provide the features needed to capture a language design. Functional languages can be viewed as a DSL for language design.

In a semantics-based approach, underlying abstractions can be used to structure the solution. Many critical facets of the semantics of traditional languages are expressed using a simple categorical concept called a monad (or triple) [27]. This mechanism allows semantic concepts to be expressed and combined in a more modular fashion than has been previously possible. It also provides structure and guidance to the design space of language features [60,27]. In particular, it is possible to articulate a short list of semantic features and characterize their potential meaningful interactions. This added structure simplifies the problem of language design.

Many effects need to be explicitly "plumed" throughout the semantics. These features can be abstracted into monads so that only the constructs affected by the feature need to use the monad. If the feature changes then only the monad needs to be changed. The rest of the semantics can be left alone. There are already many well known abstractions that are captured as monads. For example there are the state, exception, and I/O monads.

SDA supports this style of principled language definition in two ways: a methodology that builds on the rich tradition of semantics and the recent results in monadic abstraction, and a tool kit that allows implementations to achieve the reuse promised by the method.

For practical reasons the semantics of a new language are built up from the semantics of other similar languages. The language designer will most likely be familiar with writing semantics and can use their prior experience to come up with a core semantics easily. There are also many books that have examples of semantics that the language designer can use.

SDA II.III. Validate Language Design

As the language is developed, it is validated internally by the team as well as externally by the intended users. Validation ensures that the language definition is correct and complete. Feedback from the intended users is key for usability aspects of the language. This includes the ease of transitioning the language into practice once it is finished.

If formal validation is required, it is possible to characterize and prove properties about the simple domain semantics and the solver that address critical correctness issues. For example, Walton has proven properties about the MSL generator that characterize the correct interaction of parsing and generation solution modules [cite Lisa's new paper]. Similarly, Peyton Jones, Meijer, and Leijen have been able to prove associativity of a "parallel composition" operation for a domain-specific language for animation behaviors in COM [35].

SDA III. Implement the Generator and Support Products

Implementing the generator is the last phase of the SDA method. At this point the language has been defined and a prototype implementation has been developed as an embedded language in a functional host language. It is now necessary to build a tool that can be used by the application engineers.

The first step is to critique the prototype. If it is adequate there is no reason to proceed with further refinement of the implementation. Typically, the prototype may fail to be adequate because: (1) the integration requirements in the environment model require the two-stage form of a compiler or code generator, (2) the interpreter does not meet the constraints of the run-time environment model, or (3) the user interface of the interpreter may not be suitable for the intended user community. These issues are discussed below.

Staging

An interpreter is a program that contains an *eval* function that maps a program into a *behavior* function expressed in the same execution environment as *eval*. In contrast, a compiler (or a generator) is a program that contains a *compile* function that maps a program into an executable object that when subsequently executed behaves according to the specification of the program. This difference between interpreters and compilers is characterized by their execution stages. The interpreter has a single stage, containing both *eval* and the *behavior*. The compiler or generator has two stages: "compile-time" when the *compile* function is executed and "run-time" when the specified behavior occurs.

The methods for prototyping the solver as an interpreter produce a single-stage component. Single-stage systems may be appropriate for some environments, particularly ones in which performance is not critical and a high-level interface (such as COM or CORBA) can be exploited for connection with

other system assets.

In performance critical systems that require code-level integration in legacy languages (such as C or Ada) a translator is typically required. Sheard, Taha, and Benaissa have developed extensive tools and techniques based on typed-metaprogramming for the systematic conversion of single-stage interpreters into multi-stage translators [53]. These tools are part of the SDA technology.

Environment Constraints

In many applications, the generated solutions must interoperate closely with legacy code expressed in a conventional imperative language, such as C or Ada. In this case a systematic translation of the behavior from the specifications in the functional prototype to the legacy language is required. (This is called the emitter in Figure 10.) PacSoft has developed extensive compilation infrastructure to support this. Building on ideas of Kieburtz and Volpano [59], Oliva and Tolmach developed a highly parameterized functional language compiler called RML (Restricted ML) [56]. RML compiles a simple functional language to C or Ada in a type faithful way, and has extensive mechanisms for encapsulating legacy assets (such as libraries) so that they may be used efficiently by the generated code.

To use RML to implement an emitter, the solver is first staged (either manually as was done with MSL or automatically with the tools described above), then the functional program expressing the behavior of the generated component is compiled to the legacy language by the RML compiler.

Appropriate User Interface and Analyses

The third typical shortcoming of the prototype language environment is the appropriateness of the tool for use by the intended domain experts. Specific issues here can include (1) unintuitive interaction between the host and embedded language (e.g. complex type errors that expose implementation details), (2) opportunities for a more refined (or possibly more liberal) type discipline than that inherited from the host language, (3) the need to produce high-quality error messages, and (4) the opportunity to have a non-textual representation of the language.

Currently all of these "front-end" and analysis issues are addressed in SDA by using conventional compiler implementation technologies and methods. For example, the lexical analysis and parsing in the MSL implementation was developed with ML-lex and ML-yacc. The type inference algorithm was coded by hand in SML.

Another issue to be considered when critiquing the prototype is if the DSL provides opportunities for tools other than the generator that might be useful to the users. These tools might include analysis tools or test-case generators. Often these may be defined in the context of the semantics-based interpreter as a "non-standard semantics" of the model.

Once the critique is complete a design and implementation plan for the language environment

must be developed and executed.

This phase is broken down into two parts, which are shown in Figure 20:

- Design the generator and support products
- Develop the generator and support products

In the first part, the user environment and all the support products are designed. Also the plan for implementing the generator is defined. In the second part, all the products are developed based on their designs and plans. Documentation on the language and the environment are also produced.

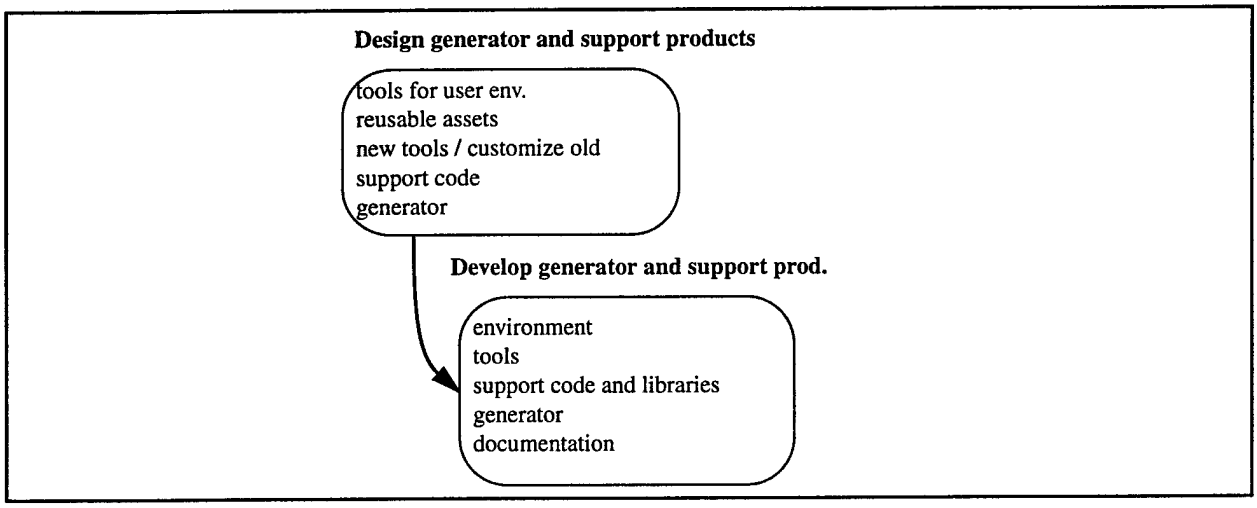

Figure 20: SDA: Implement Phase

SDA III.I. Design the Generator and Support Products

The goal of this step is to capture a complete design of the generator and the additional products that will support it. These include a user environment, analysis tools and any support products needed by the generated code.

The generator is already implemented as an interpreter. To transition the implementation of the DSL from the interpreter to a compiler, each construct must be redefined in terms of the code it computes as opposed to the value it computes. Sometimes the design choices of the interpreter may need to be revisited for parts of the compiler. Metaprogramming research is focusing on how to make this transition from interpreter to compiler faster and easier [53].

Along with the generator, various generator support products are also designed. The support products include the user environment, analysis tools, and support products for generated code.

The user environment design is based on what the application engineers want and need out of the system. Usability is a key concern. If the engineers don't like the package or if it doesn't meet their needs, they won't adopt it. The user environment includes the users interface to the language. The language can

be graphical or textual or both depending on the users needs and preferences.

The user environment must be able to transform the users input into the abstract syntax of the DSL. For example a lexer and parser may need to be built to take text from the DSL and return an equivalent expression in the abstract syntax. The lexer and parser can be generated with traditional compiler tools like ML Lex and Yacc. They are tools that generate lexers and parsers given the grammar of a language.

