AT&T Bell Laboratories

subject: **A Framework of Feature Management for Telecommunication Networks**

date: **December 17, 1993**

from: **Yao-Nan Lien**
Org. 198243000
HO 2B-510A
x6799

INTERNAL MEMORANDUM

This document analyzes service integrity, feature interaction, and feature management problems in telecommunication networks. A new computational model, the *Coordinated Finite State Machine*, is proposed to model the feature management. Together with a state representation scheme and an implementation plan, this model provides a framework on which a call processing software with symbolic and dynamic feature management can be easily developed. Further, the development of a feature under this framework can be completely independent from other features even if there is a feature interaction. The feature interactions that contend for the same event can be fully automatically detected under this framework.

1. INTRODUCTION

Public telecommunication networks must maintain a very high degree of integrity, called the *network integrity*, to ensure the operation of the network consistent with the specified behavior of the network. There are various problems in existing and future telecommunication networks that might threaten to violate the network integrity. Among them, poor management of feature interactions in call processing software is a major source of the problems. This document will first analyze some problems, namely service integrity, feature interactions, and feature management problems. Their causes, classification, difficulty, and desired properties of their solutions will be discussed. Then, we will propose a framework of feature management on which a call processing software with better feature management can be easily developed. The framework includes a state representation scheme, a computational model and an implementation plan.

Section 1 analyzes the problem domain. Section 2 introduces the finite state machine approach for the call processing. The rest of the document describes the framework.

1.1 Services and Features

A *service* is a package of incrementally added functionality providing services to subscribers or the telephone administration. Subscribers enter and communicate with a telecommunication network by sending signals through their CPEs (e.g. telephone sets). The network processes users' requests using the *call processing logic* embedded in network elements such as switches or SCPs. The most basic service is the POTS call (Plain Old Telephone Service).

Modern networks are able to offer more advanced functionalities, called *features*, to provide more sophisticated services such as *Call Waiting*, *Call Forwarding*, and *Three-way Call*. To the call processing logic, all service requests, whether from end subscribers or from system administrators, are treated as features. This document will focus on how the call processing logic handles these service requests. To simplify the discussion of the proposed framework, no distinction is made between services and features. Both of them are referred to as "features" in the description of the proposed framework.

1.2 The Service Integrity Problem

1.2.1 Definition

The *service integrity* ensures the collective behavior of all services comply with the network integrity. There are other causes that may violate the network integrity. For example, the data in some SCP/NCP may be corrupted by a misbehaved DBMS resulting in a service interruption.

The service integrity is one of the most important criteria for any service/feature designed for a telecommunication network, especially for an *Advanced Intelligent Network* (AIN), which has a goal to accelerate the introduction of new services/features in a multisupplier, competitive environment. Such an ambitious goal inherently invites more opportunity to violate the network integrity. Before this goal can be fully realized, a better understanding is needed of how to maintain the required level of integrity.

It is the service designer's major responsibility to ensure that all services/features introduced into the networks comply with service integrity rules. For this reason, it is very critical for a telecommunication network to define its integrity policy and rules, investigate the potential threat, and take appropriate actions.

1.2.2 Examples.

One such example is the *Outgoing Call Screening* (OCS) feature, which blocks all outgoing calls to certain destinations, may be violated by the *Call Forwarding* (CF) feature by forwarding incoming calls to the blocked destinations via a non-blocked destination. (Your kids might be able to make "900" phone calls by calling a friend's phone and forwarding the calls to the number they really want to dial.) The integrity of the Outgoing Call Screening feature is actually violated by the Call Forwarding feature.

I personally saw a case that the outgoing call screening feature of a particular type of PBX was violated by a very complicated combination of features, applying sequence, event, and timing.

1.2.3 Integrity Policy and Rules

Some examples of integrity policy are as follows:

- The introduction of a new service/feature must not change the expected behavior of other services/features and vise versa.
- The introduction of a new service/feature must not affect the system security.
- The customer of one service must not be able to access/change the services that he/she has not purchased.
- The customer of any service must not be able to access/change other customers' information.

A more complete set of integrity rules is not available at this point in time. It would require a considerable effort to specify the desired level of integrity.

1.3 The Feature Interaction Problem

1.3.1 Feature Interaction and Service Integrity

It is quite possible that a set of features behaves very well individually but interferes with each other when working together. This is the so called *feature interaction problem*, which is a major threat to violate the service integrity. The *feature interaction problem* is defined as: *the actual behavior or results of a feature becomes inconsistent with its specified behavior or expected results in the existence of other features.*

The example shown in Section 1.2.2, the OCS-CF interaction, is actually a well-known feature interaction problem.

