Using Tabular Expression Input to Specify and Generate Program Family Members

Using Tabular Expression Input to Specify and Generate Program Family Members

By
Deliang Han

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science
Department of Computing and Software
McMaster University

September 26, 2001

MASTER OF SCIENCE (2001) McMaster University
(Computing and Software) Hamilton, Ontario, Canada

TITLE: Using Tabular Expression Input to Specify and Generate Program Family Members

AUTHOR: Deliang Han
B.Sc. Jilin Institute of Technology
Changchun, P.R.China

SUPERVISOR: Dr. David L. Parnas

NUMBER OF PAGES: vii, 79

ABSTRACT

When one is to develop program families traditional programming methods which are intended for the development of a single program are not appropriate. There are several different approaches available. For example the Draco approach in [18] and the product line engineering process described in [23]. Here we will focus on the FAST process developed at Lucent [24] because it is a reasonably systematic process for engineering families that has been used at Lucent Technologies for years and has proven successful.

Lucent/Bell Labs have had success in reducing programming costs and speeding up development times by using the FAST process. In the FAST process, after identifying the commonalities and parameters of variation in family of programs, a special purpose programming language is developed. This language is suitable for writing members of the family. The common features of the family are already built-in; only the differences need to be programmed.

McMaster University's Software Engineering Research Group (Hamilton, Ontario, Canada) is developing a set of tools to support the use of a broad class of tabular expressions(TTS). Using tabular expressions one may present complex conditional expression in a way that is more easily read, analyzed, and more likely to be correct than either conventional mathematics or conventional programs.

In this work I propose to combine these two technologies. I will enhance the FAST process using tabular notation and table-based tools. The results of the commonality analysis of the FAST process, as developed at Lucent, will be used to develop a set of incomplete tabular expressions. The completed parts will represent the commonalities. The incomplete parts will correspond to the parameters of variation. The application engineer must complete the tables in order for the Code Generator tool of TTS to generate a prototype family member.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my supervisor Dr. David L. Parnas, for his guidance, advice, and enthusiasm throughout my thesis work. Without his constant encouragement and support, it would have been impossible for me to finish this work.
I would also like to thanks Dr. David Weiss for his valuable suggestions and comments about the survey which is Chapter 2 in this work.
I would also like to thank Ou Wei and NanNan Wang for their help on the TTS system. I really appreciate the helpful discussions and valuable suggestions.
Special thanks to my wife, Ping Fan and my whole family, for their love, encouragement and support.
Finally, I would like to thank the Lucent Technologies for the financial support.

CONTENTS

Chapter 1

Introduction

This chapter provides a brief introduction to the thesis including its background, motivation, preview of the tool and formal outline.

1.1 The Need for Program Families

Successful software products bring with them the same advantages and disadvantages of successful products in other fields; they provide capabilities that their customers need and want. They also provide income to the companies that create and sell them, and they evolve to meet changing needs. If the developers of a successful product can stay abreast of or lead their marketplace, their software may become quite long-lived and may come to exist through many variations. Maintaining such software frequently means that it must undergo continual redevelopment and require special sources of developmental expertise, i.e., that its developers become specialists in particular aspects of the product and that new developers require long training periods to maintain the software.
How shall we deal with this case?

One example of such a product is the software for Lucent Technologies' 5ESS switch, which includes millions of lines of legacy code. The switch, including the software for it, is about 15 years old and exists in many versions in the US and the international markets. It still continues to evolve as a product. There is competitive pressure to produce new versions of the software rapidly and to maintain the quality of the service offered by the switch.

This raises some interesting questions:

- When we develop a set of similar programs, do we need to write a set of requirement documents one by one and develop all the programs from scratch?
- When we face the problem that requirements are incomplete, misunderstood, poorly defined, and changeable in ways that are difficult to manage, what shall we do?

One answer for all the above questions are creating program families. Program family-oriented software development was suggested by Dijkstra and others in the software

engineering literature as early as 1972 [11]. David L. Parnas described approaches for building software families in the mid-1970s [19]. We take the definition of a program family from Parnas's paper [19]:

"We consider a set of programs to constitute a family whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members."

The program family approach works well in situations where there are different versions of a system (different family members), all of which share common requirements, design, or code. Examples of such situations are (from[10]):

- Systems that have the same requirements but must execute on different platforms, (e.g., database management systems or compilers for the same language).
- Systems that store and use the same data, but vary the processing of the data, (e.g., systems that provide different types of reports based on the same data).
- Systems that control and monitor the same devices, but have somewhat different behaviour, (e.g., telecommunications systems with different features or different billing algorithms).
- Systems that provide the same interface to the user but implement their behaviour differently, (e.g., different word processors with the same or nearly the same features).

A good example, which we will discuss in detail later, is the airline class-of-service upgrade policy, different family members use different criteria to select passengers who will go up front if there are extra seats available in the first class. They may vary policy from time to time or location to location.

1.2 Issues that Program Families Address

All program family-based approaches are designed to attack the major problems that lead to increased software development intervals and that inhibit attention to careful engineering. They all suggest that we should often make the family the unit of software development where the set of programs that constitute the family are identified by defining a set of commonalities and variabilities. The program families are not viewed as successive modifications to previous versions but as refinements of a common abstraction. Each member of a family can be characterized entirely in terms of how it refines the common abstraction [19].

They all address the following software development problems:

- Requirements that are incomplete, misunderstood, poorly defined, and changeable in ways that are difficult to manage.
- Rediscovery and reinvention, especially in situations where developers must locate and interview people who are expert(s) about one part of a system in order to understand the consequences of making a change elsewhere in the system.
- The need to adapt legacy software to new technology and new requirements, especially where the software does not accommodate change well. Also where the code is the only trustworthy documentation, and where the knowledge needed safely to change the code resides in only a few peoples' heads.
- Redundant specification, especially in situations where the information that you need and use to explain and develop a system is explained in one place and one notation during requirements determination, then explained again in another place and another notation during design, again during coding, and finally during testing.

And they all expect the following benefits:

- More efficient production of family members.
- Ease of change.
- Productivity and Quality.
- Manageability.

1.3 Contribution of this work

When program families are developed traditional programming methods (which are intended for the development of a single program) are not appropriate. There are several different approaches available e.g., the Draco approach in [18] and the product line engineering process described in [23]. Here we will focus on the FAST process developed at Lucent [24] because it is a reasonably systematic process for engineering families. It has been used at Lucent Technologies for years and has proven successful.

1.3.1 Overview of the FAST process

The FAST (Family-oriented Abstraction, Specification, and Translation) process is a family-based software development process. It was introduced into AT&T in 1992 by David Weiss and now in use at Lucent Technologies. The goal of FAST is to provide a systematic approach to analyzing potential families and to develop facilities for efficient production of family members. The FAST process consists of (2) subprocesses:

1) Domain Engineering: This process works in defining the family and developing an application engineering environment for producing family members. Defining the family means identifying potential family members and characterizing what they have in

common and how they differ. The result is a commonality analysis document. The commonality analysis is used to guide the design of a language for specifying family members. A compiler for that special language is then developed as part of the application engineering environment.

2) Application Engineering: This process uses the application engineering environment to produce family members. The application engineers need to code using the language developed during the domain engineering phase to specify the family members. Then they can be compiled.

1.3.2 Enhancing the FAST Process Using Tabular Notation and Table-Based Tools

In the process described in this work, we use the commonality analysis document to guide us designing an incomplete TTS (Table Tool System) table [22]. This is done after identifying the commonality and parameters of variation, instead of designing a language. The completed parts will represent the commonalities. The incomplete parts will correspond to the parameters of variation. Thus, in the application engineering subprocess the application engineers do not need to program but to complete the table. After this we use the Code Generator tool of the TTS system to generate family members from the table.

1.4 Thesis Outline

Chapter 2 provides a survey of work on program families.

Chapter 3 describes the FAST process in detail. It also discusses the limitations of the FAST process.

Chapter 4 gives an introduction to the TTS system.

Chapter 5 provides several examples of FAST applied to an airline upgrade policy family.

Chapter 6 provides another examples of FAST applied to a mobile phone plan family.

Chapter 7 describes the design of the AUFMG tool.

Chapter 8 is the user guide for the AUFMG Tool.

Chapter 9 gives the conclusions and future work for this thesis.

Chapter 2

A Survey of Work on Program Families

This chapter provides a survey about work on program families.

2.1 Introduction

A program family [19] is a set of similar programs. The common properties of these programs are so extensive that it is advantageous to consider the set as a whole while designing and developing them. When one is to develop program families traditional programming methods, which are intended for the development of a single program, are not ideal. This paper discusses the process of designing and developing program families and surveys work on program families.

The work surveyed is: [19], [20], [12], [18], [7], [5], [6], [4], [3], [16], [23], [10], [8], [9], [13], [14], [15] and [24].

These papers have several things in common:

1. They all suggest that we should make the family the unit of software development. Here the set of programs that constitute the family are identified by defining a set of commonalities and variabilities that determine the familial properties. The program families are not viewed as successive modifications to previous versions but as refinements of a common abstraction. Each member of a family can be characterized entirely in terms of how it refines the common abstraction.
2. They all address the following software development problems:
   - Requirements that are incomplete, misunderstood, poorly defined, and changeable in ways that are difficult to manage.
   - Methods and toolsets that fail to integrate smoothly across the system life cycle.
   - Inadequate methods and tools for effective reuse.
3. They all expect the following benefits:
   - More efficient production of family members.
   - Ease of change.
   - Productivity and Quality.

- Manageability.

Items [18], [6], [4], [23] and [24] propose systematic approaches to designing and implementing program families. Others present technologies that are used in these processes.

The approaches in [18], [7], [5], [6], [4], [23], [15] and [24] are general purposes. In [3], [16] and [14] specific application areas are discussed.

In [7] and [16], a translator is needed to translate the high level language to a lower level.

In [5] and [6], the term “abstract interface” was used and discussed in [12] in detail. The approaches used in [5] and [6] are based on the idea of identifying abstractions that are common to a family and using them as the basis for designing a specification language for family members.

In [5] reuse libraries are discussed.

Items [16], [14] and [15] discuss application modelling languages and domain specific languages.

Items [10], [9], [13] and [24] are about the Lucent FAST approach to building program families.

2.2 Notation and Terminology

This section explains the notation and terminology used in this survey.

Abstract Interface: An interface that provides access to the services of the product or component but hides the implementation.

An application generator: A tool that generates required products from an application model. Details about how to build application generators are shown in [7].

Application “Engineering”: The process of creating members of a family (applications) using the production facilities for the family.

During application engineering, the application developer specifies an application model and generates deliverable products from it.

The tasks of application “engineering” are as follows:

- Transforming the customer's input into a precise requirements specification for the member of the application family to be developed. The transformation is done using the application modelling language developed during domain analysis.
- Analysing the application model to ensure that it meets the customer's needs.
- Generating deliverable products, including documentation and code, for the application.

APPLICATION ENGINEERING PROCESS: Process for using an application engineering environment to produce members of a domain. The process is specifically designed for the domain.

APPLICATION MODELLING LANGUAGE: A language for describing or specifying a member of a domain. It allows you to specify a domain member by the requirements that distinguish it from other domain members; common requirements need not be specified. An example of application modelling language is [16].

BUILDING-BLOCK TECHNOLOGY: Also called component technology, these are methods of constructing software systems using reusable components.

COMMONALITY: An assumption that applies to all members of a family.

DOMAIN: The set of applications for the program family.

DOMAIN ANALYSIS: The process of formalizing the expert knowledge in a particular domain to create standard models for the requirements representation. It also creates standard designs for the system architecture. The product of the domain analysis is called a domain model, a specification for an application engineering environment. A domain model has three parts:

1. An application modelling language;
2. A reuse architecture; and
3. Product composition - specification of the relation between the application modelling language and components of the reuse architecture. [4] shows one kind of composition.

DOMAIN ENGINEERING": The process of creating the production facilities for a family. It produces a domain-specific software production environment used to do application engineering. This is also called product line engineering ([23] and [24]).

DOMAIN IMPLEMENTATION: The process that develops a reuse library and an application generator.

DOMAIN MANAGEMENT: The process that allocates and monitors human resources to do domain analysis and domain implementation.

DOMAIN SPECIFIC LANGUAGE: A tool-supported formal specification language created for a specific domain. [14] and [15] show the design of domain specific language.

FAMILY: A family is a set of similar programs whose common properties are so extensive that it is cost-effective to consider the set as a whole while designing and developing them.

REUSE ARCHITECTURE: A reuse library and its contents (a set of adaptable components), including both documentation and code. [3] is an example of reuse architecture.

REUSE LIBRARY: A library of reusable components and supporting mechanisms for producing code. It also contains documentation that can be used as an implementation of a reuse architecture.

PARAMETERS OF VARIABILITY: Ways in which members of a family may differ.

2.3 Result of Domain Analysis from Avionics

Batory [3] presents the results of a domain analysis, a reuse architecture, and the lessons that were learned from the development of ADAGE (a project to define and build a domain-specific software architecture environment for assisting the development of avionics software).

2.4 The Design and Implementation of Hierarchical Software Systems with Reusable Components

Genesis (the first building-blocks technology for Database management systems) and Avoca (a system for constructing efficient and modular network software suites using a combination of preexisting and newly created communication protocols) are both successful examples of software component/building-block technologies and domain modelling, By unifying the concepts behind these, Batory [4] presents a domain-independent model of hierarchical software system design and construction that is based on interchangeable software components and large-scale reuse.

This approach can be used to define reuse architectures for many different domains.

A key feature of this process is recognization of the fundamental role of symmetric components (components that can be composed in arbitrary order) in large-scale reuse.

This paper demonstrates that the design and development of program families can be achieved through domain analysis. This represents an effort to formalize the similarities and differences among systems of a mature and well understood domain.

2.5 Synthesis Approach

Campbell [5] introduces the Synthesis project, developed at the Software Productivity Consortium. Synthesis is a process for developing program families.

According to this approach development of program families can be divided into two subprocesses: domain engineering and application engineering. The details about how the two subprocesses work are presented.

In [5], useful terminology for software reuse and the software engineering field in general is introduced. Many of these terms are in the glossary.

In this paper Campbell also presents the life cycle of design and development for program families. The benefits and obstacles of the design are also introduced.

2.6 Abstraction Based Reuse

Campbell [6] discusses the idea of storing abstractions of components in a reuse library and creating automatically adapted instances as they are needed to build members of families.

This paper presents the process for building a reuse library. First, it defines the conventional model, then it proposes a new approach: Abstraction-Based reuse repositories. The new approach saves both storage space and adaptation of each component before reuse.

The conventional approach to reusing a component is through minor component modification to suit new needs. Instead of starting with individual family members it is possible to start with an abstract description of the entire family; often all possible family members can be refined directly. From this basis, a view of the reuse library results in which abstractions populate a taxonomy of constructable components and component instances. These are delivered as instantiations of abstract component families. Each abstraction is a characterization of a family of software components. Reusable components are obtained by selecting and instantiating a family abstraction by making a set of prescribed design decisions.

2.7 Building Application Generators

Cleaveland [7] presents technology available for building application generators, the product of domain analysis. Further, he explains how application generators work by translating specifications into application programs. Normally, the specifications are described using application modelling languages used by the application generator to automatically create one or more products -- typically a segment of code, a subroutine, or a software system.

The application generator translates high-level information into a low-level implementation; in fact, it is a language compiler. To change or modify this product one changes the input specification and reruns it through the generator.

Based on Cleaveland’s experience of building successful DCGS (the Dialogue-code-Generation System), the steps are:

* Recognizing domains.
* Defining domain boundaries, the range of the generator.
* Defining an underlying model.
* Defining the parameter of variability and commonality.
* Defining the specification input.
* Defining products.

* Implementing the generator.

2.8 Using C++ as an AML to Apply the FAST Process

Coplien [8] explains the idea of a multi-paradigm design with a focus on its application to C++. In this book the author uses C++ as an application modelling language to do software specification. Here multi-paradigm development is used to build a C++ class library to express commonalities.

The discussions of commonality and variability analysis for the purposes of constructing families provide insight on the FAST commonality analysis process described in [24].

This book takes advantage of commonality and variability analysis to make effective use of multi-paradigm design in C++. Commonality and variability analyses form the fundamental underpinnings of design described therein.

Coplien describes the kinds of commonalities and variabilities that are germane to each of several paradigms, along with the C++ language features that express those paradigms.