The user environment also includes the links to all the tools that are available to the user. While designing the environment, the tools that the users will benefit from should be determined. The tools needed will depend heavily on the domain. Example tools include analysis tools and specification writing support tools. Some tools may already be implemented somewhere else. It may be possible to link these tools into the environment to reduce the amount of work to do. The other tools that do not have implementations will need to be designed.

During the design step it is also necessary to prepare designs for any other support products or run time libraries that generated code might need. The specifications for run time libraries should already have been completed in the solution modeling step. Routines that the generated component call should already by specified because the interpreter has to know how to access them. They do not have to be designed or implemented yet though. If the libraries and support tools have not been specified already then designs must be produced for them.

To accomplish this it must be decided what things can be captured as libraries and which must be generated. The common aspects of related entities should be identified. Once these are known, it can be determined which entities can be captured in static code libraries and which are more appropriately modeled with high level algorithmic descriptions and parameterized functional constraints. These classifications depend on how the entities will be used. They also depend on how the entities will vary across family members.

These classifications can be used to design the run-time libraries that a generated component can access. The designs should include a specification for all the libraries. It should also specify: what the offered routines are, how they can be accessed, and who has access to them.

The outputs of these activities include plans and designs for implementing the translator, the user environment, analysis tools, and any other support products.

SDA III.II. Develop the Generator and Support Products

The plans for each part of the final product should be written with incremental development steps that can be inspected, tested, and shown to the application engineers for approval. Because the entire SDA method is iterative, the generator will be incrementally developed as the models and designs become more

detailed.

The output of this activity is all the assets that are intended to support the application engineers in specifying domain instances.

4. Integration of FAST and SDA

4.1. Introduction

In the previous chapter the current states of the FAST and SDA methods are described. Now that the methods are defined, the integration of the methods can be explored.

The integration proceeds in three steps. First a framework for comparing the methods is defined. Then the methods are evaluated and compared using this framework. Finally a proposal for the integration of the methods is suggested based on the comparison.

4.2. Framework for Comparison

It has already been proposed that the FAST and SDA methods are complementary [65]. That is, that integrating the methods would combine the strengths of each method while allaying some of the weaknesses. However, the points of integration have to be more deeply explored and rationalized.

In order to understand where the strengths of one method counter the weaknesses of the other, the methods have to be compared. This is done by comparing the similar process steps and work products of the methods.

For the evaluation both methods are divided into 9 high level process steps and a category for general process characteristics. The ten resulting categories are:

1. General
2. Domain selection
3. Information gathering
4. Domain scoping
5. Domain modeling
6. Language design
7. Generator building
8. Application Engineering Environment creation
9. Application Engineering Process capture
10. Documentation

For each category both methods are analyzed in terms of their strengths and weaknesses in that category. Table 3 summarizes the strengths and weaknesses of the two methods side by side based on this framework.

This framework was chosen because it highlights where the strengths of one process may mitigate

the weaknesses of the other. It also highlights where there are uncontested weaknesses to addressed. These capabilities support the goal of deriving an integrated process.

In the remainder of this chapter the methods are compared based on this framework and a proposal for integration is suggested. Each of the categories is discussed and analyzed to determine how the integration might proceed.

Table 3: Summary of Strengths and Weaknesses

	FAST	SDA
General	- Iterate only after complete cycle	+ Highly iterative process
	- Transitioning assets	- Transitioning assets
Domain Selection	+ Qualify Domain	- Select domain
Information Gathering	+ Commonality Analysis as method to gather information + Mature process + Reflect / explore / share / capture knowledge + Validation	- Information gathering + Workflow analysis + Validation
Domain Scoping	- Overview — basic domain concepts	+ Problem statement
		+ Global architectures and other models
		+ High level requirements
Domain Modeling	+ Commonality Analysis + Mature / accessible + Commonalities + Variabilities + Parameters of Variation + Dictionary of Terms + Issues - Informal	+ Domain Model + Problem view + Abstraction / mathematical model + Formality
		+ Solution Model + Solution view
	+ Decision model	+ Interface to the environment

Table 3 (continued): Summary of Strengths and Weaknesses

	FAST	SDA
Language Design	- Lack of support for language design	+ Denotational semantics
		+ Type systems
		+ Monads
		+ Functional languages
		- Accessibility
Generator	- Lack of generator support	+ Precise language definition
		+ Transformation tools
		+/- Other tools
		+ Incremental development
		+ Prototyping explicit
AEE	- Application Engineering Environment	- Application Engineering Environment
AEP	+ Application Engineering Process	- Application Engineering Process
Doc.	- Documentation	- Documentation

4.3. Evaluation and Comparison

In this section the FAST and SDA methods are compared within the ten categories outlined above. Each category includes a discussion of the current strengths and weaknesses of each method as well as a discussion of an integrated step. Obviously strengths that are common to both methods are left in the integrated method and will not be discussed in detail. Weaknesses that are shared by both methods or that occur in one method but do not have a counter step in the other are left for future work to improve. These are discussed in the conclusions chapter. Of particular interest here is the integration of the strengths of one method to balance the weaknesses of the other.

4.3.1. General

The general category covers concepts of the processes that span the entire method as opposed to concepts that only apply to parts of the processes. In the general category one general characteristic analyzed is the process style of the processes. Along with this the transitioning of assets is addressed as this

applies to all products created through a domain engineering project.

- **FAST**
  - **Iterative Process:** The FAST process iterates only after going through an entire cycle. It was observed during the collaboration project that the domain engineer's understanding of the domain increased and changed as they proceeded through the design and implementation of an AML. However, the domain engineers did not revisit the CA to reevaluate their model of the domain. At the end of the cycle the CA was inconsistent with the current understanding of the domain and the engineers had lost the high level information as it was buried in the implementation instead of updated in the CA. The new products can not be compared with the CA until it is updated.
  - **Transitioning Assets:** FAST has many advantages over other methods for transitioning technology to application engineers due to the fact that the domain engineers come from the application engineering organization. Some social issues involved in transitioning assets are avoided because the application engineers already know the domain experts and trust them over outsiders.

    While FAST has advantages over other methods for transitioning technology to application engineers, there are also some issues that are not addressed. The method does not explicitly address transitioning assets. There have been problems at Lucent in the past of transitioning assets because of a lack of buy in from application engineers and managers.
- **SDA**
  + **Iterative Process:** SDA is a highly iterative process for developing tool supported domain-specific languages. Non-iterative processes such as waterfall type methods require that a step be completed prior to moving on to another stage of the product life cycle. However, typically a domain is not understood completely in the beginning of a domain engineering project. Iterative processes enable process steps and workproducts to be left incomplete while deeper domain knowledge is sought. The process and workproducts are then revisited once more information is known about the domain [37].

    Iterative processes also provide a means for keeping the earlier stage's workproducts consistent with later stage's workproducts by revisiting them throughout the life cycle. If the earlier workproducts aren't kept consistent then they become worthless for the maintenance of the assets.
  - **Transitioning Assets:** SDA does not explicitly address social issues of transitioning assets to the application engineers. However, the workflow analysis requires that application engineers are selected for validation and feedback on the language. The language is designed with the user in mind so that it is easily learned by application engineers.

- **Integration Proposal**

The process style of SDA explicitly supports the revisitation and update of products throughout the analysis and design. This ensures that all products are kept consistent. It also enables the domain engineering team to proceed with steps even when a previous step is not considered complete. This way progress can continue even when something is not fully understood, and then the products can be revisited when the requisite knowledge is available. For these reasons, the integrated process will adopt the iterative process style of SDA.

As for the transitioning of assets once they are created, it is left for future work to come up with explicit activities which address this issue. Both methods provide little support for this, and it is really a management activity that has been left out of the scope of this thesis.

4.3.2. Domain Selection

The domain selection process category includes activities that determine if a target domain is a good choice for domain engineering. The assumptions section prior to the method definitions in the previous chapter captured some of the criteria for a domain to be considered for domain engineering. These include:

- Domain expertise exists
- The domain is mature and stable

- **FAST**
  - + **Qualify Domain**: On top of striving for the general criteria mentioned above, the FAST process includes a step for studying the economic tradeoffs involved in applying the FAST process to a domain. This assumes that the major goal of engineering a domain is economic. It is beneficial to do this because it mitigates the risk of putting a lot of effort into a domain engineering activity that will not pay off. It also provides estimates which can be used for planning and scheduling the domain engineering activities.
- **SDA**
  - - **Select Domain:** SDA, on the other hand, assumes that a domain has already been selected and does not contain further guidance on the selection of domains which are applicable for SDA.
- **Integration Proposal**

The economic model of FAST provides some assurance that a domain engineering effort on the target domain is worthwhile. This step can be integrated with any domain analysis process as it is quite general to domain engineering. Economic analysis for reuse is still a field of study and further findings may be integrated into the methods in the future. Also in the future, other goals besides purely economic

ones may be considered during domain selection. Examples of other goals could be improved quality and reliability within the domain. The ODM method provides activities for setting goals for a domain engineering effort and then selecting domains that may fulfill these goals[46].