Another well known example is the interaction between the Call Waiting (CW) and the Call Forwarding Busy (CFB) features. When two persons are talking to each other over a phone and a third person is trying to call one of them who subscribes to both CW and CFB features, (assuming the phone company didn't know the problem), the system runs into an ambiguous state that both features can be invoked. More specifically, the call processing logic will put a telephone (a half-call) into a "TALKING" state when it is engaging in a talk with another telephone. When the third party calls, the system sends a signal (event), CALL_PRESENTED, to the call processing logic. Normally the call processing logic will refuse to take the call and the third call will get a busy signal. If the subscriber subscribes to either the Call Waiting or the Call Forwarding feature, the call processing logic has to respond to the signal by executing appropriate actions. If the subscriber subscribes to both features and no resolution rule is specified, the call processing logic will not know what to do upon the two conflicting commands and the resulting action will be nondeterministic.

1.3.2 Causes of Feature Interactions and Their Classification

The problem has many manifestations requiring cooperation between many different groups of people to detect and to resolve.

- To electrical engineers, feature interactions may be caused by signal ambiguities and require a better signaling plan.
- To requirement writers, feature interactions may be caused by ambiguous or incomplete specification, and their solution requires a better specification development environment for creating concise, accurate specification.
- To software developers, the problem involves issues such as software flexibility and extensibility. The solution requires better methodologies and development environment for designing, implementing, and maintaining feature-rich software systems.

A classification that helps in better understanding the different types of feature interactions is the first step to approach the problem.

A comprehensive classification of feature interactions are as follows [1]:

1. Violation of feature assumptions

- assumptions about naming
- assumptions about data availability
- assumptions about administrative domain
- assumptions about call control
- assumptions about signaling protocol

2. Limitations on network support
   - limited terminal equipments signaling
   - limited functionalities for communications among network components

3. Intrinsic problems in distributed systems
   - personalized instantiation
   - timing and race conditions
   - distributed support of features
   - non-atomic operations

This classification is not an exhaustive list. However, it already covers much more than what we can deal with today. Detailed descriptions can be found in [1].

1.3.3 Classification By Contention Types

Some feature interactions can be classified into the following two categories based on the type of subject contended by the features involved in an interaction:

1. **Event-contention**

   This is the type of interactions when two or more features are interested in the same event at the same time (the same state). Different features give conflicting commands to the system to execute. The CW-CFB interaction is of this type.

2. **Data-contention**

   This is the type of interactions when some information used by one feature is modified by another feature resulting in behavior unexpected by the first feature. The OCS-CF interaction shown in Section 1.2.2 is of this type.

Again, this classification is not exhaustive. It will be essential to better understand the feature interaction problem in order to obtain a more complete list.

1.3.4 Feature Interaction Detection and Resolution

The ideal time to detect (discover) and resolve feature interactions is in the system analysis phase. During this phase, system engineers must consider in advance all possible feature interactions and provide adequate solutions in the requirements. Otherwise the implementors will take the freedom to resolve it independently possibly resulting in many surprises at test or in the field. Unfortunately, this is a time consuming manual process and is the primary cause for long

lead times to introduce new services and features.

1.3.4.1 Detection

Some challenges to the feature interaction detection are as follows:

1. A feature interaction may occur under an arbitrary number of features, an arbitrary applying sequence, and an arbitrary event and timing sequence.
2. There is no guideline or methodology to follow in examining feature interactions even for a specific combination of features. It is almost completely ad hoc. Only a few cases exist where the interaction can be discovered using software in the design time.
3. Lastly, there are too many possible combinations of features. Every combination has a potential to introduce an interaction. To examine all possible combinations is almost impossible, if not completely impossible. Based on an ad hoc estimation, the number of combinations is at least in the order of $N!$, where N is the number of features.

As a matter of fact, the number of possibilities is so huge that even a fastest computer cannot solve the problem. It will take more than 200 years to exhaust all *18!* cases for a computer that can search 1 million cases in one second! Manual detection is absolutely hopeless. It is essential to find a mechanical detection mechanism and to cut some corners to reduce the search space.

1.3.4.2 Resolution

Just like debugging a program, once a bug is found, the rest is usually relatively easy. Similarly, once an interaction is detected, the resolution is theoretically easy. (Some cases such as signaling ambiguity may not be solvable solely by call processing logic alone, though.) For the emerging infocentric services to be provided by a telecommunication network, we expect the following desired properties in feature interaction resolution.

- **Customer specific feature interaction resolution**

  Each customer must be able to decide in which way he/she will like to resolve the interaction subject to the capability and integrity requirements of the system.

- **Dynamic interaction resolution**

  If it is applicable, customers must be able to make their choices on a call-by-call basis, or even in the middle of a phone call.