The idea in this document differs from the FAST process in that it does not need to develop an application modelling language. Instead it uses C++ language as an application modelling language(AML).

2.9 SCV Approach in Software Engineering

Item [9] describes SCV (Scope, Commonality, and Variability) analysis and discusses the benefits and challenges of its application. Some examples about how to apply SCV analysis are given.

The main steps in SCV analysis are:

* Establishing the scope; the collection of objects under consideration.

* Identifying the commonalities and variabilities.

* Binding the variabilities by placing specific limits on each variability.

* Exploiting the commonalities.

* Accommodating the variabilities.

The benefits of SCV analysis include opportunities for rapid new development due to reuse as well as decreased development costs, another development is the rapid creation of new family members.

Based on the authors' experience the following basic principles for SCV analysis are:

* Make S, C and V explicit.

* Choose S to balance generation costs and family size.

* Search for C to maximize reuse.

* Bind the variabilities to minimize production costs.

* View programs as mathematical expressions; use 2 factors to find commonality.

* Periodically revisit each SCV analysis.

The authors also discuss the FAST approach, using SCV analysis to identify, formalize and document commonalities and variabilities.

2.10 Domain Engineering

Cuka and Weiss [10] discuss the idea of families using the definition from [19]. The family approach works well in situations where there are different versions of a system all of which share common requirements, design, or code. The example of a family in this paper is the C&R family.

This paper also describes an example of the FAST approach to a common telecommunications domain. The key new idea in the FAST approach is a step that explicitly defines the family. That is the basis for designing a language for specifying family members and creating an environment for analysing and generating family members. Such languages are known as application modelling languages (AMLs); for example, SPEC in this paper.

2.11 Procedure for Designing Abstract Interfaces for Device Interface Modules

Item [12] introduces the definition of abstract interfaces and abstraction are essential for building a reuse library no matter what kind of techniques used.

This is important because an appropriate abstraction for a given purpose is easier to study than the actual objects because it omits irrelevant details. A result obtained by studying an abstraction can be applied to any system represented by the same abstraction.

The abstract interface technique is related to program families because both the synthesis approach in [12] and the reuse library technology described in [20] rely on it.

2.12 Understanding a Commonality Analysis

The unique contribution of [13] is that it identifies specific strengths and weaknesses in the use of a Commonality Analysis as it relates to domain learning, and in preparation for language design. By analysing the Commonality Analysis process the authors reach the conclusion that the “Commonality Analysis process seems to be an excellent method for acquiring domain information and for building domain knowledge in a community”. A successful application of the Commonality Analysis process does produce artifacts and an

informed community. The two properties herein can be exploited to allow non-experts to acquire sufficient domain information to model the problem domain.

2.13 AML Language Design

Hook presents an example of a domain-specific language in [14]. This language was developed for the message translation and validation subdomain of command, control, communications and intelligence systems.

The approach presented in [15] is a mathematically based approach to language design and implementation.

The contribution of this paper is that it explores the systematic production of Domain-Specific Languages by showing that they may be designed in the context of domain engineering activities.

Ladd [16] presents an example of technology that can be used to create code generators for AMLs. It also gives a description of the parsing technology used in the ASPECT toolset to construct abstract parse trees.

The paper describes the motivation for A*, its design and its evolution. A* is based on [2], a language for file processing. It retains the Awk statement language and its interpreter. It also provides a mechanism for replacing the Awk parser with an arbitrary LALR parser. It further provides a new data structure and notation for parse trees, provides a way to describe parse tree traversal and augments the statement language to ease the construction of larger programs. A* has been used both in exploratory research and in production software development within Bell Laboratories.

2.14 The Draco Approach to Constructing Software from Reusable Components

[18] is an early explanation of the idea of domain engineering, including the creation of domain specific languages and an approach to building generators.

Neighbors is more concerned with the reuse of analysis and design information than with reuse of source code. The goal of his work on Draco has been to increase productivity when constructing a family of similar systems. The approach he has taken is to organize reusable software components by domain.

The basis of the Draco work is the use of domain analysis to produce domain languages that may be transformed for optimization purposes. These would be implemented by software components, each of which contains several refinements that represent the different implementations for the object or operation.

An example of how to use the Draco approach to achieve this goal is provided.

Neighbors concludes that: (1) the reuse of analysis and design is much more powerful than the reuse of code; and (2) the software constructed from reusable software components is efficient and provides optimization at the correct level of abstraction. This results in more powerful optimizations than those that are usually available to users of general-purpose languages.

2.15 Designing Software for Ease of Extension and Contraction

Parnas [20] gives a methodology that will help the software engineer build systems that can be easily extended and contracted. There are four parts in this methodology:

- Identifying the subsets. Software engineers must anticipate change before the design is begun.
- Performing an information hiding method. Here one hides the “secrets” which are likely to change, in the future, in separate modules.
- Using a virtual machine approach.
- Designing the “uses” structure.

This methodology provides a way for designing program families that allows one to create new family members either by omitting components from the family design or by adding new components to the family design. A detailed example is given to illustrate these ideas.

2.16 David L. Parnas’ “On the Design and Development of Program Families”

[19] is considered one of the pioneering papers in the design of program families. It compares the “sequential development” approach with the “stepwise refinement” approach for developing a set of similar programs. It also compares the “specification of information hiding modules” method with the “stepwise refinement” approach. Parnas demonstrated that the two methods are based on the same concepts.

This paper focuses on the design stage of program families. Ideas about how the different approaches achieved the design of program families are shown.

2.17 Software Product Line Engineering Process

Tracz [23] introduces a process of product line engineering that crosses several domain areas. It discusses the design of a family of similar systems which is qualitatively different from designing one system at a time.

The process of product line engineering is divided into steps:

The first step is to select the domain of focus, determine domain boundaries, and look at the domain in context.

The second step is to establish domain boundaries and criteria for determining what is in and out of the domain.

The third step is to design the internal structure and conceptual element of the domain.

The last step is to build the asset base architecture and implement assets and the asset base infrastructure.

The contribution of [23] is to present an approach that applies to several domain areas.

Weiss [24] introduces the FAST process -- a family-based software development process.

This book gives the reader a working knowledge of the FAST process. It provides a series of successively more detailed and more formalized descriptions of FAST, interspersed with examples.

This text gives both informal descriptions of each major step in the FAST process and a detailed example of the application of those steps to an example domain. This is known as the Floating Weather Station(FWS) domain.

The main activities of FAST are to:

- Identify the Domain
- Design the Domain
    - Domain Analysis
        - Define the Decision Model
        - Analyse the Commonality
        - Design the Domain
        - Design the Application Modelling Language
        - Create a Standard Application Engineering Process
        - Design the Application Engineering Environment
    - Implement the Domain
        - Implement an Application Engineering Environment
        - Document an Application Engineering Environment
- Develop Application
    - Model the Application
    - Produce the Application
    - Delivery and Operation Support
- Project management

- Change Family

FAST strategies integrate several software engineering ideas and methods. these include the following.

* Predicting expected changes to a system over its lifetime.

* Separating concerns.

* Designing for change using abstraction and information hiding.

* Formal specification and formal modelling.

* Application modelling languages for specifying family members.

* Composing software from adaptable, reusable components.

* Designing process and product concurrently.

* Compiler compilers.

* Template-based reuse.

* Restricting variability to gain efficiency.

In the last part of this text, the formal process model, PASTA, is introduced.

Chapter 3

The FAST Process

The FAST process (Family-oriented Abstraction, Specification, and Translation) is a family-based systematic approach to analyze potential families and to develop facilities and processes for generating family members.

The following description about the FAST process is based on [24].

3.1 Foundations for FAST

3.1.1 Basic Assumptions

When we talk about the FAST process, there are three assumptions that underlie it. Phrased as hypotheses, these assumptions are:

- The Redevelopment Hypothesis: Most software development is the modification of existing software systems. Each modification has more in common with other modifications than it has differences.
- The Oracle Hypothesis: The changes that are likely to be needed to a software system over its lifetime are predictable.
- The Organizational Hypothesis: Both the software and the organization that develops and maintains it can be organized in such a way as to take advantage of predicted changes. That is, the software and its developers may be organized so that a change of any predicted type can be made independently of changes to other types. This means that making such a change requires modifying, at most, a few modules in the system.

Together these hypotheses suggest that, whenever possible, one should make a family the unit of software development where the set of programs that constitute the family are identified by defining a set of commonalities and variabilities that determine what is in the family and what is not.

3.1.2 FAST Strategies

In order to make families the units of software development we have to adopt strategies that take advantage of the familys commonalities and variabilities. FAST uses the following strategies for creating software families:

- To identify collections of programs that may be considered as family members. This is where a family is a set of programs that have enough in common for it to be worthwhile to base the production of family members on a set of common assets.
- To design the family for producibility, i.e., create a common design for all family members. At the same time, should be designed a process for producing family members concurrently. The idea is to make it easy to produce a family member by following the process for applying the common design.
- Invest in family-specific tools to make the production of family members rapid. Such tools are the compiler for the AML language, analysis tools and editors, etc.
- For each family create a way to model family members.

FIGURE 3-1. shows the results of integrating these strategies to create a domain and produce domain members.

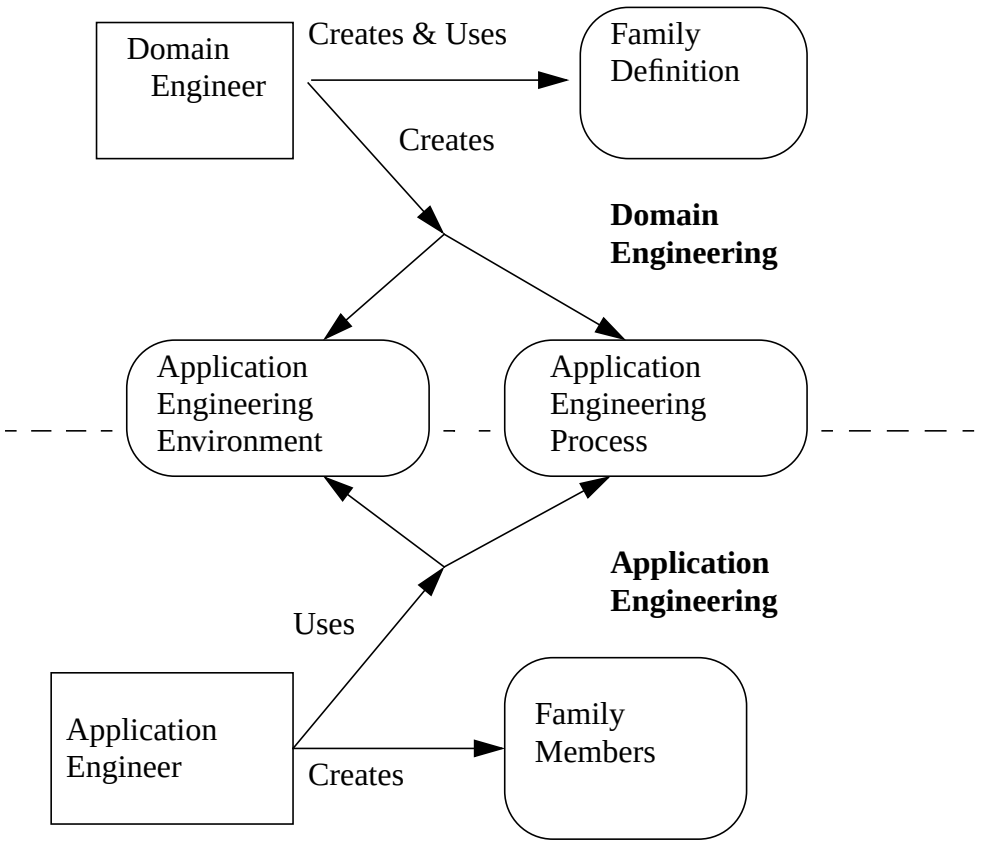


FIGURE 3-1. Outputs From Domain Engineering and Application Engineering

The FAST process is the integration of a variety of ideas and methods in the software engineering field. Those ideas and methods make it feasible to adopt FAST strategies. The most important of them include the following:

- Abstraction.
- Information hiding.
- Separation of concerns.
- Predicting expected changes to a system over its lifetime.

- Designing for change.

The FAST process consists of two subprocesses connected by feedback loops, as shown in Figure 3-2. These two subprocesses are domain engineering and application engineering.

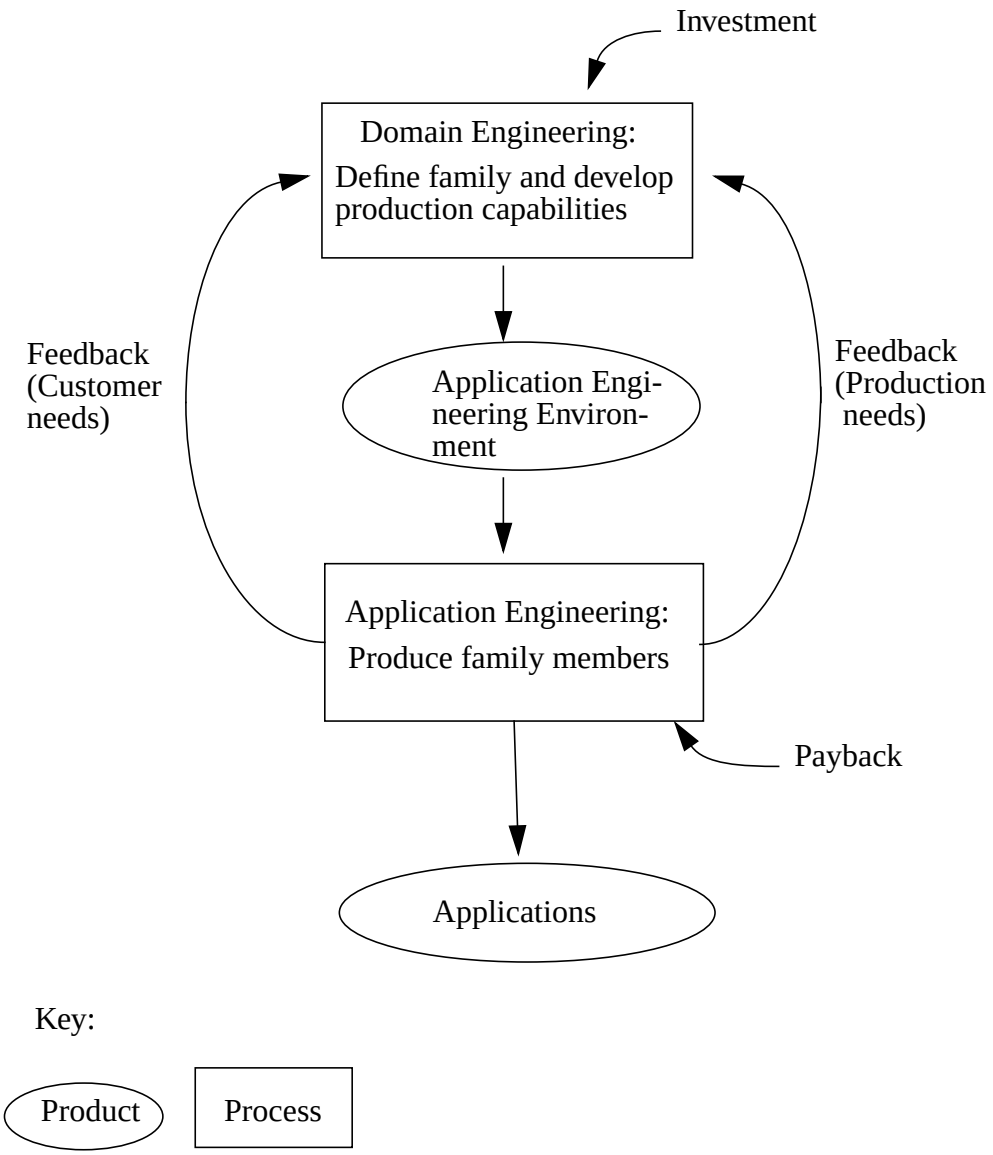


FIGURE 3-2. The FAST Process Paradigm

3.2 Domain Engineering

The purpose of domain engineering is to make it possible to generate program family members automatically. It is an investment in tools and processes. Domain engineering is a repetitive process; its result is the creation or refinement of an application engineering environment consisting of the AML, tools for the language (e.g., the compiler) and an application engineering process for using the environment. To do this, domain engineers must accomplish the following:

- Defining the family.
- Developing a language to specify family members (the application modeling language).
- Developing an environment for generating family members from their specifications (the application engineering environment).
- Defining a process for producing family members using the environment (the application engineering process).