4.3.3. Information Gathering

The information gathering process category includes activities related to the elicitation of information for the domain analysis.

- **FAST**
  - **Commonality Analysis:** FAST's commonality analysis process is the main information gathering mechanism for FAST. The FAST method is based on the Synthesis domain engineering method developed at the SPC. Many of its key concepts have been used in practice repeatedly. The commonality analysis process has been applied to over a dozen domains within Lucent. This part of the FAST method has been tested and validated many times.

    The information gathering is done directly by domain experts who already know about the domain and where to find additional information. A well defined yet flexible process is provided that guides the team of experts through gathering and analyzing the domain information. The process follows the information that is being gathered. This means that the process is flexible concerning what information to gather. As well as flexible concerning the organization of the information so that the necessary information can be captured in a form that is natural for the domain. The process also supports alternate forms of information capture, so that the analysis can be tailored to the skills of the team. However, alternate representations are not described.

    During the process the experts reflect on what they know and share this with the other team members. This information is captured in the commonality analysis document for others to learn about as well. This way the information is not lost if the experts leave. During the process the team also explores future possibilities in the domain. This is likely to increase the applicability of the domain analysis in the future.

    The commonality analysis process itself is highly iterative. The process repeats until the domain analysis team considers the document complete. After that the document is reviewed by the team.

    At the end of the analysis the products are reviewed by other domain experts and additional people who have knowledge about the domain. For example experts of neighboring domains. This helps to ensure that the analysis is correct and complete. However, as the commonality analysis is captured as plain text it does not give any stringent completeness or consistency

criteria. As they can get quite large, it becomes increasingly hard to check the entire document for any criteria.

We have observed at Lucent that many of the commonality analyses performed are considered successful in that domain abstractions are found and greater domain understanding is achieved. However they do not all lead to languages as the FAST method details. Another observation is that the process is accessible to the engineers and it is well liked by them.

- **SDA**
  - **Information Gathering:** The SDA method does not provide much support for gathering information. Previously it was assumed that an adequate domain analysis would be provided by the team of domain experts working on the domain engineering project. The SDA method assumed that the DA document along with extra information from domain experts would be enough to build the artifacts needed for the SDA method.

    The workflow analysis product of the SDA method helps identify some of the key information needed about a domain for defining a domain-specific language. It uncovers how the problem is currently being solved: what process is followed, what tools are used, what artifacts are used and produced, and what people are involved.

    The analysis helps uncover some of the concepts and notations of the domain. These will be used in the domain model and language definition as they help define the user view of the domain. However, the process for gathering these facts is not more than looking at the documents and question the domain experts.

    The analysis also provides insight into key people, or groups that may be necessary to interact with during the project. It may be necessary to contact people from other departments or domains to establish appropriate domain boundaries. Other groups of people are identified for providing feedback on the products developed, i.e, the intended users of the products. One additional piece of information that comes out of the analysis is an understanding of what tools the engineers already have or may desire in the future.

- **Integration Proposal**

The FAST method balances some of the information gathering weaknesses of the SDA method by providing the commonality analysis process for domain experts to follow as they extract relevant information about the domain. However, the commonality analysis process has its own weaknesses such as lack of abstraction support and lack of formal completeness or consistency checks. These are discussed in more detail under the domain modeling subsection.

4.3.4. Domain Scoping

The domain scoping process category contains activities for specifying more precisely which domain is being engineered.

- **FAST**
  - **Overview:** The FAST commonality analysis includes an overview section that captures a high level description of the domain. In this section, FAST advocates scoping the domain. However, there is little guidance on how to do this or how to represent the scoping information. Without proper boundaries and interfaces captured for a domain, the focus of the commonality analysis may be wrong.

    We observed during one domain engineering project that the boundaries between two domains were assumed instead of defined and validated by experts in the other domain. The domain experts for the domain in question assumed that certain functionality was provided by the other domain, that wasn't. Explicitly defining the boundaries and interfaces would have captured this mismatch earlier in the domain engineering process.
- **SDA**
  - **Domain Description:** The SDA domain description captures initial domain understanding. This includes a high level problem description, a global architecture for the domain and an initial set of functional requirements. The problem description and high level requirements are similar to the overview of FAST.

    The global software architecture is really the main scoping product of SDA. A global architecture looks at the overall system organization and system level properties. It focuses on the entire system not just the domain of study that is a part of the system. Discovering an architecture for the whole system helps the understanding of how the domain fits into that system. A view of the overall system also enables decisions be made at the system level instead of only at the domain level. This is important because there are often issues that cannot be encapsulated in only one place [45].

    The global software architecture provides the context of use for the domain of study. It captures at a high level the inputs, outputs, and required functionality of components in the domain.

    The global software architecture also determines the neighboring domains of the domain under study. The neighboring domains interact with the generated components. Studying the overall system defines the domain boundaries. Establishing these boundaries is called scoping the

domain. The boundaries define the interfaces to neighboring domains precisely so that no ambiguities arise concerning which domains are responsible for what.

- **Integration Proposal**

The precise scoping of the domain is important for further domain analysis activities. Therefore, the integrated method will include the additional scoping activities of SDA as well as the high level problem description from either method.

4.3.5. Domain Modeling

The domain modeling process category includes the major domain analysis activities. The information gathering process defines how to collect this information. Therefore, these activities are described in terms of the information that is captured during domain modeling and the representation of that information.

- **FAST**
  - **Commonality Analysis:** The commonality analysis document is the key output of a domain analysis in FAST. In it the commonalities of a domain are captured. These are the assumptions that can be reused for all domain instances. It also contains the variabilities of a domain. The variabilities define how the domain instances may differ. These are further qualified with the parameters of variation, which capture the range the variabilities span over and the binding times of the variabilities. The variabilities can be directly used to define the decision model, which is the basis for the application modeling language constructs.

    A domain dictionary is also captured during the commonality analysis process. The dictionary ensures that standard terminology is defined and agreed on by experts. The dictionary can be used as a reference by all people involved with the domain.

    A list of issues and their decisions are captured during the analysis as well. These provide important information about the history of the domain and the rationale for why it is the way it is. The issues are valuable during future iterations of the FAST process, as they capture decisions which can be reexplored to see if the assumptions still hold.

    Apart from providing the basis for a language, the commonality analysis also contains much value in and of itself. It can be used by application engineers during development. The commonality analysis can also be used as an aid in training new application engineers in a domain.

    The document is to be structured according to the needs of the domain. The structure chosen provides a higher level view of the domain information. The commonality analysis provides some guidance on how to formulate abstractions of the domain. The approach taken is bottom up.

However, apart from this, the commonality analysis does not have explicit means to raise the level of abstraction in a domain. Abstractions may be found when trying to structure the material, but they are not actively sought out, especially not mathematical abstractions, which would be precise and partially solved.

The commonality analysis is captured as text. This representation can be tailored to the skills of the domain engineering team. However, this is not explicitly supported by the FAST process. As text, the commonality analysis is difficult to analyze for consistency and completeness. The lack of formality is considered a weakness of the FAST method. Even though the method is designed to be accessible and therefore purposefully informal. The benefits of formality are considered in this thesis to outweigh the drawbacks.

Without proper rigor there could be incomplete, inconsistent, or ambiguous information captured about the domain. There are some informal checks and reviews that help find some problems, but there are no systematic mechanisms defined to find out if there are any problems.

The analysis of commonality analyses done previously by Walton and Hook captures three additional criticisms of the commonality analysis [65]. The first is that the level of information captured during an analysis varies. The second is that all information is captured in one document. And the third is that the current state of domain is not distinguished from the future desired state.

+ **Decision Model**: The FAST process also produces a decision model. The decision model sequences the decisions that need to be made while specifying family instances it is derived from the variabilities and the parameters of variation. The decision model is used to define the language and the standard application engineering process.

- **SDA**

The SDA method captures three distinct models of the domain: the domain model, the solution model and the environment interface specification. The three views differentiate between concepts as expressed to and by the user or the problem level information, concepts used for expressing a solution, and concepts only required for implementation. Distinguishing between the three views is a strength of the SDA method, because it separates the knowledge as needed for a good language design.

In order to benefit from the three views there must be a clear mapping between them. Otherwise the problems of not having separate views are reintroduced. There may be misunderstandings between the users and the designers. Ambiguities in the user view which translate into incorrect specifications for the solution view are another potential problem. A precise mapping can be captured with an interpreter if the models are captured in a programming language.