1.3.5 Programming Support

1.3.5.1 Feature Interaction Detection

Although the ideal place to detect feature interaction problem is in the system analysis phase, it is not practical to expect a complete and perfect design specification in the real world. Research that uses formal specification languages and methodology has been done for several years [4]. For some reasons, to the author's knowledge, no successful story has been reported yet. The only complete machine readable specification in most of the real systems is the source code itself (assuming correct implementation, of course.) To system designers, the source code is actually a precious resource for feature interaction detection. Therefore, it is very desirable to have a programming environment that can extract high level service logic from the call processing source code itself and detect some feature interactions.

Three levels of detection capability are defined in the following table:

Table 1. Classification of Feature Interaction Detection Capability

Level	Feature Interaction Detection Capability
3	Fully Automatic Detection
2	Semi-automatic Detection
1	Manual Detection

- **Level 1 - Manual Detection**

  At this level, designers have to manually detect all potential feature interactions on their own.

- **Level 2 - Semi-Automatic Detection**

  At this level, the programmers have to follow a certain way to write their programs such that the system can automatically detect all feature interactions.

- **Level 3 - Fully Automatic Detection**

  At this level, the system will detect all possible feature interactions without any human intervention.

With Level 2 capability, programmers need to predict potential interactions and follow certain rules to design their software. Otherwise, the system won't be able to detect it. It is unknown at this point in time whether Level 3 capability is feasible for all possible cases or not. We have found a Level 3 (fully automatic) detection mechanism for the event-contention type of interactions and can only find a Level 2 (semi-automatic) detection mechanism for the data-contention type of interactions. They will be discussed in Section 6.

1.3.5.2 Feature Interaction Resolution

Once a resolution is specified, how to support the dynamic customer specific resolution is another great challenge to the program development. One desired property in a programming language is to support symbolic feature interaction resolution. In other words, the developers must be able to specify the resolution by feature's names with minimum effort. It is for sure there is no built-in mechanism for this in conventional general purpose programming languages such as C or C++.

1.4 The Features Management Problem in Call Processing

The feature management involves the following tasks:

- **Feature packaging** - makes features available/unavailable to a system. This includes:
  - add a feature into a service package
  - remove a feature out of a service package
- **Feature provisioning** - makes a feature available to an individual call/customer. This includes:

- activate a feature
- deactivate a feature

- **Run time feature management** - includes:
  - invoke a feature
  - disable a feature
  - set the precedent order or logic of feature execution.

Note that a feature can only be made available to a call if it is available to the system; similarly, a feature can only be invoked by a call if it is available to the call; and a feature can be disabled whether it is already invoked or not.

The run time feature management is essential to resolve the event-contention type of feature interactions. Therefore, a good feature management is needed to support feature interaction resolution.

With limited programmability in conventional call processing environments, feature management is very inflexible resulting in very slow feature creation and complicated maintenance. One major source of the problem is that **there is no notion of "features" in the programming environment**. A feature is usually represented by many pieces of codes spread all over the entire call processing code. To improve the flexibility and the programmability of the network, the programming environment should have the following properties:

1. **Symbolic feature management** - allows features to be managed by names. This implies that it should have a notion of "feature" in its computational model.
2. **Dynamic feature management** - supports run time feature management and is essential for the dynamic feature interaction resolution.
3. **Independent feature development** - allows the creation of separable software components at the feature level. This implies that no communication between features is allowed at the code level or at the execution time. This is a fundamental difference from current approaches, where features are required to communicate to each other to detect and resolve feature interactions. In that case, all features involved in an interaction must understand each other and embed the communication protocols in their codes. Adding a new feature into the system will have to examine and modify all features that interact with this new feature. This will definitely turn a software system into a spaghetti system.
4. **Multiple feature invocation**

   This means that a feature can be invoked when another feature has already been invoked. For a service such as the following hypothetic example, the *Call-Waiting-Conference*, this is an essential property.

   > When a subscriber *A* is talking to other two persons, *B* and *C*, alternatively in a Call-Waiting call, *A* decides to connect *B* and *C* together to form a conference call. This feature will allow *A* to do so by pushing a special button (or flash) to switch to a Three-Way Conference call, instead of having *A* hang off and dial *B* and *C* again.

   In the above example, the Call-Waiting-Conference feature can be invoked while the Call-Waiting feature is not finished. If the independent feature development is enforced, the implementation of Call-Waiting feature would not be able to know the existence of Call-

Waiting-Conference. Therefore, Call-Waiting feature itself cannot hand the control over to other features. If this property is not available, the call processing logic would have to wait until the Call-Waiting feature terminates before it can invoke another feature. The multiple feature invocation property is especially important when the service design environment wish to design reusable services/features such that new services can be created by combining basic services or features together.