Domain engineering consists of two subprocesses: analyzing the domain and implementing the domain, as shown in Figure 3-3.

3.2.1 Domain Analysis

3.2.1.1 Economic analysis of the domain

An economic analysis of the domain is an estimation or a re-estimation the economic value of applying the FAST process. Is it worth it to apply FAST process to the domain or to decide whether there are sufficient potential family members to justify the investment in domain engineering? Generally speaking, the more members of the family, the more suitable it is to apply FAST on that domain.

3.2.1.2 Define the family

Defining the family means identifying potential family members and characterizing what they have in common and how they differ. We call this a commonality analysis. The results are what is called a commonality analysis.

The standard form for a FAST commonality analysis includes the following sections:

- Introduction: Describes the purpose of the commonality analysis.

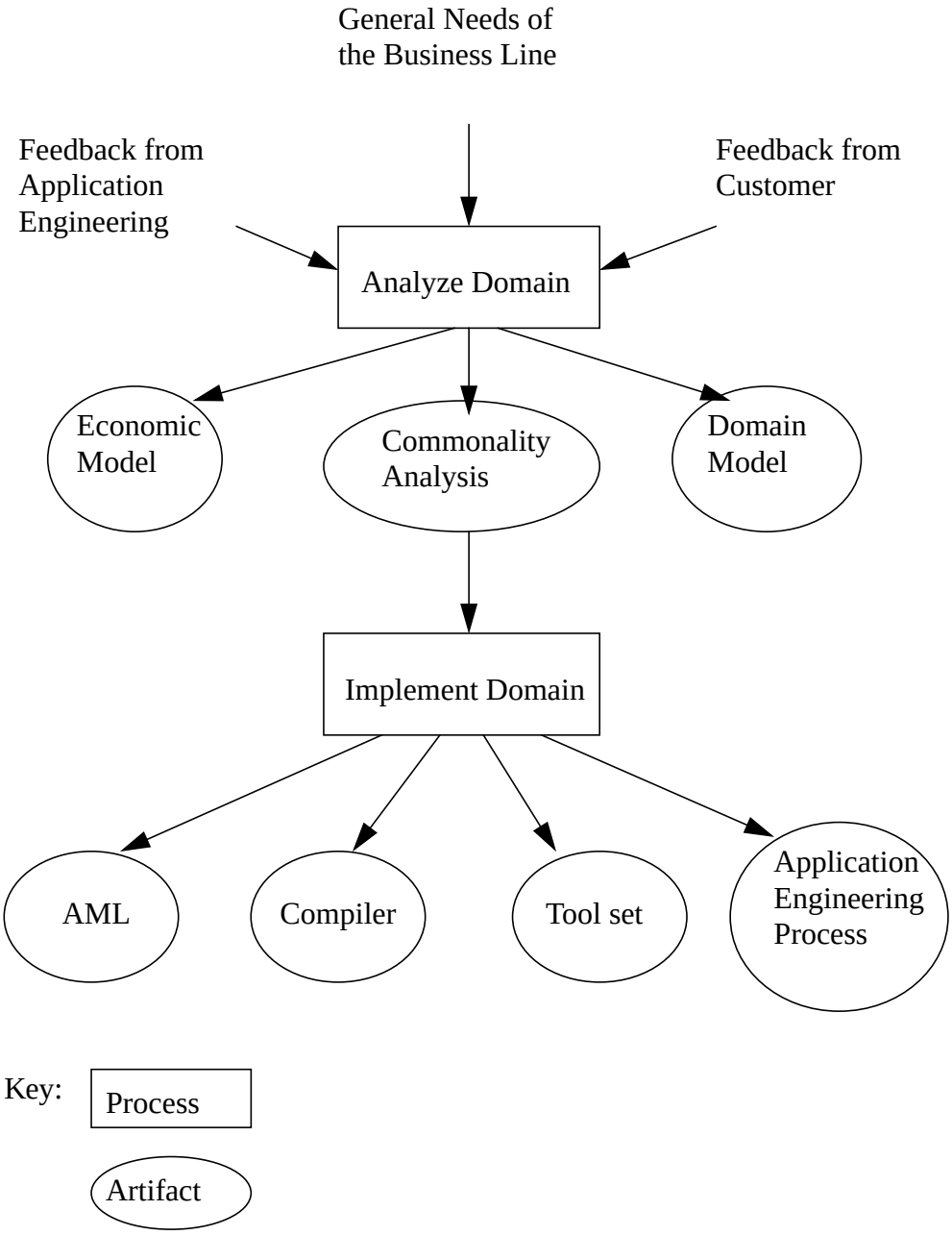


FIGURE 3-3. Ideal Domain Engineering Process

- Overview: Gives a brief overview of the domain.

- Dictionary: Defines technical terms for the domain that are used in the commonality analysis.
- Commonalities: Lists assumptions that are true for each member of the domain.
- Variabilities: Lists assumptions about how members of the domain may vary.
- Parameters of Variation: Specifies the value space for each variability and the time at which the value of the variability must be fixed.
- Issues: Describes issues that arose during the analysis and how they were resolved.

3.2.1.3 Design the Application Engineering Environment

Commonality analysis is used to guide the design of the AML. The AML is used by the application engineers to specify different family members. The commonalities of the family are built in to the language. The well-designed AML is a key part of the application engineering environment. By using this language the application engineers should be able to specify particular family members just by specifying the variations considered during the definition of the family.

Other parts of the application environment are the compiler and the analysis tools for the AML. Such tools may be designed to appear to the application engineer as an integral part of the compiler.

3.2.2 Domain Implementation

3.2.2.1 Define the Application Engineering Process

The application engineering process is designed for generating family members automatically.

The ideal application engineering process includes the following steps:

1. The customer provides the requirements for the application.
2. The application engineer presents the requirements for the application as an application model and writes codes using the AML to specify the requirements.
3. The application engineer analyzes and refines the model until he/she is satisfied that it meets the customer's requirements. He/she may then generate a deliverable set of code and documentation from the model.

4. The customer inspects the application as the application engineer has modelled it. This is done either by viewing the results of analyses or by testing the application as generated from the model.
5. The customer either accepts the generated application or returns to step 1.

3.2.2.2 Design Implementation

Based on the design document, the steps in this phase are to:

1. Create the AML for that domain.
2. Create the compiler for the AML.
3. Create analysis tools.

3.2.3 Artifacts of Domain Engineering

After the implementation of the domain, we get the following artifacts:

1. An economic model of the domain.
2. A commonality analysis.
3. A decision model, a description of the decisions that must be made, and the order in which they are made to produce an application.
4. The AML.
5. The compiler for the AML.
6. The tool set that forms the application engineering environment.
7. A description of the application engineering process used to model and generate applications.

3.3 Application Engineering

The application engineering process is the process that uses the products of domain engineering to produce new family members that satisfy customer requirements. The intent of the application engineering process is for the application engineer to use requirements facilities without having to involve the concerns of designing and coding. Freed from this the engineer can better interact with customers. The ideal application engineering process is shown as Figure 3-4.

The output of the application engineering is a model of the application created by the application engineer in AML and deliverable code for the application.

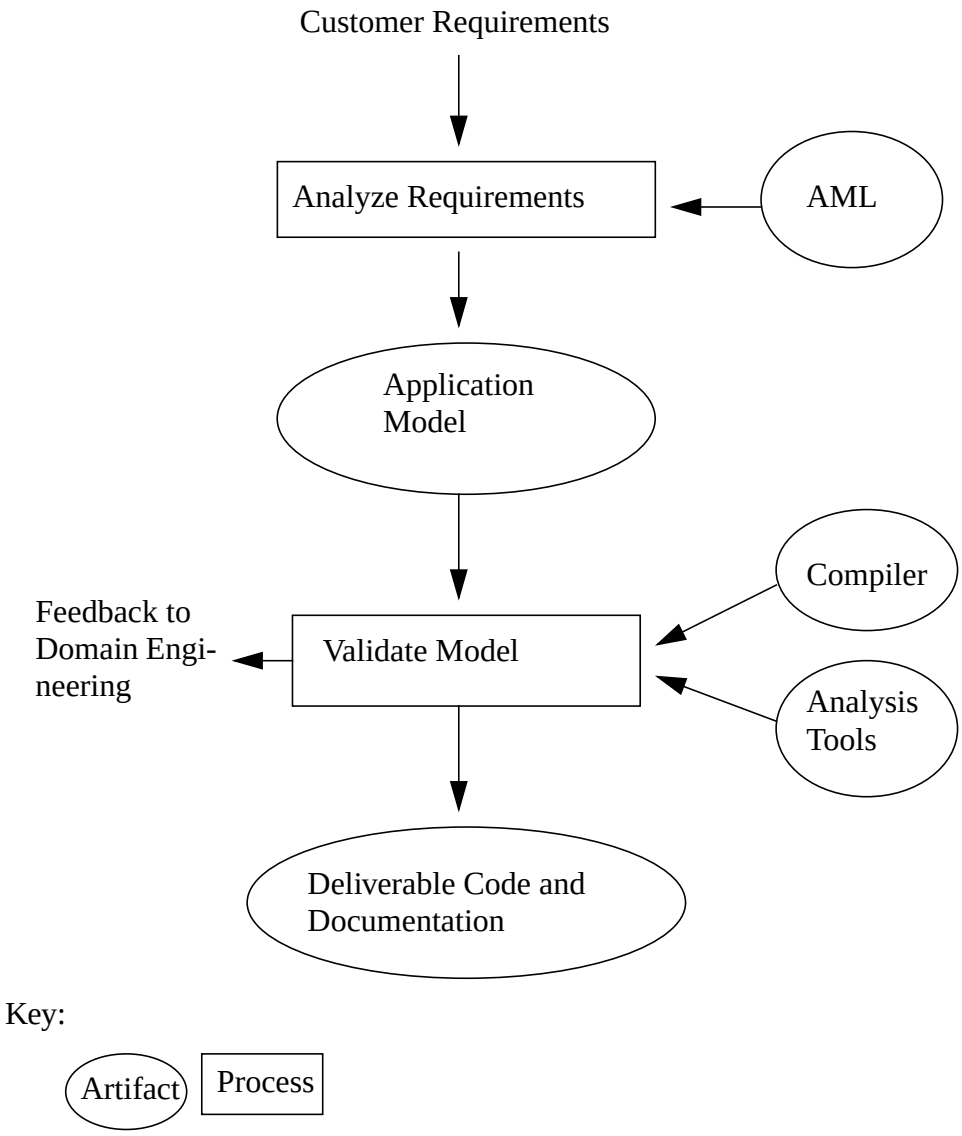


FIGURE 3-4. Application Engineering Process

3.4 The Economics of FAST

The basic economic assumption underlying FAST is that investment in domain engineering will be paid back by more efficient production of family members. We start by distinguishing between two cases, one where you pay little or no attention to domain engineering, and one where you design the domain with the intent of making production of family members more efficiently.

3.4.1 Case 1: No Domain Engineering

Assume that the cost of producing a new member of the family without domain engineering is approximately constant (denoted by CT). This is a typical case. The cost for producing N family members then is N*CT.

3.4.2 Case 2: Domain Engineering

Assume that the cost of domain engineering is I, representing the investment in producing family members more efficiently. Further, assume that a result of this investment is that the cost of producing a family member is CF. The cost for producing N family members then is I + N*CF. If the domain engineering is successful, it should be true that CF<CT. To pay back the investment in domain engineering N must be large enough so that I<N*(CT-CF).

3.5 Benefits of FAST

Family-based processes such as FAST, where the knowledge needed to produce family members rapidly is packaged for widespread and efficient use and reuse without abuse, are worth creating because successful software systems inevitably evolve into families. FAST provides the following benefits in this evolution:

- More efficient production of family members.
- The ability to support a greater variety of family members.
- Graceful aging.
- Improved quality. Improvements in common designs and code lead to improvements in all family members, thereby leveraging quality improvements.

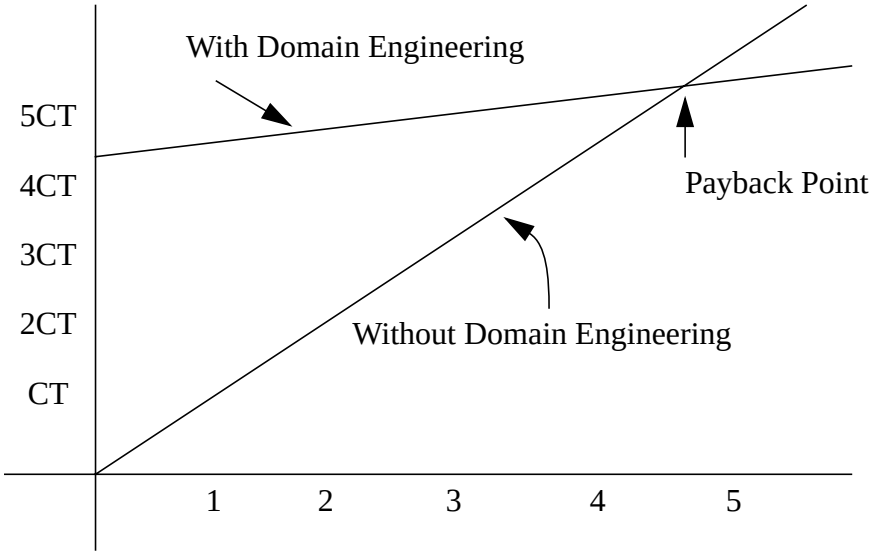


FIGURE 3-5. Cost of Producing N Family Members

- Competitive advantage in domains of application. Organizations that adopt the family approach will have a competitive advantage in the domains where they apply it. They may adopt a business strategy of developing a family for the purpose of rapidly producing many variations on a product at low cost with high quality.

Chapter 4

Introduction to the TTS System

The TTS (Table Tool System) is a project of the Software Engineering Research Group at McMaster University. It is an integrated set of tools which manipulate multi-dimensional tabular expressions. The manipulation includes entering, formatting and transforming tabular expressions. This tabular representation of mathematical expressions improves the readability of complex design documentation.

The TTS aims to automate checking of software specification and design documents and to provide aid during software testing and maintenance. Abstract communication interfaces make it possible to integrate new tools without knowledge of those already existing.The TTS Developer's Guide [22] describes the TTS in reasonable detail.

The TTS is decomposed into three tiers. These are: kernel, infrastructure and applications. The following description is partially taken from [22].

4.1 Kernel

The kernel module hides the data structure representing expressions and algorithms for manipulation. A list of terms used with these modules is given as follows.

Expn

An Expn is a data object that represents a mathematical expression. An expression can be viewed as a prefix expression tree where the nodes of the tree are symbols. Symbols in an expression are identified by their ids. For example, a simple application "x + y" can be represented as follows if the id of "+" is 1, id of "x" is 111 and the id of "y" is 112. Expressions are grouped into three categories: atom (constants or variables), application (use of functions or predicates with one or more arguments), and tables.

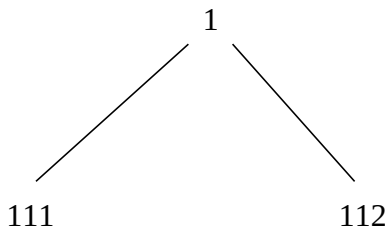


FIGURE 4-1. Expression “x + y” Represented in TTS

Index

An index is a data object that consists of a grid number and also a sequence nembers to identify a cell within a table (e.g., the row number and column number of the cell).

Path

A path is a sequence of objects, each of which is either an Index or an integer. A path is used to represent the position of a sub-expression within an expression. Integers are used to specify a particular argument of an application. Index objects specify a particular element of a table.

Shape

A shape is a data object that represents the shape of a table, the number of grids, their dimensionality and the length of each dimension. Shapes exist so that a user can separate the task of describing the shape of a table from the action of creating the table.

SymTbl

A SymTbl is a data object that represents a symbol table. This SymTbl table describes a collection of symbols. An id is used to associate a symbol and a node of the Expn data object. The information about a symbol is organised into a set of information classes. An information class is referred to by its name. Some classes are shown below.