+ **Domain Model:** The user view presents information to the users in terms and notations that they

are familiar with and that have specific meanings in the domain. The domain model is the basis for the domain-specific language. The language is used by the user to express specifications of problems in the domain, therefore it is beneficial to have the cognitive distance from the language notations to the notations already known and used by the users to be as small as possible.

The domain model captures the structure of the domain at the problem level. It also precisely defines the variabilities that domain instances can range over, again at the problem level.

The domain model is captured formally with a functional language. Therefore the domain model is precise and unambiguous. It can be checked for certain properties such as completeness or consistency more easily than models captured as plain text.

There are also some drawbacks associated with using formal methods. These are typically related to the problems of introducing formal specifications into current software processes [37]. The engineers have to be trained to use the formal notation if they are to own and update the products. However, they should be able to validate the models even without knowing how to create them themselves if a modeling expert works through the models with them.

Emphasizing the need for abstraction of domain concepts is another strength of the SDA method. Both the problem and solution view of the domain in SDA should be abstractions of the domain. Raising the level of abstraction in a domain provides for benefits in the areas of understandability and reuse. An abstraction of the domain provides a view of the entire domain that can be understood at a high level. The details are left out and considered in another lower level view.

Another benefit of abstraction is the possibility for more reuse within a domain. With abstraction many things that would be considered different are thought of the same way when their details are left out. Therefore there is the possibility to reuse general artifacts that work for one of the elements on the other similar elements.

SDA is special in that it prefers mathematically rigorous models of the domain. Mathematical models may have certain properties that can be used to reason about properties of the domain. Another benefit of mathematical abstractions is that these structures, for example graphs and finite state machines, already have well known solutions. Therefore the new domain can benefit from the work that has already been done on these abstractions.

Unfortunately desiring abstractions of the domain and developing them are entirely different issues. Currently there are not well established mechanisms for determining appropriate abstractions. This issue is discussed in more detail in the SDA general weaknesses section under the heading of abstraction.

Coming up with abstractions for a domain, either domain-specific or mathematical abstrac-

tion, may require a lot of thought and previous experience. Some abstractions are very natural for a domain yet that doesn't make them the most powerful abstractions for the domain. There may be an abstraction that fits the domain better but isn't quite as obvious. Finding these abstractions for a domain requires previous experience with abstractions and other domains.

Known abstractions can be tested on the domain in question, or a previously studied domain may have similarities to the current domain and the previous domain's abstraction can be tested against the current domain. Once the principles of abstractions are understood, new abstractions can be created for a domain.

Once an abstraction is chosen it must be validated to check if it really covers the domain as needed. There are no objective criteria for checking if one abstraction covers a domain better than another. The abstractions must be tested against the domain to see how they fit, and how they can be expanded to cover the important concepts not yet accounted for.

Coming up with domain-specific abstractions is difficult because the application engineers may be aware of the abstractions when they are using the domain engineering products. They will have to approve of them and trust that they will cover all the cases they are supposed to handle. To come up with mathematical abstractions the student of abstraction will of course benefit from having a strong background in mathematics.

Courses are being considered that will help teach abstraction, however, currently it seems the only way to really learn how to come up with models is to witness many being chosen and discarded or validated based on their coverage of the domain.

An appropriate domain model provides much valuable information on the domain and specifically for the language design. However, SDA does not provide mechanisms for discovering the domain model. It is assumed that the information required can be found in the domain analysis or provided by the domain experts.

+ **Solution Model:** The solution view captures the solution structure, or architecture of a generated component. It is a refinement of the entire system's architecture but only for the domain being studied. Defining the architecture of the component breaks up the solution into parts that can be expressed as modules.

The model is still an abstraction of the entire functionality so only the signatures of the modules needs to be specified. The interfaces to the modules specify what functions and types the modules offer. The amount of information studied is bounded and the details are left out so that the problem can be understood at a high level without the details of implementation. Accompanying the model should be an understanding of how the modules interact.

The modular structure can be reused on all the systems. When an iteration on the system is done the abstraction is looked at and changed instead of looking at all of the implementation details. Once it is clear where the abstraction is affected the underlying implementation can be changed where necessary. The unaffected pieces can be left alone.

+ **Interface to Environment:** Currently SDA requires that code be integrated with the legacy environment at the source code level. The interface to the legacy environment is used to capture the interfaces to the legacy code and constraints on generated code. Precisely defining the interfaces allows the generated components to interact with and even reuse code that already exists. However, there is currently no support for capturing non-functional requirements. They are just captured as text.

- **Integration Proposal**

Both methods have strengths and weaknesses concerning domain modeling. The integrated process should try to combine the benefits in a way the diminishes the weaknesses without introducing new ones.

The formal methods of the SDA process can be integrated with the commonality analysis process to create a process for extracting, abstracting, and representing domain information. The integrated process is an iterative application of extracting commonalities and variabilities, and abstracting, formalizing and representing them. As well as a process of taking the high level mathematical models and refining them. This is a bottom up and top down approach. Eventually the two definitions converge on a common model of the domain.

The high level models help structure the information gathering process. If the domain models are built after the commonalities and variabilities are already captured then there may be no opportunity to abstract to a higher level without duplication of effort. The commonalities and variabilities are a good start at abstraction. However they are a bottom up method for coming up with a domain model. They don't emphasize coming up with a general abstraction that can be refined to better suit the domain.

Another benefit of integrating the formality of SDA with the commonality process is that capturing the common and variable assumptions formally provides a framework for specifying correctness and completeness criteria for the models. This was one of the weakness of the FAST commonality analysis.

One of the keys is to have the domain experts and formal method experts working side by side to come up with this model of the domain. This process aids in the accessibility issue of formal methods. This is because the domain experts aren't expected to learn the formal methods. The method experts model the domain during information capture and validate the formalisms with the domain experts. The method experts use the terminology and notations of the domain where ever possible in the models. There may still

be problems in getting the domain experts to approve of the abstractions used. This is also a problem with the FAST method. However bringing the two groups together so that they can learn from each other is a good place to start. In our work with Lucent it has been observed that some abstractions can be validated and approved if given the proper backing.

Additionally the decision model of FAST can also be integrated into the process. The decision model provides the application engineers with guidance during system development. The process for coming up with the decision model will have to be tailored for the integrated method because the parameters of variation will be captured more formally. It will still be possible to come up with a skeletal process by analyzing the domain-specific language and the variabilities allowed as parameters.

4.3.6. Language Design

The language design process category captures activities for deciding on and capturing the domain-specific language specification.

- **FAST**
  - **Language Design**: Although the FAST method calls for a language design, it does not provide a detailed process for defining a language design from the commonality analysis and decision model. The syntax of the language comes from the decisions that must be made. This is captured as a grammar.

    FAST defines two different ways to design the language. One is using defined components and composing them to create a new domain instance. The other way is to compile a low level domain instance from a high level specification. However, processes for defining the semantics if compilation is the approach taken, or deducing a domain design and composition mapping if composition is the approach taken are undefined.
- **SDA**

SDA includes many mechanisms which help structure, analyze, and validate a language design. Among these are: type systems for structuring the syntax, semantics for precisely defining the meaning, monads for structuring and reusing semantics, and functional languages for prototyping the language. Each of these is discussed in more detail below.

+ **Type Systems:** Some people may argue that type systems are overly restrictive and that perfectly good programs cannot be written because they won't type check. Others argue that type annotations are too cumbersome. SDA promotes building typed languages because they help the language user by providing assurance of program correctness at compile time. Only programs that use types consistently can be compiled. These constraints can be checked statically to provide addi-

tional assurance of a programs correctness. The programmer is protected because they can only use types where they are expected and they can only use operations on types that are recognized by that type.

Types also provide an abstraction mechanism for the language designer. Domain-specific types can be introduced in the language and the language user will have no way to know how they are implemented or to access the underlying implementation. They won't need to know how the type is implemented as long as they are given operations that work on the type.

+ **Denotational Semantics:** Providing a formal denotational semantics for a language is very beneficial. First of all a denotational semantics provides an independent specification for the meaning of the language [41, 42]. It doesn't depend on how concepts are actually implemented and it doesn't depend on the machine the language will run on. The same semantics can be used if an array is used to implement a queue or if a linked list is used. The semantics can be reused if the language is to run on a Unix box or a PC. Developing a language from a formal semantics specification ensures that there is no ambiguity in how a specification in the language will behave.

Another benefit of denotational semantics is that they are based on well understood mathematical and logical foundations. This provides that language designer with the capability to reason about properties of the language. For example if two concepts in the language are supposed to be inverses the language designer can prove that this property holds according to the semantics.

The domain model and the solution model provide the two ends of the semantics. However, there is not a description of the information needed to build the semantics. This is a topic that must be explored in the future.