- The Name of a symbol is a string. For example, “2”, or “y”. Names do not necessarily distinguish symbols. Since an expression represented in symbol ids is not easy to understand, symbol names will be used instead of their ids to represent the expression.
- A tag class is an integer. It indicates if a symbol is a constant (ConsTag), a predicate constant (PConstTag), a variable (VarTag), a function application (FATag), a logical expression (LETag), a quantified logical expression (QLETag), a predicate expression (PETag), a function table (FTableTag) or a predicate table (PdTableTag).
- An Arity class contains an integer describing the arity of a function symbol.
- A CType class contains a string describing the C data type for a variable symbol.

4.2 Infrastructure

The TTS infrastructure module hides data structures and algorithms that enable users to manipulate TTS objects. The modules in the infrastructure provide facilities to develop TTS applications.

The infrastructure contains three modules: Tools, Utilities and the Tool Integration Framework.

4.2.1 Tools

The Tool module provides some primitive services that operate on TTS objects, such as Expn and SymTbl. The tool module includes the following tools:

4.2.1.1 Context Manager

A context is an ordered sequence of named expressions together with a symbol table containing information about the symbols in the expressions. Since all of the expressions use the same symbol table, ids will have consistent interpretation within the context. This simplifies manipulation and interpretation of expressions. The CM (context manager) tool manipulates expressions in a context. For example, it can load a context from a file or save it in a file.

4.2.1.2 Table Construction Tool

The TCT (Table Construction Tool) allows the user to construct and edit an expression by building it up from smaller expressions. It uses the information module to retrieve symbols and the TH (Table Holder) to retrieve the expression.

4.2.1.3 Table Formatting/Printing Tool

The TPT (Table Printing Tool) prints and displays expressions. It allows the user to adjust such things as the print size of parts of expressions and the width and height of table rows or columns. it does not allow any modifications to the contents of the expression or changes to the symbol information that may change their interpretation. This tool produces a Postscript representation of the expression which is suitable for display or printing.

4.2.1.4 Symbol Editor

The Symbol Editor is a tool for editing or viewing the information associated with symbols stored in the Information module. It allows a user to modify the set of symbols available for use in expressions. New symbols may be created and assigned presentation and semantics information.

4.2.1.5 Inversion/Normalization Tool

The Table Invertor tool allows users to transform tabular expressions into different forms. For example, a normal table may be inverted, an inverted table may be normalized, and the headers of a table may be combined.

4.2.1.6 Specialization and Simplification Tool

The Specialization and Simplification Tool is used for simplifying tabular expressions by taking into account user-supplied constraints on the variables that appear in the expression.

4.2.1.7 Carving and Slicing Tool

The Carving/Slicing tool allows users to extract portions of a tabular expression. For example, one or more rows may be carved from a multi-row table, or a three-dimensional table may be sliced in order to obtain a two-dimensional table.

4.2.1.8 Code Generator

The Code Generator tool allows the user to generate C++ code for all of the expressions in a context. It generates code to evaluate the tabular expressions. The Code Generator takes a context and a C header file as its input and produces a C++ file and a C++ header file as its output. The input C header file contains data structure definitions. Note that there are

some restrictions on the syntax of expressions in the context supplied to the Code Generator:

- The root symbol of every expression must be “defined”.
- The left branch of the root must be a function with n arguments, where n > 0.
- The right branch of the root is a function definition. Every variable in the expression, except index variables, must be one of the arguments of the function specified by the left branch of the root.
- All symbols (variables and functions) must have information assigned in the CType information class.

4.2.1.9 Composition Tool

The Composition Tool computes the functional composition of two tabular or non-tabular expressions.

4.2.1.10 Transformation Tools

The transformation Tool transforms generalized decision tables to structured decision tables and vice versa.

4.2.1.11 Table LaTeX Tool and Document Indexing Tool

The Table LaTeX Tool (TLT) produces a LaTeX representation of expressions and the Document Indexing Tool (DIT) automatically generates a set of indices for expressions. This indicates the table, row, column and page number where each symbol appears.

4.2.1.12 Table Checking Tool

The Table Checking Tool performs completeness and disjointedness verification of table conditions.

4.2.1.13 Alias Table Tool

The Alias Table Tool works as a preprocessor for the Code Generator Tool. It uses the Code Generator to evaluate aliases based on their definitions and assignments to the variables contained in the alias definition expressions. The Alias Table Tool replaces aliases contained in the tabular expressions with the evaluation results. The Alias Table Tool improves the readability of tabular expressions used in computer system descriptions.

4.2.1.14 Type Checking Tool

This tool is under developed. One of the Master student in the MCSERG group is working on it.

4.2.2 Utilities

Utilities are modules that implement algorithms which may be useful to more than one tool or application but cannot be invoked directly by a user. There are Kernel Utilities and General Utilities. Kernel Utilities are utilities that make use of the TTS kernel. General utilities are those utilities that do not make use of the TTS kernel.

4.2.3 Tool Integration Framework

The Tool Integration Framework (TIF) provides a means for tools to interact with each other. In this case the user works without designers needing to know anything about other tools in the TTS. It provides a means of passing objects between tools as well as a method of informing tools of significant events.

4.3 Applications

This layer contains programs that combine infrastructure modules for the manipulation or interpretion groups of tabular expressions: it creates them as documents which describe some aspect of the computer system. The Test Oracle Generator is an application of TTS.

Chapter 5

FAST Applied to an Airline Upgrade Policy Family

This chapter describes how to enhance the FAST process using tabular notation and table-based tools. I will apply the enhanced FAST approach to an Airline Upgrade Policy family. Analysis of the family is provided.

5.1 Enhancing the FAST process using tabular notation and table-based tools

Lucent/Bell Labs have had success in reducing programming costs and speeding up development times by using the FAST process. In the FAST process, after identifying the commonalities and parameters of variation in family of programs, a special purpose programming language is developed. This language is suitable for writing members of the family. The common features of the family are already built-in, only the differences need to be programmed.

McMaster University's Software Engineering Research Group is developing a set of tools to support the use of a broad class of tabular expressions(TTS). Using tabular expressions one may present a complex conditional expression in a way that is more easily read, more easily analyzed and more likely to be correct than either conventional mathematics or conventional programs.

I propose to combine these two technologies. The results of the commonality analysis process, as developed at Lucent, will be used to develop a set of incomplete tabular expressions. The completed parts will represent the commonalities. The incomplete parts will correspond to the parameters of variation. The application engineer must complete the tables. Only then can the Code Generator tool of TTS be used to generate a prototype family member.

Figure 5-1 shows the combination of the FAST approach with in a TTS system. After the commonality analysis, instead of the implementation of the AML(Application Modelling

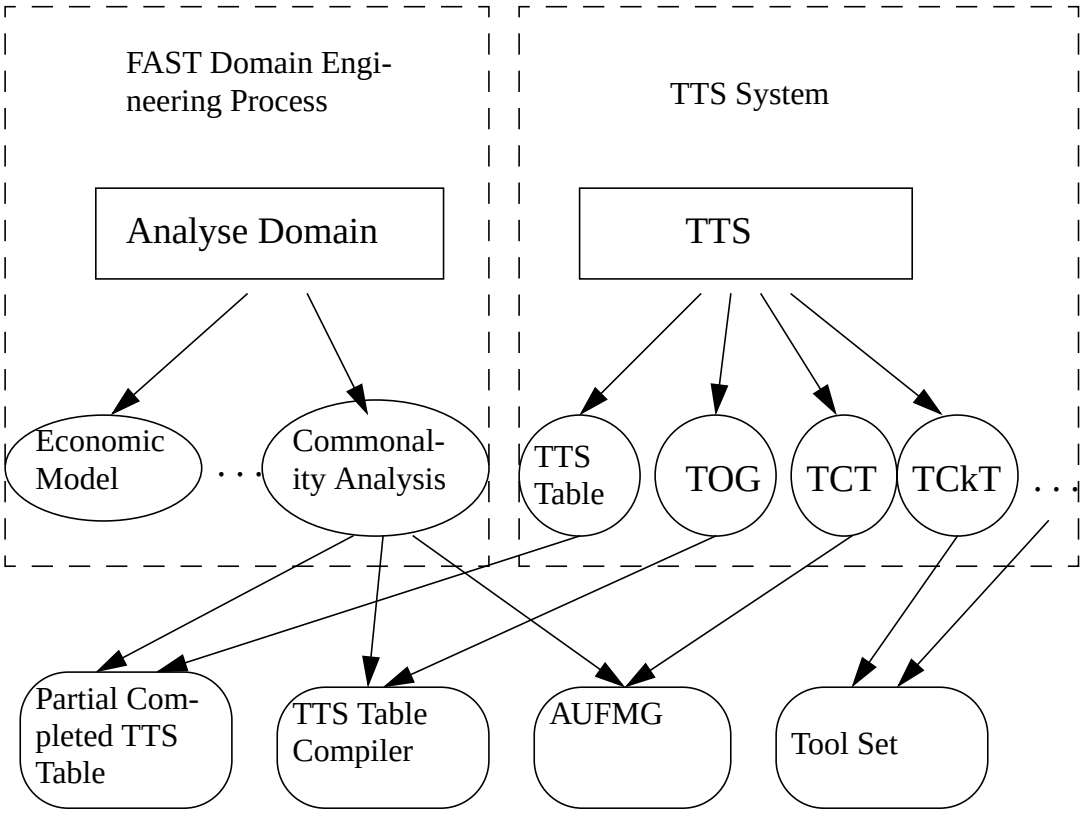


Figure 5-1 FAST Approach Combined with TTS System

Language) and the compiler for the AML, we edited the TTS table to form a partially completed TTS table. The completed part represents the commonalities of all the family members, the rest is left for the application engineer to fill in. This represents the variabilities of the family.

We did not need to implement the compiler because we had the CG tool of TTS to act in this manner.

Based on the commonality analysis we needed to modify the TCT tool which acts as an editor for the TTS table. After the modification no one would be able to modify the fixed part of the table; only the incomplete part could be filled in or modified.

We also have some tools available for analytical purposes.

The advantages of the combination are:

1. For the domain engineers. They do not need to implement a compiler for the AML, which is critical and time consuming. They only need to put the commonalities in the TTS table using the TTS system which is efficient and meets demands.

2. For the application engineers. They do not need to write lines of code using AML; they only need to complete the TTS table.

5.2 Definition of Terms

In the rest of this chapter contains three examples of program families. Each of three examples concern an airline upgrade policy. An airline upgrade policy family is the set of policies used for airlines to decide who will move when there are extra seats available in first class. In this section we will describe the definition of terms used in the examples. The following apply to all three:

EmpSeatsUpfront: No. of empty unreserved seats upfront.Type: Integer.

EmpReUpfront: No. of empty reserved seats upfront.Type: Integer.

EmpSeatsCoach: No. of empty unreserved seats in coach. Type: Integer.

coupon: Passenger who has an upgrade coupon. There are five types of coupon: SuperE-lite, System, NorthAmerica, Prestige, None.

price: Classes of the ticket price. There are three types of price: FullFare, HalfDiscount and Discount.

status: Frequent-traveler status of the passenger. There are four types of status: 1K, EXE, Premier and None.

weight: Weight of the passenger(kg). Its type is double.

age: Age of the passenger(year). Its type is integer.

The purpose of the analysis of the family is to provide:

- A way to specify a particular member of airline upgrade policy family.
- A way to generate the software for the family members.

Below, I will provide three examples of airline upgrade policy families. For each example I will describe the commonalities and variabilities of the family. I will also provide a member of the family as example.

5.3 Commonality Analysis of Example 1

5.3.1 Commonalities

The following statements are basic assumptions about the way that airline upgrade policies are described:

1. They are all specified by tabular expressions that were developed by McMaster University's Software Engineering Research Group.
2. Each table is proper, that is, all the elements in both rows and columns of a table must be mutually exclusive.
3. Each table contains at least 1 row and 1 column.
4. The value of the function must be Yes, No or Wait.
5. If EmpSeatsUpfront is less than or equal to 0, the value of the function could only be No or Wait; cannot be Yes.
6. All family members are characterized by 2 parameters price and EmpSeatsUpfront.

TABLE 1. Factors of Commonality

Factor	Meaning	Type	Default
EmpSeatsUpfront	No. of empty unreserved seats up front	Integer	0
price	classes of the ticket price	{FullFare, HalfDiscount, Discount}	Discount

7. For the first example, the values of the 2 parameters are also fixed. Namely, for price, it is FullFare, for EmpseatsUpfront, it is 0.

5.3.2 Variabilities

The following statements describe how they may vary:

1. The possible values of the policy function may vary.

Based on the commonalities and the variabilities, we constructed the following partially completed TTS table:

TABLE 2. Table for Airline1 Family

	price=FullFare	$price = \neg FullFare$
EmpSeatsUpfront>0	?	?
EmpSeatsUpfront<=0	?	?

5.3.3 An example of the family member

An example member of this Family could be:

TABLE 3. a member of Airline1 Family

	price=FullFare	$price = \neg FullFare$
EmpSeatsUpfront>0	Yes	No
EmpSeatsUpfront<=0	Wait	No

5.3.4 Issues

After the analysis of this family, we had to consider the following issues:

1. Should we add additional factors to the system?

Answer1: Yes, this would broaden the appeal of the system to a much larger customer set.

Answer2: No, this would complicate the software and make it more difficult to achieve our objectives. We have no customers asking for this capability now.

Resolution: We will stay with the factors we have for now but keep in mind that we may need to change our design to accommodate additional factors.

5.4 Commonality Analysis of Example2

5.4.1 Commonalities

The following statements are basic assumptions about the airline upgrade policy domain for example2; they are true of all family members.

1. They are all specified by tabular expressions that developed by McMaster University's Software Engineering Research Group.
2. Each table must be proper, that is, all the elements in both the rows and columns of a table must be mutually exclusive.
3. Each table must contain at least 1 row and 1 column.

4. The value of the function must be Yes, No or Wait.
5. If EmpSeatsUpfront is less than or equal to 0, the value of the function could only be No or Wait, it cannot be Yes.
6. They all use 2 parameters to specify their conditions.
7. The 2 parameters are price and EmpSeatsUpfront.

TABLE 4. Factors of Commonality

Factor	Meaning	Type	Default
EmpSeatsUpfront	No. of empty unreserved seats up front	Integer	0
price	classes of the ticket price	{FullFare, HalfDiscount, Discount}	Discount

5.4.2 Variabilities

The following statements describe how they may vary:

1. The value of the function may vary.
2. The values of the parameters may vary.

Based on the commonalities and the variabilities, we constructed the following partially completed TTS table:

TABLE 5. Table for Airline2 Family

	price= ?	*price* = ¬
EmpSeatsUpfront> ?	?	?
EmpSeatsUpfront<= ?	?	?

5.4.3 An example of the family member

An example member of this Family could be:

TABLE 6. a member of Airline2 Family

	price=Discounted	*price* = ¬*Discounte*
EmpSeatsUpfront>5	No	Yes
EmpSeatsUpfront<=5	No	Wait

5.4.4 Issues

After the analysis of this family, we have to consider the following issues:

1. Should we add additional factors to the system?

Answer1: Yes, this would broaden the appeal of the system to a much larger customer set.

Answer2: No, we have no customers asking for this capability now.

Resolution: We will stay with the factors we have for now but keep in mind that we may need to change our design to accommodate additional factors.

5.5 Commonality Analysis of Example 3

5.5.1 Commonalities

The following statements are basic assumptions about the airline upgrade policy domain, they are true of all family members:

1. They are all specified by tabular expressions that were developed by McMaster University's Software Engineering Research Group.
2. Each table must be proper, that is, all the elements in both the rows and columns of a table must be mutually exclusive.
3. Each table must contain at least 1 row and 1 column.
4. The value of the function must be Yes, No or Wait.
5. If EmpSeatsUpfront is less than or equal to 0, the value of the function could only be No or Wait, can not be Yes.
6. They all use certain limited factors to set up their conditions. Those factors are listed below:

TABLE 7. Factors of Commonality

Factor	Meaning	Type	Default
EmpSeatsUpfront	No. of empty unreserved seats up front	Integer	0
EmpReUpfront	No. of empty reserved seats up front	Integer	0
EmpSeatsCoach	No. of empty unreserved seats in coach	Integer	0

TABLE 7. Factors of Commonality

EmpReCoach	No. of empty reserved seats in coach	Integer	0
status	frequent-traveler status of the passenger	{1K, EXE, Premier, None}	None
coupon	the passenger who has upgrade coupon	{SuperElite, System, North-America, Prestige, None}	None
price	classes of the ticket price	{FullFare, HalfDiscount, Discount}	Discount
age	age of the passenger, (year)	Integer	30
weight	weight of the passenger, (kg)	Double	100
stand-by	whether the passenger is standing by or not	Boolean	False
Rtime	time of reservation	Date/Time	02/17/00/ 00:00:00
Ctime	time of checking in	Date/Time	02/17/00/ 00:00:00

5.5.2 Parameters of Variability

The following statements describe how they may vary:

1. The number of factors used may vary.
2. The factors used may vary.
3. The possible values of the factors may vary.

In this big family the application engineer should determine the shape of the table, what variables are used and the possible values of the variables.

5.5.3 An example member of this Family

	$coupon = SuperElite$	$(status = 1K) \wedge (coupon = \neg SuperElite)$	$(status = \neg 1K) \wedge (coupon = \neg SuperElite)$
$EmpSeatUpfront > 0$	Yes	Yes	No

$EmpSeatsUpfront \leq 0 \wedge EmpReUpfront > 0$	Wait	Wait	No
$EmpSeatsUpfront \leq 0 \wedge EmpReUpfront \leq 0$	No	No	No

5.5.4 Issues

After the analysis of this family, we have to consider the following issues:

1. Should we add additional factors to the system?

Answer1: Yes, this would broaden the appeal of the system to a much larger customer set.

Answer2: We have no customers asking for this capability now.

Resolution: We will stay with the factors we have for now but keep in mind that we may need to change our design to accommodate additional factors.

Chapter 6

FAST Applied to a Mobile Phone Plan Family

6.1 Mobile Phone Plan Commonality Analysis

6.1.1 Introduction

The mobile phone plan family is a family of plans for mobile phone customers to choose from. For example, one of the plans may be: monthly rate $30, this includes 200 free minutes, free long distance call and pay full price for cell phone.

The the purposes of this analysis is to provide:

- A way to specify a particular member of mobile phone plan family.
- A way to generate the software for the family members.

6.1.2 Commonality

For purpose of illustration we assume that the following statements are basic assumptions about the mobile phone plan domain, they are true of all family members.

1. They all use the formula to calculate the monthly rate:

	FreeWeekend = true	**FreeWeekend = false**
Contract <= 0	$Baserate + (Freemin)/4 + (DiscofPhone)/20 + 10/(CostPerMin) + 20/(CostofLongD)$	$Baserate + (Freemin)/4 + 10/(CostPerMin) + 20/(CostofLongD)$
Contract > 0	$Baserate + (DiscofPhone)/20 + 10/(CostPerMin) + 20/(CostofLongD)$	$Baserate + 10/(CostPerMin) + 20/(CostofLongD)$

Table 6-1 Monthly rate table

2. The factors that they use are all selected from the following list:

Factor	Meaning	Type	Default
MonthlyRate	How much is the plan per month (Dollar)?	Real	0
CostPerMin	How much is it for a minute (Cent)?	Real	0
BaseRate	How much is it for the basic rate per month (Dollar)?	Real	0
CostOfLonD	How much is it for long distance call (Cent)?	Real	0
DiscOfPhone	How much is the cell phone discounted (Dollar)?	Real	0
FreeMin	How many free minutes for that plan (Minute)?	Integer	0
Contract	How long is the contract? (year)	Integer	0
FreeWeekend	Is it free during weekend and weeknight?	{Yes, No}	No

Table 6-2 Factors of Variability table

6.1.3 Parameters of Variability

The following statements describe how they may vary:

1. The number of factors they actually use may vary.
2. The factors that they use may vary..

6.1.4 Issues

1. Should we add additional factors to the system?

Answer1: Yes, this would broaden the appeal of the system to a much larger customer set.

Answer2: No, this is enough to illustrate the ideas.

Resolution: We will stay with the factors we have for now but keep in mind that we may need to change our design to accommodate additional factors.

6.2 An Example of the Mobile Phone Plan Family

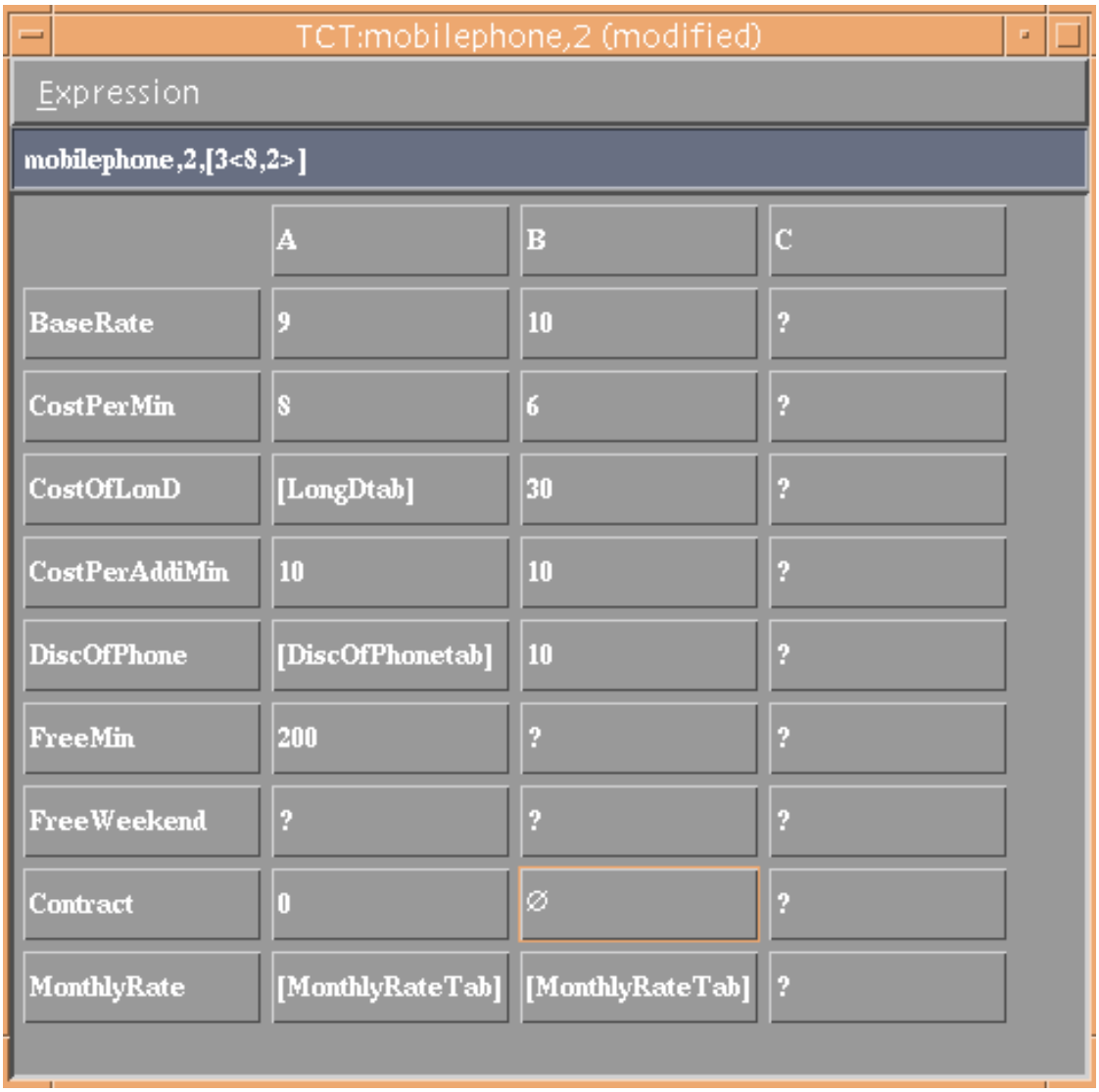
Here is an example family of the mobile phone plan:

Plan A is a small family, it only allows the user to select if he/she wants a Free Weekend or not.

Plan B is a larger family than plan A. It allows the user to select the amount of free minutes, if the plan contains free weekends or not and if the user wants a contract.

Plan C is the largest family, it allows the user to define which factors to use and the values of the factors.

Note: The user of this tool is not the customer but the sales representive.



	A	B	C
BaseRate	9	10	?
CostPerMin	8	6	?
CostOfLonD	[LongDtab]	30	?
CostPerAddiMin	10	10	?
DiscOfPhone	[DiscOfPhonetab]	10	?
FreeMin	200	?	?
FreeWeekend	?	?	?
Contract	0	∅	?
MonthlyRate	[MonthlyRateTab]	[MonthlyRateTab]	?

Figure 6-1 Mobile Phone plan Family Main Window

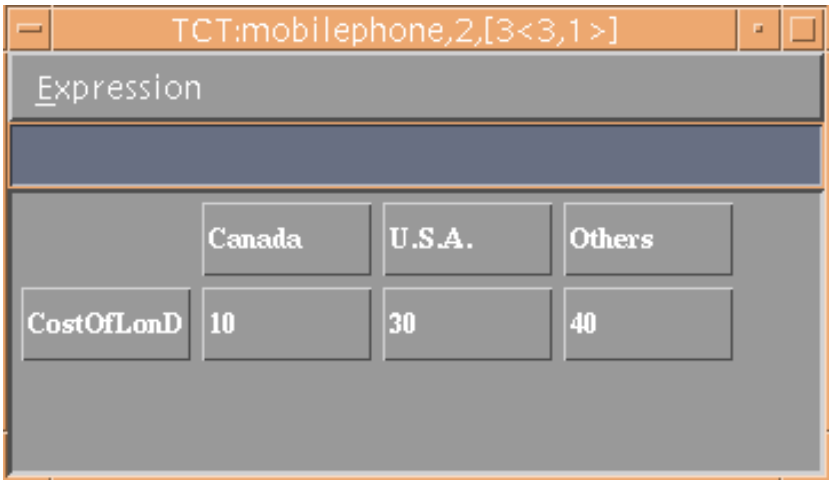


Figure 6-2 Cost of Long Distance Call Window

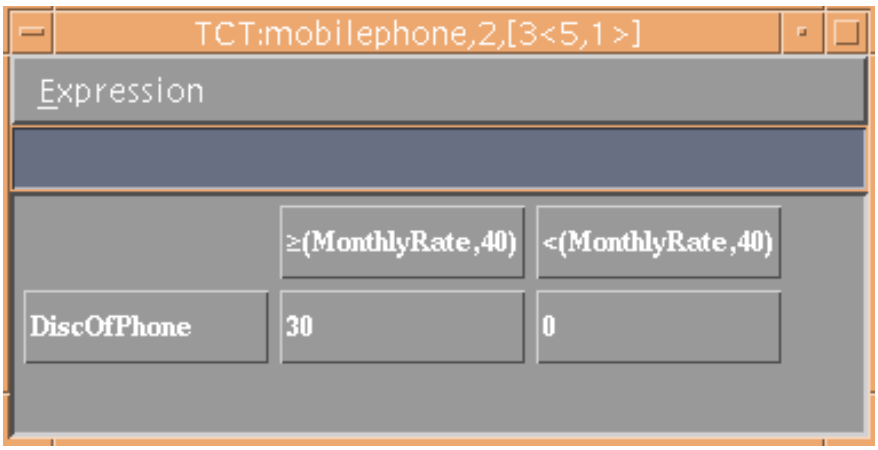


Figure 6-3 Discount of Phone Window

Chapter 7

Airline Seats Upgrade Policy Family Member Generator (AUFMG) Tool Design

This chapter discusses the design of the AUFMG tool. It includes the module decomposition, the access programs in each module, the modules use relation and the user guide for the AUFMG tool.

During the implementation of the AUFMG tool we used some data types from the TTS Kernel. These include Expn, Path, SymTbl, Inst_ptr, Id, Exp_struct_ptr, etc. For more details about the data types refer to the TTS Developer Guide[22].

7.1 Definition of Terms used in this Chapter

- **Instance:** An instance is a tabular expression. It is composed of grids and cells. Each cell in a tabular instance can be edited as a scalar expression (This is a restriction. In TTS a table cell can contain another table). The instance data structure contains all the information that you need to know about the tabular expression, for example, the shape of the table, the path, and so on.
- **Node:** A node is a cell in an instance. It contains a variety of information with regard to a component in a predicate expression. For example, the string that will display in the cell, the information of its left cell and right cell. Nodes are used to represent the contents of the Table Holder in a way that can be easily displayed to the user.
- **Node offset:** Node offset is the location of the symbol in a node structure. It indicates how many characters from the left of the user display this node begins at.

7.2 Module Decomposition

To make the design easy to understand and change the AUFMG tool is decomposed into a set of modules using the “information hiding” principle [21].

7.2.1 Instance Module (inst.c)

This module contains programs that deal with AUFMG's instance structure. The secret of this module is the data structure representing one AUFMG instance.

Access Program	Description
bool main_CheckInstModStat (Inst_ptr)	Checks to see if there are any unapplied changes in the cell editor that need to be applied to the instance.
bool main_InitInstanceStruce (Inst_ptr)	Initializes all member values and/or structure pointers to default values.
void main_SetInstanceName (Inst_ptr, char *)	Changes the name of the instance.
void main_FreeInstance (Inst_ptr)	Frees all memory associated with this instance.
Path GetInstPath (Inst_ptr)	Get the path of the instance.
void SetInstPath (Inst_ptr, Path)	Set the path of the instance.
Expn GetInstExpn (Inst_ptr)	Get the Expn of the instance.
void SetInstExpn (Inst_ptr, Expn)	Set the Expn of the instance.
SymTbl GetInstSym (Inst_ptr)	Get the SymTbl of the instance.
void SetInstSym (Inst_ptr, SymTbl)	Set the SymTbl of the instance.
Shape GetInstShape (Inst_ptr)	Get the shape of the instance.
void SetInstShape (Inst_ptr)	Set the shape of the instance.

7.2.2 Node Utilities Module (nodeutils.c)

The Node Utilities Module is used to generate the node structure of the AUFMG. The secret of this module is the data structure representing nodes.

Access Program	Description
void refresh_node_string (Exp_struct_ptr, SymTbl)	This function creates the node string, the actual X string that is displayed in a table cell, including those of all of the nodes children.
Exp_struct_ptr create_node (Expn, Path, Exp_struct_ptr, SymTbl)	This function creates a new node and recursively creates child nodes. Exp_struct_ptr is a pointer to the node that is to be the parent of the node created by this function.

Access Program	Description
int node_assignnodeoffsets (Exp_struct_ptr, int o)	This program assigns the node offsets which correlate the node structure (representing the Table Holder contents) to the user display of a scalar expression. Offset int o is the total offset so far assigned to the node structure to which this node belongs and indicates at how many characters from the left of the user display this node begins. The value returned is the end offset of this node (i.e. where the node ends).
void node_freenode (Exp_struct_ptr)	This program frees memory used by the node, freeing Paths and all child nodes recursively.
Exp_struct_ptr ec_getclosestnode (Exp_struct_ptr s, Exp_struct_ptr c, int o)	Given the starting node s, the closest node c and the node offset value o, this program finds the node in the node structure that most closely represents the cursor's location. (i.e. finds where in the node structure the cursor is located, where an action would affect the node structure). The closest node c is included as this is a recursive function; if the current node is closer to the cursor position o than c, this node becomes c and is returned.
void eoc_insertnode (Exp_struct_ptr, Id, EType)	This function inserts the specified ID (which is a default symbol) into the node's expression at the current insertion point.

7.2.3 Display Table Module (c_table.c)

This module contains functions that bring together the node structure and the tabular user interface to display the instance. The secret of this module is the appearance of the instance on the screen.

Access Program	Description
bool ct_refreshtabledisplay(Inst_ptr)	This function gleans all data required from the provided instance structure. Using this, the entire node structure for the table is created and a node structure is assigned to every cell.
void ec_refreshInstance (Inst_ptr)	This function performs a complete refresh of the instance's user display freeing the current associated node structure and rebuilding it from scratch to reflect a change in the structure of the working expression.
void main_TabularInstanceBuild (Inst_ptr)	Builds the visible representation of the tabular instance.

7.2.4 Table Widget Module (tablewidget.c)

The Table Widget Module contains functions that aid in the display of a tabular instance of AUFMG. This module hides the X window functions dealing with the widget.

Access Program	Description
void TableWidgetDataEntry (Inst_ptr, int, int, int, Expn_struct_ptr)	This function assigns the proper callback information and node data structures to the cell widget (grid, row and column) in a tabular instance of the AUFMG.
void build_table_window (Inst_ptr, int, int)	This function creates the user interface for a tabular instance of the AUFMG.
void close_table_window (Inst_ptr)	This function deletes the user interface for a tabular instance of the AUFMG.