+ **Monads:** Monads are an abstraction mechanism that aid in structuring denotational semantics. They are used to encapsulate certain features, typically features that have effects, that otherwise would be "plumbed" through the entire semantics explicitly. The type of features that can be abstracted with monads follow a similar structure. There are many papers that cover monads in-depth [60, 27].

Features that have effects like state or exceptions can be added to the semantics after the core semantics are complete. They must be passed through the semantics everywhere explicitly stating how they effect every computation. If the feature changes then the entire semantics must be updated to reflect the changes. If a new feature that follows the same structure is added then everywhere the old feature appears needs to be updated to incorporate the new feature.

When a monad is used instead of specifying the features throughout the semantics the monad specifies where these type of features will affect the semantics. If a feature must be

changed then only local changes are made to the monad and the semantics are left unaffected. When new features are added the monad is redefined and the semantic expressions themselves are unaffected [60].

Monads make it easier to analyze the semantics of a language because the monad encapsulates the extra features which make the real computations easier to reason about. Monads also make the semantics more concise because the feature's implementation is moved to one place instead of repeatedly used throughout the semantics. The trade-off is that people must understand monads and how they work.

Monads can also be used when implementing a semantics in a functional language. Some languages like Haskell and ML have mechanisms for aiding in the specification and update of monads.

+ **Functional Languages:** Functional languages can be used as a formal method for capturing domain models, the syntax of a language, and the semantics of a language. Functional languages are similar to denotational semantics, as they are based on the same underlying principles. Functional languages are declarative, so an expression is given a meaning in terms of what it produces instead of how the end result is produced. Because of their similarities to denotational semantics, the mapping between a denotational semantics for a language and an interpreter in a functional language is almost trivial. A language prototype can be made quickly so that users can test the language and see how it works.
- **Language Design:** Semantics based language design, functional programming, and semantics structuring using monads are all techniques that take years to master. Currently there isn't even comprehensive training material that explains the underlying theory or teaches the techniques. Even if there were courses designed to teach these concepts it is not clear that it would be worthwhile turning the domain experts into formal method experts. The time involved may be extensive and it is possible that the concepts may still not be grasped by all the people who attend. Because of this the SDA method advocates having domain engineering teams that consist of both domain experts and language designers trained in formal methods.

  Training materials are still being developed because it may be worthwhile for an organization to invest in educating some of its own engineers in formal methods, but it is not necessary to train all of them. The trained individuals can then work with many domain engineering teams at the organization and take the place of the PacSoft formal method experts.

• **Integration Proposal**

SDA's key strengths are its support for language design and development. These strengths balance

out FAST's lack of support for language support. By using mathematical models, denotational semantics, monads, type systems and functional languages, SDA provides a framework for creating languages that have "nice" properties. Also the formality provides the possibility for some properties of the language and the domain to be analyzed more rigorously than the FAST method provides.

As for the problem with education, this is discussed in the conclusions as future work.

4.3.7. Generator Building

The generator building process category includes activities that support the implementation of the language.

- **FAST**
  - **Generator:** FAST does not provide guidance on how to build a generator from the language design.
- **SDA**

SDA provides strong support for implementation of the generator. The specification of the language is captured precisely and tools are provided for transforming high level functional languages into more common imperative languages. Additional tool support is provided by general purpose language development tools. Additionally incremental development is adopted as the language implementation strategy. These are discussed below.

+ **Transformation Tools:** The transformation tools currently available include the PacSoft "pipeline" that takes a high-level functional specification of a language semantics and transforms it into another language by performing a series of automated program transformations that are applied during the course of program generation [6,56]. Not only do these transformation take the high level functional specification and turn it into a first order specification, they also optimize the code with performance improving transformations.

  The benefits of the transformation tools are that the domain-specific language implementations can be written in functional languages which offer high level implementations but have been long deemed too slow for real applications. The transformation tools transform the high-level domain-specific programs into low level imperative programs that are comparable to programs written directly in imperative languages. This performance improvement from the transformation tools has not yet been proven to be equivalent to directly writing in an imperative language, however studies are underway [cite PRISM performance].
+ **Other Tools:** Tools are available that help with language implementation. Examples of such tools are the programs Lex and Yacc that generate a lexer and parser for a language given a grammar

and some other information about the language. These tools automate parts of language development so that the designer can concentrate on other things. Further support is desired.

+ **Incremental Development:** Incremental development is a well known mechanism for developing assets incrementally in small coherent chunks.
- **Tool support:** There are not many tools available to support language implementation. There is also not a clear understanding of what tools might be useful.
+ **Prototyping:** Prototyping is a well known mechanism for developing partial-products quickly that can be validated early on by customers to ensure that the correct product will be built [Pressman]. Prototypes typically do not implement total system functionality, but they provide a means for capturing and agreeing on the requirements of the systems to be built. As mentioned in the previous section, analysis work up front, investing resources in understanding what the customer wants in a product flushes out problems before they are too expensive to fix.

  SDA prototypes a language for a domain using a functional language interpreter. This provides the customer with a sub-optimized working version of the language. It also provides a complete specification of the language which can be used during generator implementation.

- **Integration Proposal**

As SDA provides support for generator development and the integrated method will also contain the complete SDA language design, the integrated method will provide the SDA support for generator implementation.

4.3.8. Application Engineering Environment Creation

The application engineering environment creation process category includes activities for designing and developing a complete environment to support the creation of domain instances. An application engineering environment incorporates all of the products for application engineers to use for building applications in the domain.

- **FAST**
  - **Application Engineering Environment:** FAST provides suggestions for what tools would be included in an application engineering environment. These include: simulators, analysis tools, testing tools, model comparison tools, performance estimate tools, and support tools such as editors.

    FAST provides some guidance on what to include in an application environment. However, there are not details on how to design and build a user interface, or the other tools that would be built from scratch.

- **SDA**
  - **Application Engineering Environment:** SDA provides some support for designing the application engineering environment. These come mainly from the workflow analysis identifying possible tools and from identifying users for providing feedback of designs.

    SDA provides some guidance on how to determine what to include in the application engineering environment. However, there is no support for how to build or integrate the tools into the environment.
- **Integration Proposal**

As both methods are weak in the support of building an application engineering environment, it is left to future work to define activities which address this asset.

4.3.9. Application Engineering Process Capture

The application engineering process capture process category includes activities that support the description of a process to create domain instances using the new assets.

- **FAST**
  + **Application Engineering Process:** FAST encourages that an application engineering process be documented that guides application engineers in using the products developed during the domain engineering activities. The application engineering process is derived from the decision model.
- **SDA**
  - **Application Engineering Process:** SDA does not cover this.
- **Integration Proposal**

As only FAST provides any support for creating a standardized process for creating domain instances these activities are integrated into the integrated method. However, the decision model will not be enough to guide the process. Information on the resulting language must also be integrated into the application engineering process.

4.3.10. Documentation

The documentation process category includes activities for documenting all of the assets produced during domain engineering.

- **FAST**
  - **Documentation:** Along with the application engineering process, FAST requires that user guides, training material, and reference material be developed for the application engineering environ-

ment. However, no guidance is provided on how to create the documentation required.

- **SDA**
  - **Documentation:** SDA does not cover this.
- **Integration Proposal**

As with many domain engineering methods, documentation is not covered. It is left to future work to see how documentation activities can be integrated into the complete process instead of left to the end as an afterthought.

5. Related Work

There are many different domain analysis methods available. Some of them have been compared in survey papers by Arango and Prieto-Diaz [3,66]. However there are only a few that are well defined and used repeatedly in practice. These include: Feature oriented domain analysis (FODA), Organizational domain modeling (ODM), the Domain-Specific Software Architecture (DSSA) approach, and the sandwich method (by Ruben Prieto-Diaz). They are all based on the same underlying goal of analyzing and modeling the domain to achieve reuse from the similarities in a domain, however, how this is accomplished and to what extent the domain is analyzed, structured, and represented varies considerably.

Some of the characteristics that vary among domain analysis methods are: the level of formality in the method and products, the information capture techniques, the overall goals of the method, and the experiences of applying the method.

Along with these domain analysis methods there are also a number of approaches to domain engineering that only specify what the domain model should include. They do not have any processes for analyzing the domain and figuring out how to obtain the models. They only specify the results of the domain analysis and design process. Many of these approaches build generators from the models of the domains. This is similar to the goals of the SDA and FAST approaches. The Amphion and Draco approaches are two such methods and will be described following the four domain analysis methods.

This sections describes all of these methods and how they relate to SDA and FAST. A brief introduction to each related method and the characteristics it possesses are described. Both FAST and SDA are defined in detail in sections 3.3 and 3.4 respectively.

5.1. Feature oriented domain analysis (FODA)

Feature oriented domain analysis (FODA) is a domain analysis method being researched and applied by the SEI [10,47]. The basic premise is to model a domain as a set of features that a user commonly expects in applications in the domain. The goal of the FODA process is to create reusable components and architectures for a domain. The method doesn't have support for building languages which generate complete components from specifications as in SDA and FAST. FODA also does not use any formal methods to collect or capture domain information.