7.2.5 Cell Editor Module (tctemotif.c)

The Cell Editor Module contains functions for creating the cell editor's user interface. The secret of this module is the appearance of the Cell Editor's user interface and the X window functions used to create the user interface..

Access Program	Description
void build_window (Expn_struct_ptr)	This routine creates the Cell Editor's user interface.
void close_window ()	This routine closes the Cell Editor.

7.2.6 Menu Module (tctmmontif.c)

The Menu Module contains functions for creating the menubar of the AUFMG's Main Window and the Cell Editor. The secret of this module is the appearance of those menubars and the X window functions for creating those menubars.

Access Program	Description
widget create_expression_menubar (widget, Inst_ptr)	This function creates widgets for and assigns callbacks for the main menu of a cell editor.
widget create_table_menubar (widget, Inst_ptr)	This function creates widgets for and assigns callbacks for the main menu of a tabular instance of the AUFMG.

7.2.7 Dialogue Module (tctdmotif.c)

The Diaglogue Module contains functions that are responsible for the dialogue box creation and methods to output various information to the user. The secret of this module is the X window functions for creating these dialog boxes.

Access Program	Description
void createsymbolbox (Inst_ptr)	This function creates a list box containing all of the user defined symbols available for that instance of the AUFMG. These symbols can be inserted directly into the AUFMG's edit window.
int tdm_getintinput (Inst_ptr, char *title, char *ques, int upper, int lower)	This function prompts the user for an integer value, using prompts provided (title and ques) and checking against given upper and lower bounds. Retrieved integer value is returned.

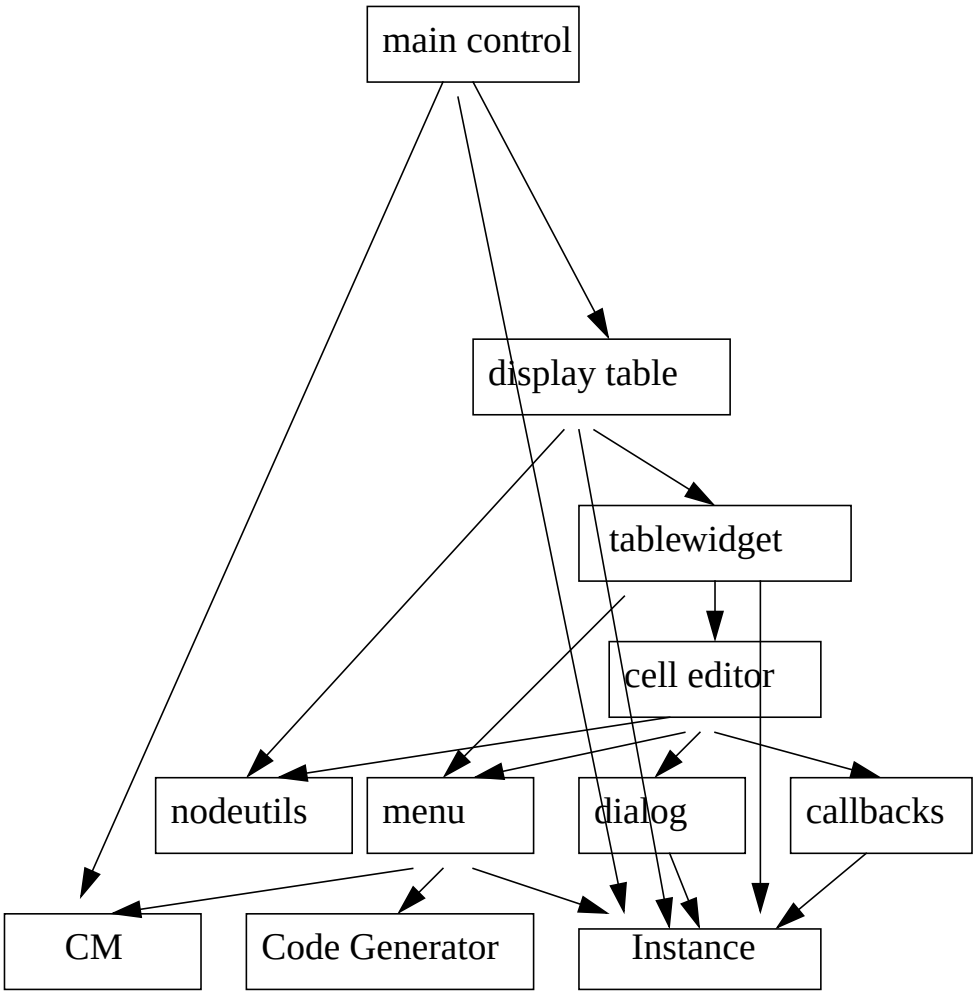
7.2.8 Callbacks Module (editor_callback.c)

The functions contained in the Callbacks Module provide resources that respond to user generated actions. The secret of this module is the X window functions for these call-backs. They are the only functions that provide “callback” behaviour.

Access Program	Description
void ec_insertsymbol (Exp_struct_ptr, Id)	This function inserts the given symbol into the current node structure.
void ec_updatedisplay (Inst_ptr)	This function refreshes the display of the instance.
bool ec_isdefaultoperator (Exp_struct_ptr, Id)	This function returns BOOL_TRUE if the specified symbol Id in a node has a default class assignment.
char * ec_getnodepathstring (Inst_ptr, Exp_struct_ptr)	This function returns a string that describes the path to an expression referenced by the provided node, i.e. the 2nd child of expression “count” would be “count, 2”.

7.3 Module Uses Relation

Figure 7-1 illustrates the Modules Uses Relation of AUFMG tool. Module A is said to use Module B if at least one program in Module A relies on the correct behavior of at least one program in Module B.



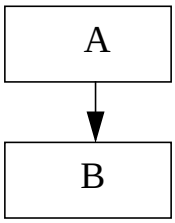


Figure 7-1 Module A uses Module B

7.4 Program Uses Relation

This section illustrates the detailed design of AUFMG in terms of the uses relation between the access programs. The following table provides the mapping between the program ids and the program names:

ID	Display Table Module
1	bool ct_refreshtabledisplay(Inst_ptr)
2	void ec_refreshInstance (Inst_ptr)
3	void main_TabularInstanceBuild (Inst_ptr)
	Table Widget Module
4	void TableWidgetDataEntry (Inst_ptr, int, int, int, Expn_struct_ptr)
5	void build_table_window (Inst_ptr, int, int)
6	void close_table_window (Inst_ptr)
	Cell Editor Module
7	void build_window (Expn_struct_ptr)
8	void close_window ()
	Menu Module
9	widget create_expression_menubar (widget, Inst_ptr)
10	widget create_table_menubar (widget, Inst_ptr)
	Dialog Module
11	void createsymbolbox (Inst_ptr)
12	int tdm_getintinput (Inst_ptr, char *title, char *ques, int upper, int lower)
	Callbacks Module
13	void ec_insertsymbol (Exp_struct_ptr, Id)
14	void ec_updatedisplay (Inst_ptr)

15	bool ec_isdefaultoperator (Exp_struct_ptr, Id)
16	char * ec_getnodepathstring (Inst_ptr, Exp_struct_ptr)
	Instance Module
17	bool main_CheckInstModStat (Inst_ptr)
18	bool main_InitInstanceStruce (Inst_ptr)
19	void main_SetInstanceName (Inst_ptr, char *)
20	void main_FreeInstance (Inst_ptr)
21	Path GetInstPath (Inst_ptr)
22	void SetInstPath (Inst_ptr, Path)
23	Expn GetInstExpn (Inst_ptr)
24	void SetInstExpn (Inst_ptr, Expn)
25	SymTbl GetInstSym (Inst_ptr)
26	void SetInstSym (Inst_ptr, SymTbl)
27	Shape GetInstShape (Inst_ptr)
28	void SetInstShape (Inst_ptr)
	Node Utilities Module
29	void eoc_insertnode (Inst_ptr, Id, EType)
30	void refresh_node_string (Exp_struct_ptr, SymTbl)
31	Exp_struct_ptr create_node (Expn, Path, Exp_struct_ptr, SymTbl)
32	int node_assignnodeoffsets (Exp_struct_ptr, int o)
33	void node_freenode (Exp_struct_ptr)
34	Exp_struct_ptr ec_getclosestnode (Exp_struct_ptr s, Exp_struct_ptr c, int o)

Program A uses Program B if there are situations where the correct functioning of Program A depends on the correct behavior of Program B [21].

The program uses relation of AUFMG is shown as a matrix in the following table. “*” in the matrix indicates that the program located in the corresponding row uses the program located in the corresponding column.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
1				*																							*			*		*		
2					*												*					*	*	*	*			*		*				*
3					*	*																	*		*				*		*		*	*
4																					*		*		*									
5							*		*	*								*	*								*							
6								*												*														
7									*		*	*	*	*	*	*																*		
8																																	*	
9																							*											
10																							*											
11																									*									
12																										*								
13																									*									
14																	*						*				*							
15																									*									
16																					*													

Programs after 16, from 17 to 34, do not access any other programs. They are the programs that only used by others.

Chapter 8

User Guide for the Airline Seats Upgrade Policy Family Member Generator

The AUFMG tool is used to generate a member of the airline seats upgrade policy family. This tool is based on the TCT (Table Construction Tool) of TTS [17] and uses the Code Generator of TTS [1] to generate the source code.

8.1 Overview of the AUFMG tool

There are two GUIs in this tool, one is the Main Window of AUFMG, the other one is the Cell Editor.

To invoke the main window of AUFMG, type “fmg” from the prompt.

To invoke the Cell Editor, click on any headers or main grids of a table. If that cell is editable, the Cell Editor window will show up. In some cases the user is not allowed to edit the header. Then, instead of the Cell Editor, the user will get an error message.

8.2 The Main Window

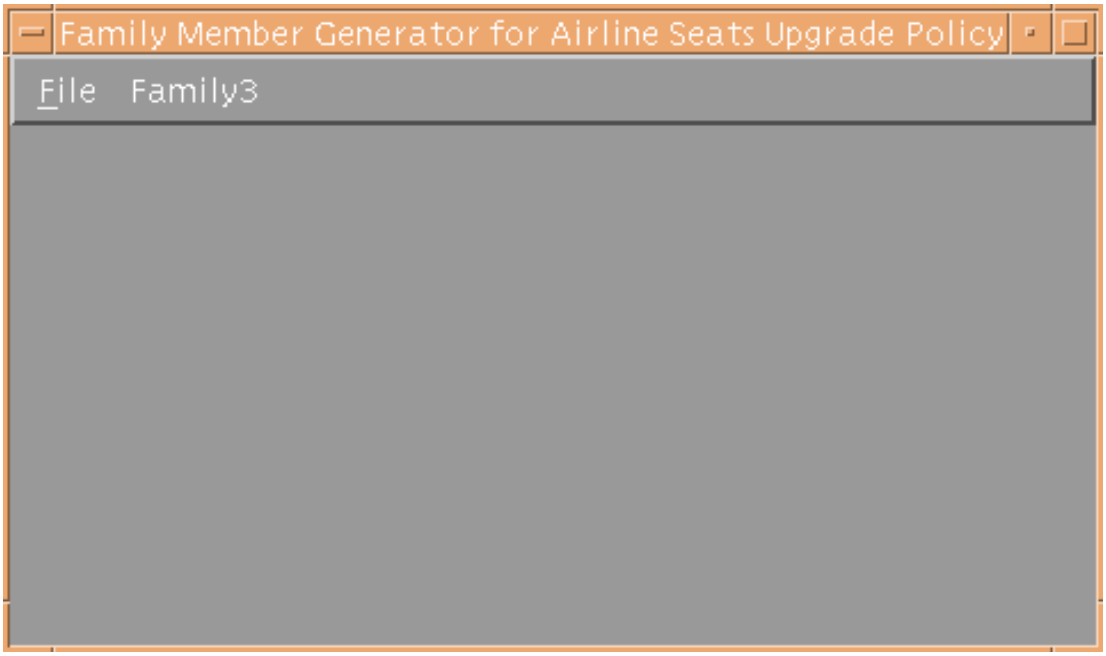


Figure 8-1 the Main Window of AUFMG

8.2.1 Under the File menu, there are 7 items:

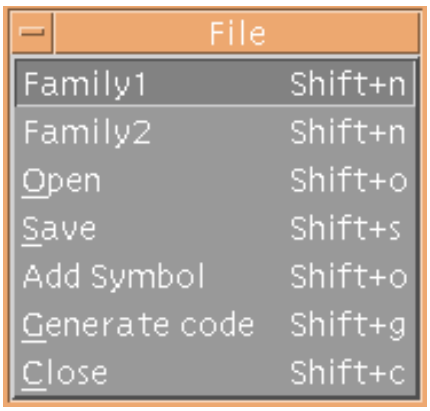


Figure 8-2 File Menu

- Family1 is used to edit table for members of Family1.

After you click the Family1 button a partially completed table will show up:

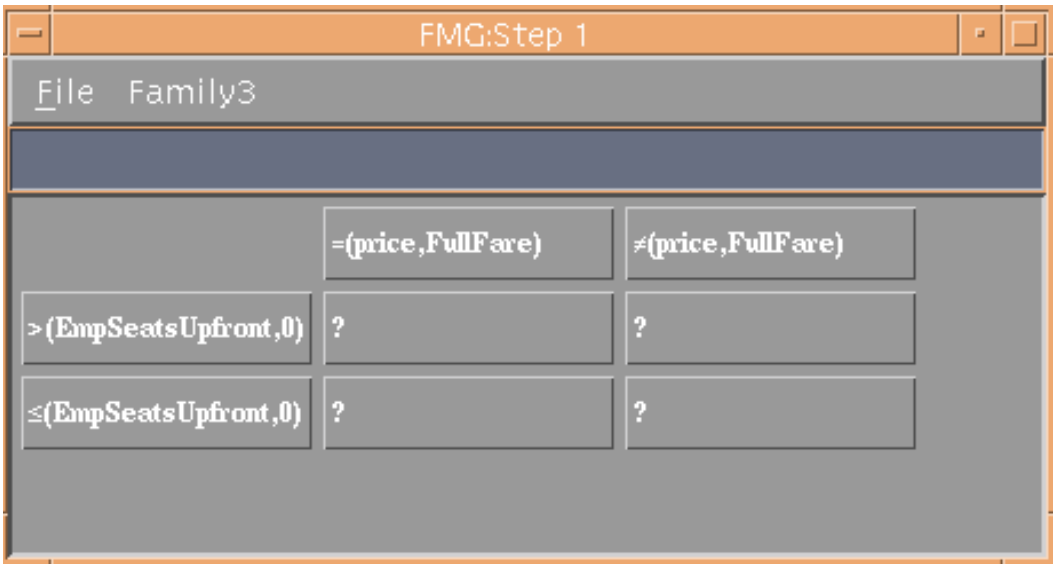


Figure 8-3 The Family1 partial completed table

In this family you can only edit the cells in the main grids (marked with “?”. See Chapter 5 for details).

- Family2 is used to edit the table for members of Family2.

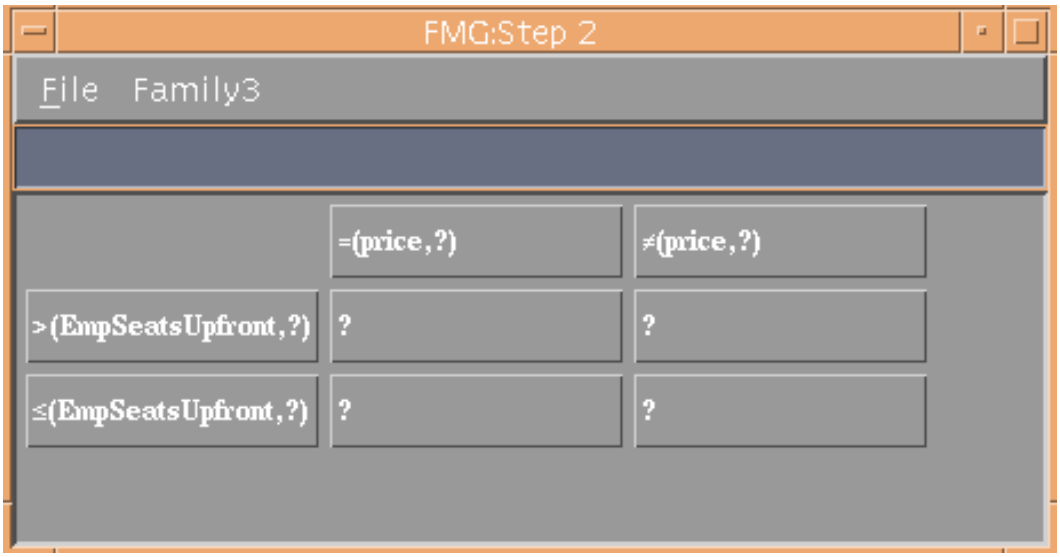


Figure 8-4 The Family2 partial completed table

The difference between family1 and family2 is that in family2 you can edit both cells in the main grids and in the headers. That is you can change the values of the variables in the headers but you cannot change the variables in the headers.

- The Open button is used to open a file previously saved.

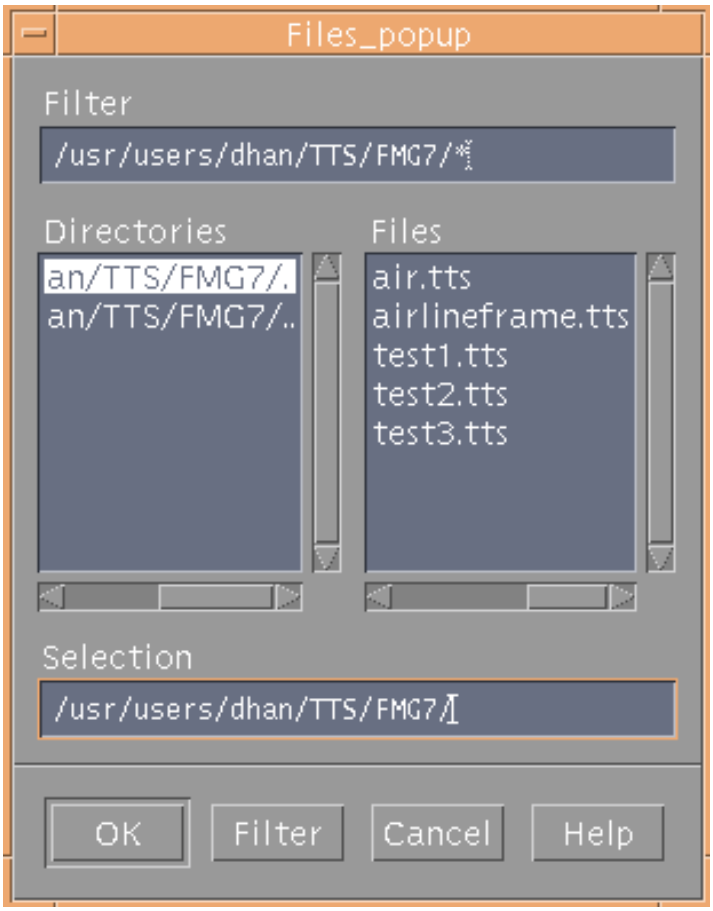


Figure 8-5 Open file window

- The Save button is used to save a table that you edited in the .tts format.

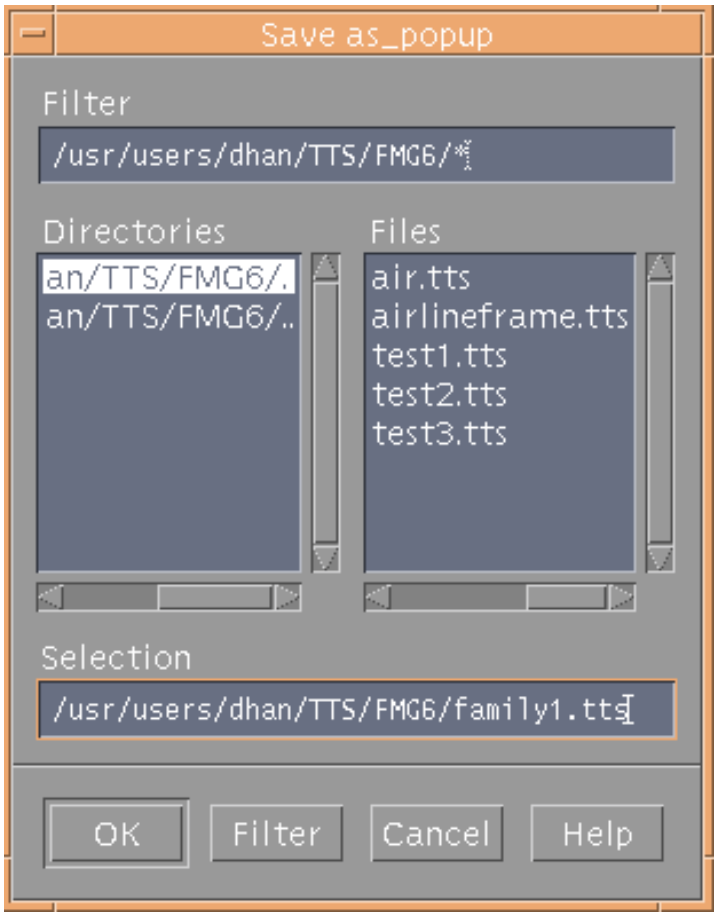


Figure 8-6 Save As window

- The Add Symbol button is used to add integer symbols in the symbol box. You have to add the symbol into the symbol box before you can use it.

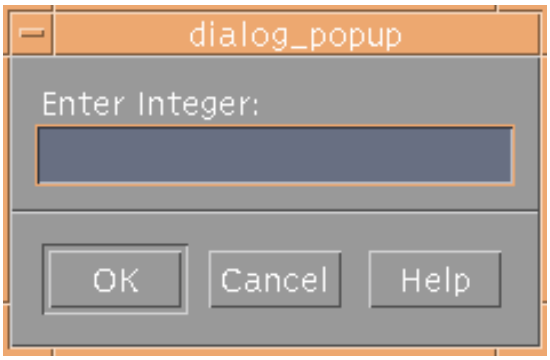


Figure 8-7 Add Symbol window

- The Generate code button is used to generate source code for the table you just edited.
- Close is used to exit AUFMG tool.

8.2.2 The Family3 menu

The Family3 menu is used to create table for family3. In family3 you can define the shape of the table (No. of columns and No. of rows), you can also edit all the cells in both main grids and headers. When you click on it the following window will pop up:

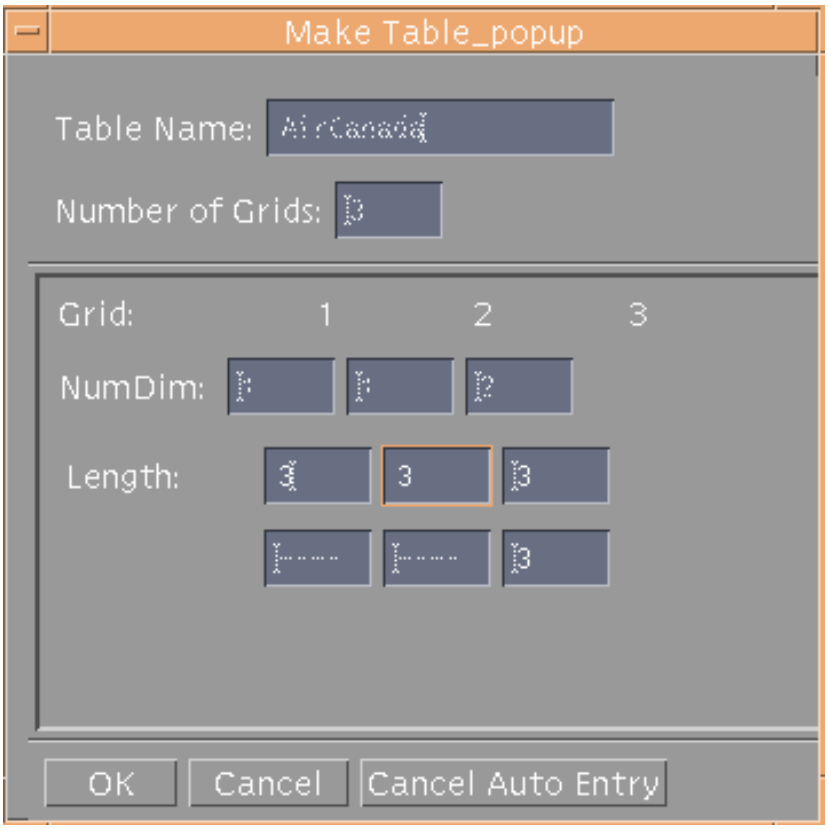


Figure 8-8 GUI for create new table

Here you input the name of the table and the shape of the table. An empty table will show up in the Main Window. See Figure 8-24.

8.3 The Cell Editor

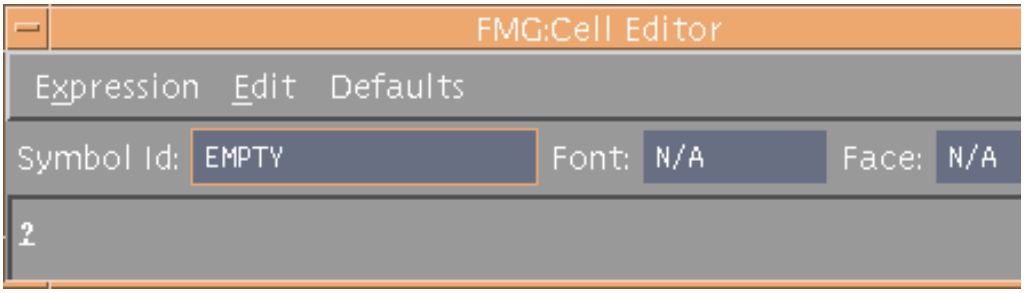


Figure 8-9 Cell Editor

When you invoke the Cell Editor, a Symbol Box comes with it. The Symbol Box allows you to select the symbols you need to display in that cell.

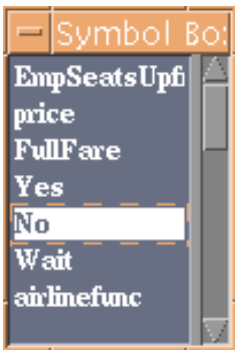


Figure 8-10 Symbol Box

Under the Expression menu, the following items appear:

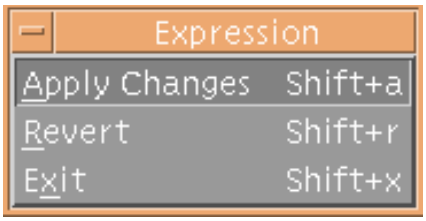


Figure 8-11 Expression Menu

- Apply Changes: This applies to any modifications made in this instance.
- Revert: This discards any changes made in this instance.
- Exit: This option closes the Cell Editor.

The Defaults menu and the symbol box are used to insert symbols into an expression. You simply select a default symbol or double click on a symbol in the symbol box to insert a symbol into the expression.

The Defaults are grouped according to function and tag type:

Figure 8-12 Defaults symbol menu of Cell Editor

Let us take the predicates for example. When you click on that button, all the symbols for this group will show up:

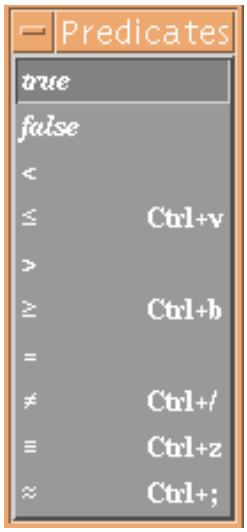


Figure 8-13 Symbols in Predicates group of Defaults menu

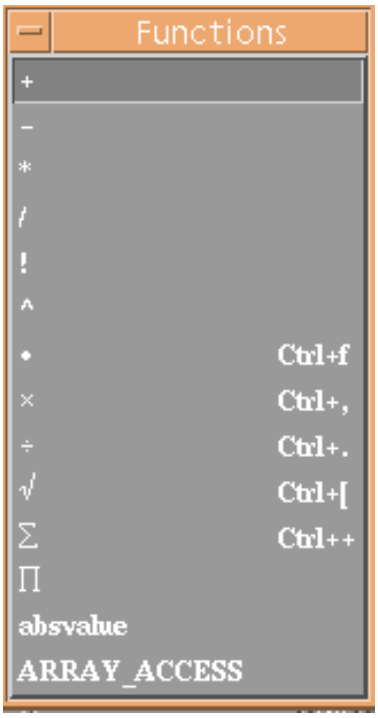


Figure 8-14 Symbols in the Functions group of Defaults menu

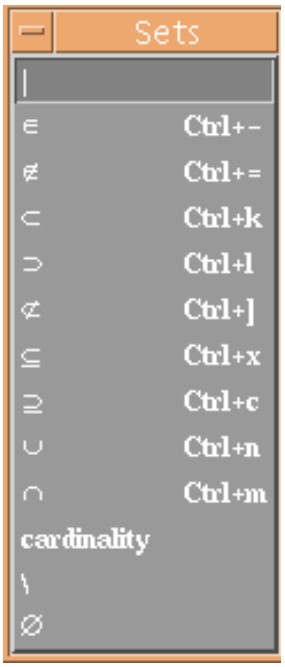


Figure 8-15 Symbols in Sets group of Defaults menu

Figure 8-16 Symbols in the Logical group of the Defaults menu

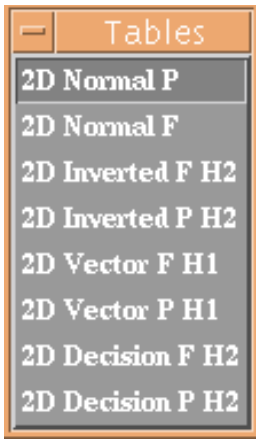


Figure 8-17 Symbols in the Tables group of the Defaults menu

Figure 8-18 Symbols in the Events group of the Defaults menu

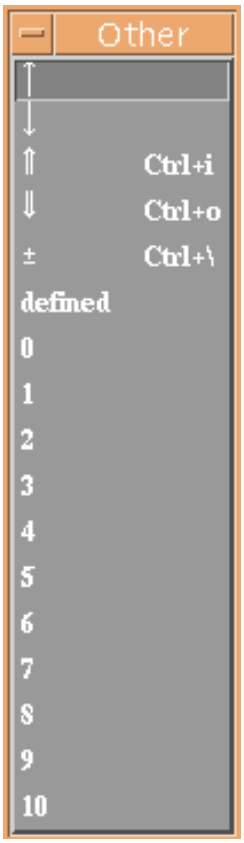


Figure 8-19 Symbols in the Other group of the Defaults menu

8.4 Steps to Create Family members from the examples showing in Chapter 5

Based on the three examples in Chapter 5, this section describes the steps for creating family members.

8.4.1 Create a Member of Family1

This section describes the steps for creating a member of family1 which is described in Chapter 5.

1. Type “fmg” then press ENTER from the prompt, the AUFMG main window will show up as shown in Figure 8-1.
2. Select the “Family1” button from the File menu. A partially completed table will show up. See Figure 8-3.
3. Click the cells in the main grid, then the “Cell Editor” window and the “Symbol Box” window will show up. See Figure 8-9 and Figure 8-10.
4. In the “Symbol” window, double click the “Yes”, “No” or “Wait” button. “Yes”, “No” or “Wait” will show up in Cell Editor window.