There are three main phases of FODA, context analysis, domain modeling, and architecture design. The context analysis phase has two products: a structure diagram which shows the domain in relation to neighboring domains, and a context diagram, which shows the interactions with neighboring

domains.

The domain modeling phase has four products: an information analysis, a features analysis, an operational analysis and a dictionary. The information analysis captures the entities and relationships of the domain. The features analysis has three sub-products: The context analysis, the operational analysis and the representation analysis. Features in the features model may be defined as alternative, optional, or mandatory. The commonalities and variabilities are captured this way. The context analysis defines who uses the domain, where the domain is used, and how the domain is used. Context features would also represent issues such as performance requirements, accuracy, and time synchronization that would affect the operations. The operational model defines what the required functions of the domain are (i.e. what the application does). The representation analysis defines the available input/output data representations of the domain. The dictionary captures the terminology of the domain

The architecture design phase of the FODA method captures the partitioning strategy for allocating domain features to software elements and a coordination model describing how these elements are activated and share information.

FODA has been applied to a few domains in the course of its development. These include the window manager domain and the movement control domain for the Army Tactical Command and Control System. The experiences have been documented in various SEI reports [10,17,34].

5.2. Organizational domain modeling (ODM)

Organization Domain Modeling (ODM) proposes to be a highly tailorable and configurable domain engineering method, useful for diverse organizations and domains, and amenable to integration with a variety of software engineering processes, methods and implementation technologies [46,52]. The method offers a systematic, exemplar-based approach to analysis of commonality and variability within both legacy systems and requirements for new systems. The method produces a domain architecture, and like FODA does not have support for creating domain-specific languages and application generators.

In addition to focusing on the strictly technical aspects of domain engineering, ODM emphasizes analysis of the diverse stakeholders that form the organization context within which each domain engineering effort is conducted. With this analysis, the domain analysis method is tailored to the domain of interest. The ODM domain analysis phase is therefore structured into two phases: Plan Domain Engineering, and Model Domain.

The ODM modeling life cycle details the transformation from descriptive modeling of legacy systems, artifacts, and experience to prescriptive specification of architecturally integrated assets, designed for well-specified range of variability and characterized in terms of features relevant to domain practitioners.

The Plan Domain Engineering phase includes a number of "upstream" tasks in domain engineering, many of which are not formally supported in other methods: identifying project stakeholders and objectives, and strategic domain selection from a candidate set of "domains of interest." The domain definition process iterates between a set of exemplars and a set of defining rules to uncover implicit contextual assumptions in the domain scope, and to identify domains related via a rich set of interconnection types, including generalization, specialization and "analogy" domains.

The Model Domain phase produces a Domain Dossier, Domain Lexicon, and a set of concept and feature models, the format and content of which are dependent on particular supporting methods chosen. A rich body of experience is emerging on modeling techniques and principles for model organization that best support the ODM approach. The integrated domain model (including the concept and feature models) captures both the commonality and variability of a given set of applications. ODM provides criteria for selecting from a wide variety of data elicitation techniques, including artifact analysis, interviewing of domain informants, scenario-based elicitation techniques, and process observation.

ODM has also only been extensively applied to a couple of domains during its development. The most extensive application of ODM to date has been on the Army STARS Demonstration Project, in combination with Reuse Library Framework (RLF) and the Conceptual Framework for Reuse Processes (CFRP) [51,50]. In addition, the Air Force CARDS Program has applied ODM in several different areas: as a means of structuring a comparative study on Architecture Representation Languages; on the Automated Message Handling System (AMHS) domain analysis effort; and for product-line analysis as part of the Hanscom AFB Domain Scoping effort (jointly with ESC, MITRE, and SEI).

5.3. Domain-Specific Software Architectures (DSSAs)

The DARPA Domain-Specific Software Architecture Program contained six projects to study the principles of domain-specific architecture development [25]. Multiple variations of a DSSA approach were developed. Described here is the IBM Federal Systems approach to DSSA, which has been documented by Will Tracz and Lou Coglianese [57]. This approach focuses on architecture and component reuse like FODA and ODM. It is also similar to these methods as it does not specify any formalisms for capturing information. The approach contains 5 high-level steps: scope, define elements, define constraints, develop architecture, develop reusable workproducts.

In the scoping step the goals of the domain engineering project, as well as, a high-level description of the domain are captured. Also captured are the resources available for the project. These are used to limit the scope of the domain. Finally the verification process to be used on the products produced during domain engineering is defined.

In the define elements step, domain concepts, or elements, are identified and described. Once the elements are identified, the relationships between the elements are defined. This includes capturing inheritance and aggregation relations as well as capturing dataflow and control flow between elements. All this information is used for creating a requirements document, which can be used for discussing the domain with customers.

In the next step, define constraints, requirements on the implementation are captures. These include software, hardware and performance requirements. These are used to limit the possible architectures.

The first three steps are considered the domain analysis process of DSSA. In the next step the architecture is developed. In this step the components are identified and the decision taxonomy for mapping variabilities to changes is captured. For each module, or component, the interface, behavior, entry and exit conditions, constraints, and configuration parameters are defined. Once this is accomplished, a matrix is created for tracing the requirements to the architecture components.

In the final step, reusable components are developed for use in creating domain instances.

The DSSA method described here has been applied to the Avionics domain in the ADAGE project [58]. Other projects have also applied the DSSA approach [25].

5.4. The sandwich method

The sandwich method was developed by Ruben Prieto-Diaz. Many of the domain-specific software architecture (DSSA) and software technology for adaptable, reliable systems (STARS) projects have been based on the work of Ruben Prieto-Diaz and one of the other pioneers in the domain analysis field, who Prieto-Diaz has worked with, Guillermo Arango [38,39,40,66].

The goal of the sandwich method is to specify components that can be implemented and put in a library for future reuse. A classification scheme is produced that aids developers in selecting components to use for their projects.

According to Prieto-Diaz, the purpose of doing a domain analysis is to produce a model of the domain that includes information on three aspects of the problem domain.

1. Concepts to enable the specification of systems in the domain,
2. Plans describing how to map specifications into code, and
3. Rationales for the specification concepts, their relations, and their relations to the implementation plans.

The method is named the sandwich method because the approach alternates between top down and bottom up development of domain abstractions. Prieto-Diaz's approach to domain analysis involves the

development of a faceted classification scheme. Facets are characteristics by which the domain objects may be described. Each facet is associated with a list of terms defining the possible values for that facet.

The domain analysis process consists of:

- identifying the relevant objects and operations in the domain
- finding a general description of the classes of objects in the domain, usually expressed in frames, and
- constructing a taxonomy of the domain objects, expressed as relationships between the frames.

This is not a formal method and it doesn't support the development of languages or generators directly. It is hard to tell how many applications of the method have been done, because the method is usually renamed and adapted for each project. In "Status Report: Software Reusability" Prieto-Diaz speaks of three experiences he has personally participated in [40]. They are in the domains of command and control systems, flight simulators, and avionics displays.

5.5. Amphion

Amphion is a domain engineering approach based on formal specification and automated deductive synthesis for program development [21,22,36]. The approach relies on rich and mature libraries of subroutines to exist for the domain of study. These libraries implement the required functionality of the domain.

The purpose of Amphion is to define a domain theory which captures a language for defining problems in the domain as well as axioms for how this abstract language maps to the subroutines implemented in the library. The languages are declarative and define how the inputs are related to the outputs in a problem specification. Problems can then be specified by users in the new language. Deductive program synthesis is carried out on the specification by the Amphion architecture to prove that a solution exists for which the outputs can be derived from the inputs. During the proof, the computation which calculates the outputs from the inputs is captured. The computations are in the form of subroutine calls to the libraries.

The domain analysis phase of the Amphion approach requires that experts in deductive program synthesis interact with domain experts to define the abstract domain theory, which is the language for expressing problems in the domain. The two groups of experts also capture the implementation axioms that are used in generation of the solution. The axioms capture the relationships between the abstract theory and the concrete theory that is embodied in the library subroutines.

The Amphion approach is similar to both SDA and FAST in that it captures a language for specifying new members of a family. The approach is most similar to SDA in that it captures formally the language and the semantics of the language. However, Amphion differs in that it uses logic to capture the

relationships where SDA uses the functional paradigm instead.

The Amphion approach has been applied to at least one domain with success. The domain of solar system kinematics, which is supported by the SPICELIB libraries developed by the Navigation Ancillary Information Facility (NAIF) at NASA's Jet Propulsion Laboratory, has a rich domain theory of over 300 axioms. Amphion is also being applied to the domains of numerical aerodynamic simulation and space shuttle flight planning.

5.6. Draco

The Draco approach for achieving reuse in a domain is one of the earliest approaches to constructing software from reusable components within a specific domain [29,30]. The approach was developed at the Information and Computer Science Department at the University of California, Irvine.