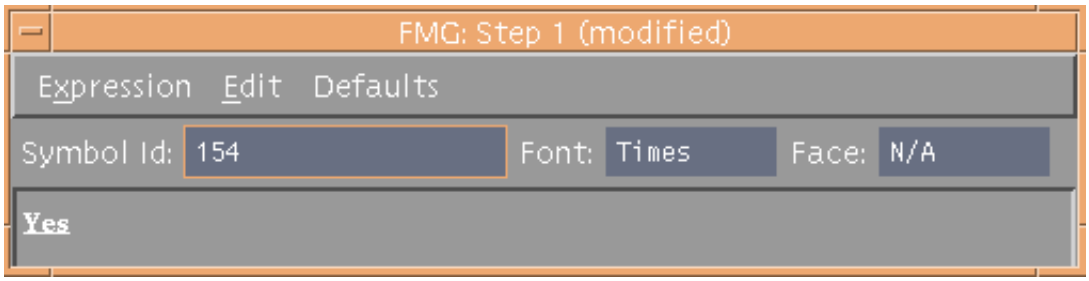
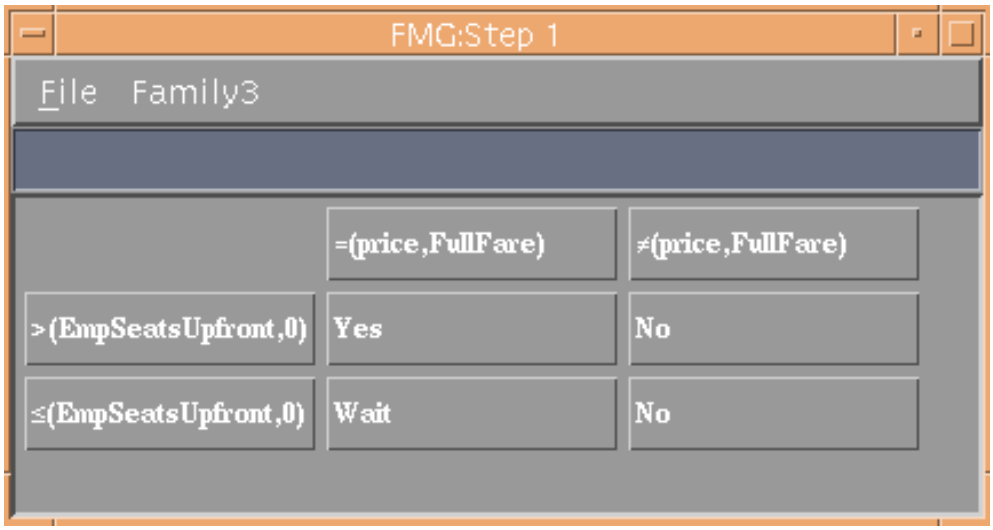


Figure 8-20 Cell Editor with “Yes” symbol

5. Click the “Apply change” button under Expression menu in “Cell Editor”, this will show up in the cell of the main window.
6. Redo step 3, 4 and 5 for all the cells in the main grid. The table will looks like the following:



. Figure 8-21 A example member of Family1

7. Select the ”Save AS” button under file menu and provide a file name you want.
8. Select the “Generate Code” button under file menu. 2 files with the name you provide in setp7 plus “.c” and “.h” at the end will be generated.
9. Select the “Exit” button under the file menu to close the program.

8.4.2 Create a Member of Family2

This section describes the steps for creating a member of the Example2 family as described in Chapter 5.

1. Type “fmg” then press ENTER from the prompt. The AUFMG main window will show up. See Figure 8-1.
2. Select the “Family2” button from the File menu. A partially completed table will show up. See Figure 8-4.
3. Click the cells in the headers then the “Cell Editor” window and “Symbol Box” window will show up. This allows you to input the values for the headers.

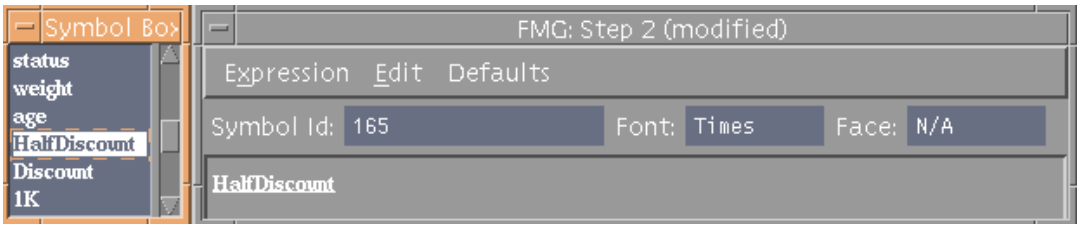


Figure 8-22 Cell Editor and Symbol box with the value “HalfDiscount”

4. In the Symbol window, double click the symbol that you want; it will then show up in the Cell Editor window.
5. Click the “Apply change” button under Expression menu in “Cell Editor”. It will show up in the header.
6. Click the “Exit” button. This will terminate the Cell Editor.
7. For the cells in the main grids, perform steps as 3 to 6 in section 8.4.1.
8. Redo step 3 to step 7 for all the cells in the table. An example member of Family2 may look like the following:

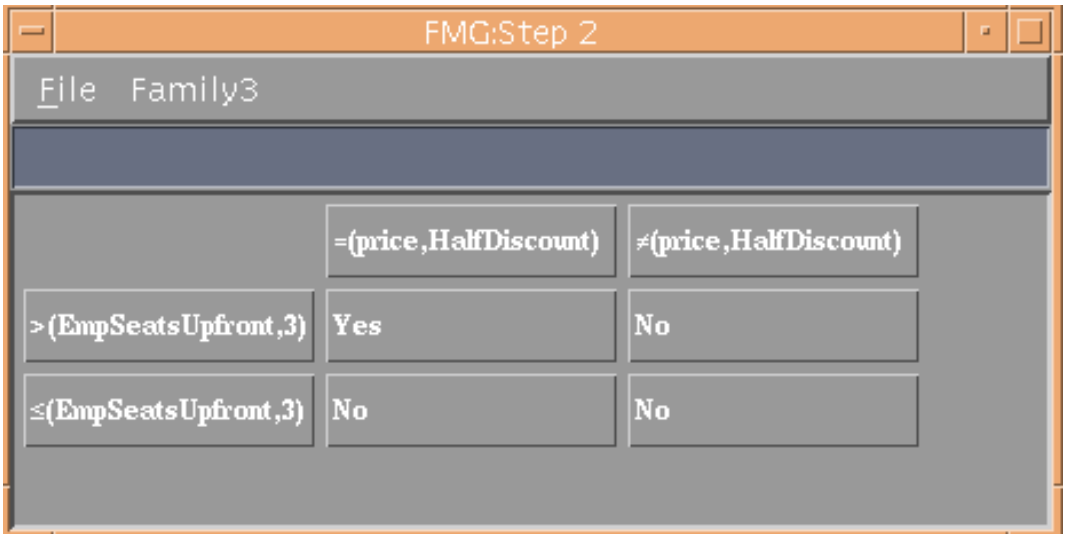


Figure 8-23 A example member of Family2

9. Select the ”Save AS” button under the file menu and provide a file name you want.

10. Select the “Generate Code” button under file menu. 2 files with the name you provide in setp7 plus “.c” and “.h” at the end will be generated.
11. Select the “Exit” button under file menu to close the program.

8.4.3 Create a Member of Family3

This section describes the steps for creating a member of Family3 as described in Chapter 5.

1. Type “fmg” then press ENTER from the prompt. The AUFMG main window will show up.
2. From Family3 menu, select “tables” button. A list of table types will show up. Select the table type you want. A make table window will pop up as Figure 8-8.
3. Input the table name and No. of column and the length of table. A table with empty headers and the main grid will show up.

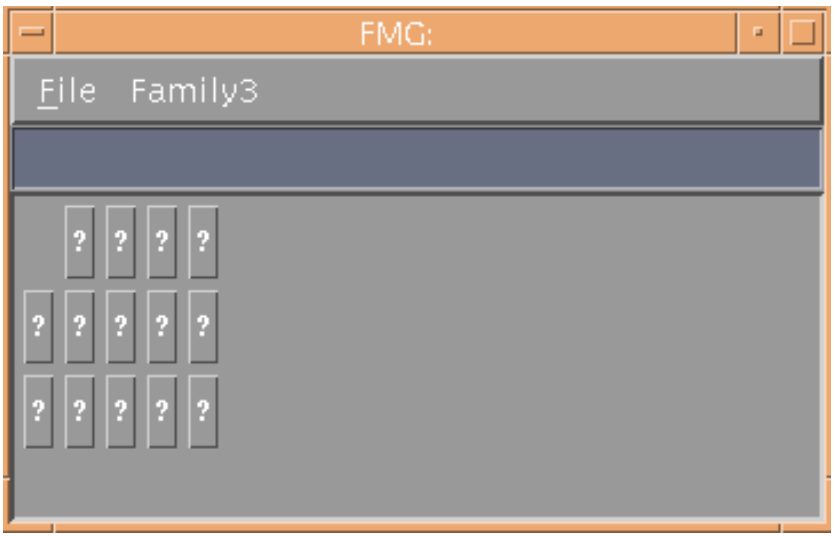


Figure 8-24 empty new table

4. For the cells in the main grids do steps as 3 to 6 in section 8.4.1.
5. Click the cells in the headers. The “Cell Editor” window and “Symbol” window will show up which allows you to input the predicates and symbols for the headers.
6. Select Predicates from the Defaults menu, click =, < or > etc. This will show up in the cell editor.

7. In the “Symbol” window click the symbol that you want, then it will show up in Cell Editor window. Then again in the “symbol” window, double click the value you want for this symbol, it will also show up in the Cell Editor window.
8. Click the “Apply change” button under Expression menu in “Cell Editor”. It will show up in the headers.
9. Redo step 5 to step 8 for all the cells in the headers until you are finished. An example member of Family3 should look like this:

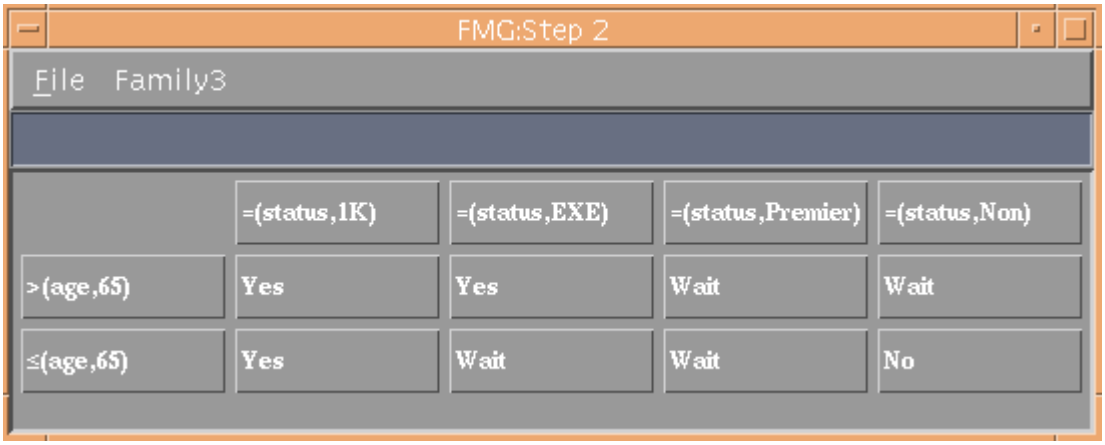


Figure 8-25 An example member of Family3

10. Select the ”Save AS” button under file menu and provide a file name you want.
11. Select the “Generate Code” button under file menu, 2 files with the name you provide in setp7 plus “.c” and “.h” at the end will be generated.
12. Select the “Exit” button under file menu to close the program.

Chapter 9

Conclusions and Future Work

9.1 Conclusion

This work is based on the theory of domain engineering as developed at Lucent, and TTS, which was developed at McMaster University's Software Engineering Research Group.

This work demonstrates how tables can make applying the FAST process easier. Instead of writing in a special purpose language we can let people complete tables and generate code from them.

This work illustrates different families with a large family member generator which gives the PLE (Product Line Engineering) more choice in tables.

Together with this work an application is developed which shows three airline seats upgrade policy families as examples. By using this application the user can generate member programs for airline seat upgrade policy example families.

Information hiding technology [20] was applied during the process of design and implementation of this application.

After finishing this work we can get the following conclusions:

- When the assignment is a set of programs and they have many thing in common, you may consider using the FAST process. The more programs in this set, the more suitable the FAST process is. For example, an application has many versions, or needs to run on many different platforms.
- When you want to develop a set of programs and you want to get the applications fast, you may consider using the FAST progress together with TTS technology. For example, the mobile phone plan family we described in chapter 6, in order to change their plan quickly to response to the market change, it was a good idea to use TTS applied with the FAST process. The only thing you need to do is to fill in the table to get the applicatlion immediately.

- If you want to use a table as the user interface, you should consider applying the TTS together with FAST process. At Lucent they have developed an application which lets the user draw nodes to show the properties of the switches and relations between them. It is a very nice tool, but a problem appeared when they tried to draw more than 100 nodes. There was a mess on the screen! By applying TTS together with the FAST process we solved this problem. Although the table is long it shows the properties and relations of the switches clearly.

9.2 Future Work

This application can only be used to generate programs for certain airline seat upgrade policy family members. It is not a general tool. By applying the model presented in this work, how ever, it is easy to generate programs for any other families. The family we discussed in Chapter 6 would be a good example to work on.

A more general program family member generating tool of TTS should be considered as the future work after this to make it easy to apply the FAST process on any other families.

This tool relies heavily on the TTS and if a user want to use this tool, he/she must have some basic knowledge about the TTS. Some future work should be done to separate this tool from the TTS. The ideal would be that the user of this tool should be able to use it without knowing anything about the TTS.

Bibliography

[1] Abraham, Ruth, “Evaluating Generalized Tabular Expressions in Software Documentation”, CRL Report 346, February 1997.

[2] A.V. Aho, B.W. Kernighan, and P.J. Weinberger, The AWK Programming Language. Reading, Mass.: Addison Wesley, 1986.

[3] Batory, D., Coglianese, L., Goodwin, M., Shafer, S., “Creating Reference Architectures: An Example from Avionics”, private communication.

[4] Batory, D., O'Malley, S., “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, ACM Transactions On Software Engineering and Methodology, October 1992.

[5] Campbell, G.H. Jr., Faulk, S.R., Weiss, D.M., “Introduction to Synthesis”, INTRO-SYNTHESIS-PROCESS-90019-N, 1990, Software Productivity Consortium, Herndon, VA.

[6] Campbell, G. H. Jr., “Abstraction-Based Reuse Repositories”, REUSE _REPOSITORIES -89041-N, 1989, Software Productivity Consortium, Herndon, VA.

[7] Cleaveland, J.C., “Building Application Generators”, IEEE Software, pp. 25 - 33, 1988.

[8] Coplien, J., “Multi-Paradigm Design for C++”, Addison Wesley Longman, 1998, ISBN 0-201-82467-1.

[9] Coplien, J., Hoffman, D., Weiss, D., “Commonality and Variability in Software Engineering”, IEEE Software, November/December 1998, pp. 37-45.

[10] Cuka, D., Weiss, D., “Engineering Domains: Executable Commands as an Example”, Lucent Technologies, private communication.

[11] Dijkstra, E. W., “Notes on Structured Programming”. Structured Programming, O.J. Dahl, E.W. Dijkstra, C.A.R.Hoare, eds., Academic Press, London, 1972.

[12] Heninger, K., Parker, R., Parnas, D.L., “A Procedure for Designing Abstract Interfaces for Device Interface Modules”, IEEE Fifth International Conference on Software Engineering, 1981.

[13] Hook, J., Walton, L., “On Understanding a Commonality Analysis”, Oregon Graduate Institute of Science and Technology.

[14] Hook, J., Walton, L., “The Design of Message Specification Language”, June 1997,Web Site, Oregon Graduate Institute of Science and Technology.

[15] Hook, J., Widen, T., “Software Design Automation: Language Design in the Context of Domain Engineering”, Oregon Graduate Institute of Science and Technology.

[16] Ladd, D.A., Ramming, J.C., “A*: A Language for Implementing Language Processors”, IEEE International Conference on Computer Languages, 1994.

[17] Li, Weimin, “Table Construction Tool”, CRL Report 330, July 1996.

[18] Neighbors, J., “The Draco Approach to Constructing Software from Reusable Components”, IEEE Transactions on Software Engineering, SE-10, 1984.

[19] Parnas, D. L., “On the Design and Development of Program Families”, IEEE Transactions on Software Engineering. Vol. SE-2, No.1, March 1976.

[20] Parnas, D.L., “Designing Software For Ease Of Extension and Contraction”, IEEE Transactions on Software Engineering, SE-5, March 1979, pp. 128 -138.

[21] Parnas, D.L., “On the Criteria to be Used in Decomposing Systems into Modules”, Commun ACM, vol. 15, pp. 1053-1058, December 1972.

[22] Software Engineering Research Group, “Table Tool System Developer’s Guide”, CRL Report No. 339, Telecommunications Research Institute of Ontario(TRIO), McMaster University, Hamilton, Ont., January 1997.

[23] Tracz, W., “A Product Line Engineering Process”, set of slides from private communication.

[24] Weiss, David M., Lai, Chi Tau Robert, “Software Product Line Engineering”, Addison Wesley Longman, 1999.