Draco consists of a system which takes as input a domain description and produces an environment for specifying and analyzing specifications in the new language. The domain description consists of a parser for a new domain-specific language, a pretty printer which translates the internal form of the language back into the external syntax, optimizations, components which implement parts of the domain functionality, generators which generate solutions for algorithmic knowledge, and analyzers which help create and test specifications in the new language.

The domain description is defined by domain experts and method experts who have knowledge of the Draco system and domain modeling. The domain is initially analyzed for objects and operations that are common to systems in the domain. These concepts are then modeled and designed in terms of concepts already known to the Draco system. Like Amphion there is not a process for coming up with the appropriate abstractions in a domain.

The Draco approach is similar to SDA and FAST in that it creates a new domain-specific language for each domain which is used to specify new instances. However, the approach is not formal and as mentioned above, there is not a method for analyzing the domain.

The Draco approach has been applied to some domains [29,30,31]. However, documentation of the results is limited.

6. Conclusions

In the past, systematic approaches to domain-specific language design have been inadequate. Domain engineering provides an opportunity for designing and developing DSLs. FAST provides a domain analysis approach that elicits the core domain knowledge from experts. SDA provides a systematic, tool-supported approach for the design and implementation of domain-specific languages, built on the rich tradition of language principles that are the core of the programming language community. However, these two approaches need to be integrated before they can be used together.

The thesis of this research work was that by incorporating the strengths and weaknesses of two methods, a stronger method would arise. The strengths and weaknesses of both FAST and SDA were defined and analyzed. The integration shows how the strengths of one method can be used to balance the weaknesses of the other. However, the integrated method still has to be validated on a real project.

Explicitly defining the integrated method is a first step in testing whether principled language design can be practiced as a reuse principle by software developers, and that it is not the sole province of highly trained experts. The integration of SDA with domain engineering methods provides a mechanism for bringing these "academic best practices" into engineering use. There, they can be applied and evaluated on real world problems.

Validating the method would require multiple applications of the method to various domains in order to come up with data on the process and products of a complete effort. To compare the method to the current approaches used, the current approaches must also be analyzed.

Preliminary observations showed that when domain experts and language designers schooled in formal methods collaborate, formal models from the SDA method could be used to check the domain expert's models for consistency and completeness. Similarly, a commonality analysis along with input from domain experts provided an SDA team with information to base an initial mathematical model on. This further indicates the need to work together, as the commonality analysis on its own did not provide the right level of abstraction for a high level model.

Work still needs to be done on making the formal models more accessible to the domain experts. Having the domain experts and the method experts work together is a start. However this requires that method experts participate in all domain analyses and even then some of the formalisms may not be accepted. Observations showed some possibilities for formalizing abstractions, however there was also resistance.

Many of the problems are related to abstraction in general not just formal abstractions. As we have observed, the engineers validate abstractions by testing them against as many examples as they know, and

then still they may not trust them. Further work has to be done on coming up with more ways to get abstractions extracted, represented, and validated.

Terminology

- **Application Engineer**: Person who performs application engineering.
- **Application Engineering**: Process of creating application products.
- **Application Generator**: Tool which automatically transforms a domain-specific language into a lower level language. (Figure 2 on page 3)
- **Application Modeling Language**:
- **Application Products**: Things produced during application engineering. Might include code, documents, training material, etc.
- **Architecture:** Garlan and Shaw describe an architecture as a system's structure or organization [45]. The architecture defines the relationships among the major components of the system with system specific details abstracted away.
- **Commonality Analysis:** The FAST methods domain analysis approach.
- **Component:** A component is any portion of an application or a full application.
- **Component Generator:** See application generator.
- **Decision Model**: Model of the variabilities or decisions that an application engineer has to make when creating a domain instance.
- **Denotational Semantics**: Approach for specifying semantics in which the meaning of elementary forms are directly specified, and the meanings of composites are specified in terms of the meanings of their immediate constituents [54].
- **Domain:** A distinct functional area that can be supported by a class of software systems with similar requirements and capabilities" [19].
- **Domain Engineering**: Process of creating reusable assets to support application engineering. Might include dictionary of terms, domain-specific language, application generator, etc. (Figure 1 on page 2)
- **Domain Expert**: Person with experience and knowledge in a domain.
- **Domain Model**: User view model of the entities, relationships and operations in a domain.
- **Domain-specific Language (DSL)**: high-level language which is used for specifying family members in a domain. Incorporates notations and concepts from the domain.
- **Environment Interface Specification**: Specification of the legacy system where the generated component must function.
- **FAST:** Family-oriented Abstraction, Specification and Translation.
- **Formal Methods**: Formal methods provide a rigorous mathematical basis to software develop-

ment. They work on improving the correctness, and reliability of produced code [Bowen The Ten Commandments of Formal Methods].

- **Functional Languages:** Programs are made up of functions with inputs and outputs, no assignment, no side effects, no flow of control. different way to modularize functions and glue them together. Provides two new glues that greatly improve modularization: reduce and lazy evaluation [16].
- **Monads**: Categorical concept for structuring semantics [61].
- **Neighboring Domain**: Neighboring domains are any domains that interface with the domain being investigated.
- **Reusable Asset:** Reusable assets include processes, products, and tools that are intended to assist application engineers in developing and maintaining product instances, or applications, in the domain.
- **SDA:** Software Design Automation.
- **Solution Model**: Solution view model of the components that provide the necessary functionality for a domain.
- **Target Language**: Language used in the domain's environment. The language that must be generated from a specification.
- **Translator**: Mechanism and tools used for translating a DSL specification into a target language.
- **User View**: Presentation of the domain to the application engineers and domain experts in concepts that are familiar to them.
- **Users**: People who use the Application Products.

References

[1] Allen, R. A Formal Approach to Software Architecture. Ph.D. Thesis. *Technical Report CMU-CS-97-144*, Carnegie Mellon University, May 1997.

[2] Arango, G. and Prieto-Diaz, R. (eds.) Domain Analysis Concepts and Research Directions. In *Domain Analysis and Software Systems Modeling*, pp. 9-31, IEEE Computer Society Press, 1991.

[3] Arango, G. Domain Analysis Methods. In *Software Reusability*, eds. Shaefer, W., Prieto-Diaz, R. and Matsumoto, M. Ellis Horwood, 1993.

[4] Ardis, M. and Weiss, D. Defining Families: The Commonality Analysis. *Proceedings of the Nineteenth International Conference on Software Engineering*, pp. 649-650, IEEE Computer Society Press, May 1997.

[5] Armitage, J. Process Guide for the Domain-Specific Software Architectures (DSSA) Process Life Cycle. *Technical Report CMU/SEI-93-SR-021*, Carnegie Mellon University Software Engineering Institute, 1993.

[6] Bell, J. et al. Software Design for Reliability and Reuse: A Proof-of-Concept Demonstration. *Proceedings of TRI-Ada '94*, pp. 396-404, ACM, November 1994.

[7] Cambell, G., Faulk, S. and Weiss, D. Introduction to Synthesis. *Technical Report INTRO_SYNTHESIS_PROCESS-90019-N*, Software Productivity Consortium, 1990.

[8] Cleaveland, J. Building Application Generators. *IEEE Software* 5(4), pp. 25-33, July 1988.

[9] Clements, P. A Survey of Architecture Description Languages. *Eighth International Workshop in Software Specification and Design*, pp. 16-25, Paderborn, Germany, March 1996.

[10] Cohen, S., Stanley J., Peterson, A. and Krut, R. Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain. *Technical Report CMU/SEI-91-TR-28*, ADA 256590, Carnegie Mellon University Software Engineering Institute, 1992.

[11] Dijkstra, E. W. Structured Programming. In *Structured Programming,* eds Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. Academic Press, 1972.

[12] Draghicescu, M. The role of model building in domain analysis. *Technical Report*, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1996.

[13] Frakes, F., Prieto-Diaz, R., and Fox, C. DARE: Domain Analysis and Reuse Environment. *7'th Annual Workshop on Software Reuse*, 1995.

[14] Gupta, N., Jagadeesan, L., Koutsofios, E. and Weiss, D. Auditdraw: Generating Audits the FAST Way. *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, January 1997.

[15] Hook, J. and Walton, L. The Design of Message Specification Language. *Technical Report,*

Department of Computer Science and Engineering, Oregon Graduate Institute, June 1997.

[16] Hughes, J. Why Functional Programming Matters. In *Research Topics in Functional Programming*, chapter 2, pages 17--42, ed. Turner, D., Addison-Wesley, 1990.

[17] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21*, ADA 235785, Carnegie Mellon University Software Engineering Institute, 1990.

[18] Kieburtz, R. et al. A software engineering experiment in software component generation. *Proceedings of the 18th International Conference on Software Engineering*, pp. 542-553, IEEE Computer Society Press, March 1996.

[19] Kogut P. and Nilson, R. Domain Engineering Methods and Tools Handbook Volume I — Methods, Comprehensive Approach to Reusable Defense Software (CARDS), *Technical Report*, Software for Adaptable, Reliable Systems (STARS), 1994

[20] Kruegar, C. Software Reuse. *ACM Computing Surveys*, Vol. 24, No. 2, pp. 131-183, June 1992.

[21] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I. A Formal Approach to Domain-Oriented Software Design Environments. *Proceedings of the Ninth Knowledge-based Software Engineering Conference*, 1994.

[22] Lowry, M., Philpot, A., Pressburger, T., and Underwood, I. AMPHION: Automatic Programming for Scientific Subroutine Libraries. *Proceedings of the International Symposium on Methodologies for Intelligent Systems*, pp. 326-355, October 1994.

[23] McIllroy, M. D. Mass Produced Software Components, *Software Engineering*, pp. 138-150, NATO Science Committee, January 1969.

[24] Meijer, E. et al. Functional programming with bananas, lenses, envelopes and barbed wire. *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pp. 124-144, Vol. 523, Lecture Notes in Computer Science, Springer-Verlag, August 1991.

[25] Mettala, E. and Graham, M. The Domain-Specific Software Architecture Program, *Technical Report CMU/SEI-92-SR-009*, Carnegie Mellon Software Engineering Institute, 1992.

[26] Mili et al. Reusing Software: Issues and research directions. *IEEE Transactions on Software Engineering*, Vol. 21, No. 6, pp. 528-561, June 1995.

[27] Moggi, E. Notions of Computation and Monads. *Information and Computation*, 93(1), pp. 55-92, 1991.

[28] Mosses, P. *Action Semantics*. Cambridge Tracts in Theoretical Computer Science, Number 26, Cambridge University Press, 1992.

[29] Neighbors, J. Draco: A method for engineering reusable software systems. In *Software Reusability Vol 1 — Concepts and Models*, eds. Biggerstaff, T. and Perlis, A. ACM Press, pp 295-319, 1989.

[30] Neighbors, J. The Draco approach to constructing software from reusable components. *IEEE Transactions on Software Engineering*, Vol. 10, No. 5, pp 564-574, September 1984.

[31] Neighbors, J. The evolution from software components to domain analysis. *International Journal of Software Engineering and Knowledge Engineering*, Vol. 2, No. 3, pp. 325-354, May 1992.

[32] Oliva, D. Baseline performance measurements of un-optimized generated code. *Technical Report*, Department of Computer Science and Engineering, Oregon Graduate Institute, February 1995.

[33] Parnas, D. On the Design and Development of Program Families, *IEEE Transactions on Software Engineering,* Vol. SE-2, No. 1, pp. 1-9, Mar. 1976.

[34] Peterson, A. and Cohen, S. A Context Analysis of the Movement Control Domain for the Army Tactical Command and Control System (ATCCS). *Technical Report CMU/SEI-91-SR-3*, Carnegie Mellon University Software Engineering Institute, 1991.

[35] Peyton Jones, S. et. al. Scripting COM Components in Haskell. To appear in *Proceedings of The Fifth International Conference on Software Reuse,* June 1998.

[36] Pressburger, T. and Lowry, M. Automating Software Reuse with Amphion. *NASA workshop on Software Reuse*, Fairfax, VA September 1996.

[37] Pressman, R. *Software Engineering: A Practitioners Approach.* McGraw-Hill, 1994.

[38] Prieto-Diaz, R. Domain Analysis For Reusability. *Proceedings of the Eleventh Annual International Computer Software and Application Conference (COMPSAC)*, pp. 23-29, 1987.

[39] Prieto-Diaz, R. and Arango, G. (eds) *Domain analysis and software systems modeling.* IEEE Computer Society Press, 1991.

[40] Prieto-Diaz, R., Status Report: Software Reusability, *IEEE Software*, Vol. 10, No. 3, pp. 61--66, 1993

[41] Schmidt, D. *Denotational Semantics: A Methodology for Language Development.* Allyn and Bacon, Boston, 1986.

[42] Schmidt, D. Programming Language Semantics. *ACM Computing Surveys*, Vol 28, No. 1, pp. 265-267, Mar 1996.

[43] Shaw, M. and Gaines, B. Comparing Conceptual Structures: Consensus, Conflict, Correspondence and Contrast. *Technical Report*, Knowledge Science Institute, 1989.

[44] Shaw, M. and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[45] Shaw, M. and Garlan, D. An Introduction to Software Architecture. *Technical Report CS-94-166*, Carnegie Mellon University, School of Computer Science, January 1994.

[46] Simos, M. Organization Domain Modeling (ODM): Formalizing the Core Domain Modeling Life Cycle. *Proceedings of the Symposium on Software Reusability SSR'95*, pp. 196-205, April 1995.

[47] Software Engineering Institute, Model-Based Software Engineering. http://www.sei.cmu.edu/technology/mbse/is.html, April 25, 1998.

[48] Software Engineering Technology, Inc. *The Host at Sea Buoy Project: Mission Statement, Black Box Specification, and Construction Plan*. March 1991.

[49] Software Productivity Consortium. Reuse-Driven Software Processes Guidebook, Version 02.00.03. *Technical Report SPC-92019-CMC*, Software Productivity Consortium, November 1993.

[50] Software Technology for Adaptable Reliable Systems. *STARS Conceptual Framework for Reuse Processes (CFRP), Vol. I: Definition*, Version 3.0, Unisys STARS TC STARS-VC-A018/001/00, STARS Technology Center, Arlington VA, October 1993.

[51] Software Technology for Adaptable Reliable Systems. *Army STARS Demonstration Project Experience Report*, STARS-VC-A011/011/00, Dec 1995.

[52] Software Technology for Adaptable Reliable Systems. *Organization Domain Modeling (ODM) Guidebook*, Version 2.0. Unisys STARS Technical Report STARS-VC-A025/001/00, Reston VA, June 1996.

[53] Taha, W. and Sheard, T. Multi-Stage Programming with Explicit Annotations. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 203-217, 12-13 June 1997.

[54] Tennent, R. D. *Principles of Programming Languages*. Prentice-Hall International, London, 1981.

[55] Tolmach, A. Elaborating the specification of Message Specification Language. *Technical Report*, Department of Computer Science and Engineering, Oregon Graduate Institute, July 1995.

[56] Tolmach, A. and Oliva, D. From ML to Ada. To appear in the *Journal of Functional Programming*, original version May 1997.

[57] Tracz, W. and Coglianese, L. Domain-Specific Software Architecture Engineering Process Guidelines, *Technical Report ADAGE-IBM-92-02*, Loral Federal Systems, 1992.

[58] Tracz, W. and Coglianese, L. An Adaptable Software Architecture for Integrated Avionics, *Technical Report ADAGE-IBM-93-03*, Loral Federal Systems, 1993

[59] Volpano, D. and Kieburtz, R. Software Templates. In *Proceedings of the ICSE 8*, pp. 55-61, August 1985.

[60] Wadler, P. Comprehending monads. *Proceedings of Conference on Lisp and Functional programming*, pp. 61-78, ACM Press, 1990.

[61] Wadler, P., Monads for functional programming. *Marktoberdorf Summer School on Program Design Calculi*, Springer Verlag, NATO ASI Series F: Computer and systems sciences, Vol. 118, pp. viii+409, 233-264. August 1992.

[62] Walton, L. Domain Analysis for Switch Maintenance Configuration Control. *Technical Report*,

Department of Computer Science and Engineering, Oregon Graduate Institute, June 1996.

[63] Walton, L. and Hook, J. Creating and verifying domain-specific design languages. *Technical Report*, Department of Computer Science and Engineering, Oregon Graduate Institute, September 1995.

[64] Walton, L. and Hook, J. Message Specification Language (MSL): A domain-specific design language for message translation and validation. *Technical Report*, Department of Computer Science and Engineering, Oregon Graduate Institute, September 1994.

[65] Walton, L. and Hook, J. On understanding a commonality analysis. *Proceedings of the OOPSLA'96 Workshop on Domain Analysis: Processes and Results*, October 1996.

[66] Wartik, S. and Prieto-Diaz, R. Criteria for Comparing Reuse-Oriented Domain Analysis Approaches. *International Journal of Software Engineering & Knowledge Engineering*, 2(3), pp. 403-431, 1992.

[67] Weiss, D. Software Synthesis: The FAST Process. *Proceedings of the International Conference on Computing in High Energy Physics (CHEP)*, Sept. 1995. Available from http://www.hep.net/conferences/chep95/welcome.htm.

[68] Weiss, D. and Lai, R. A formal model of the FAST process. *Technical Report BL0112650-95-09TM*, AT&T Laboratories, 1995.