2. FINITE STATE MACHINE AND CALL PROCESSING LOGIC

The call processing logic is a process that has been set up to handle a telephone call. It interprets the incoming signals (events) to a phone call and takes appropriate actions in response to the events.

It is a practical tradition to use a "state" based machine, possibly a Finite State Machine (FSM), to handle all possible types of phone calls. However, with so many features available to a phone call, the state machine will be too complicated to manage. One possible solution is to decompose a FSM into some number of FSMs.

2.1 Decomposition of FSM

There are two levels of decomposition. The first level decomposes the FSM of a full-call into some number of half-call FSMs and protocol FSMs. The second level decomposes a half-call into a feature management FSM and some number of feature FSMs each representing a feature.

The first decomposition is already well defined and well accepted by the telecommunication industry. However, there is no known good way to do the second decomposition yet. The following sections describe the current approach in telecommunications for the first level of decomposition and discuss several approaches for the second level of decomposition.

A decomposition is well-defined if the behavior of all sub-FSMs working together is equivalent to the original FSM. As far as the state space is concerned, it must satisfy at least the following conditions:

1. The original state space must be reconstructible from the state spaces of all sub-FSMs.
2. There is only one current state.

2.1.1 Finite State Machine for a Half-call

One good index to measure the complexity of a FSM is the size of its state space. With profound features in a telecommunication network, there are almost infinite (or unbounded) number of states. The following hypothetic "*Phone-Across-America*" example may help readers to imagine how complicated a phone call may be.

> *A phone chain is set up to connect all phones in the whole country (or the whole world) using a chain of Three-Way calls (assuming they all have Three-Way call feature.) In the phone chain, there are analog phones, ISDN phones, shared DN phones, and 800 phones, etc.*

This scenario is entirely possible using today's technology. It would be a nightmare to define a state space that can handle such a complicated scenario. Fortunately, current networks are designed in such a way that each phone call can be decomposed into many separate FSMs. For example, a POTS call can be decomposed into three FSMs, one for originating half-call, one for terminating half-call, and one for the POTS protocol itself. These FSMs working together become

a full-call FSM. Note that these FSMs are not independent of each other. Both originating and terminating FSMs depend on the protocol FSM. During a POTS phone call, two half-call FSMs work together under POTS protocol, to perform a full call.

Not only the decomposition greatly simplifies the problem, it also allows the originating and the terminating FSMs to be reused in many different features with minimal modification.

2.1.2 Feature as a Finite State Machine

With so many features supported by a network, the state space for a half-call is still huge. It can be further decomposed into a feature management FSM and some number of feature FSMs each representing a feature. In this section, various approaches to represent features are discussed. Our approach will be presented in Section 5.

2.1.2.1 One-Process-Per-Feature Approach

One approach (e.g. A-I-Net) is to make each FSM by itself an independent machine having its own state space and control. These FSMs are naturally mapped into processes so that they can be independently designed and interact to each other like distributed processes.

As we stated in the beginning of this section, to the user who is using the phone and features, there should be only one state space and one current state, which is the status of the phone call. No matter how the state space is decomposed, one state space and one current state must be well defined. The idea of making each feature having its own state space will eventually limit the type of services that the platform can provide. At least it will be very difficult for this model to support multiple feature invocation. Once a call is controlled by a feature, other features cannot see the current state and thus, cannot be activated by itself. The only way to do that is to modify the behavior of the controlling feature having it watch for the user's requests and invoke the corresponding features. As we pointed out in Section 1.4, this will violate the requirement of independent feature development.

Further, the system is not well defined if one feature makes a state change and other features cannot "see" the new state, because there should be one single current state seen by all features.

Modeling each feature as a process is also not a good design choice. First, each feature doesn't have to be a real machine (process) even though it is modeled as a finite state "machine". A FSM only contains the instructions telling the system what to do in certain state upon certain event. It doesn't have to be implemented as a complete machine in a physical form (process). Furthermore, it is not desirable to implement FSMs as processes since processes are very expensive in real-time world. It is almost impossible to design a system that needs to use hundreds of processes to process a phone call.

Finally, as mentioned in Section 1.4, allowing communications between processes is not desirable. The communications between two features requires a mutual agreement between them (a protocol), which violates the independence property of the features. Each feature must know in advance which features it might interact with. That requires an exhaustive manual feature interaction detection. Further, it has to implement a protocol within itself for every feature it interact with.

2.1.2.2 One-Object-Per-Feature Approach

In this approach each feature is modeled as an object. Although object-oriented paradigm may provide a better programming environment, additional effort is needed to find a good model. **Object-orientation doesn't automatically lead to a good design**, however. It is yet to be proven how an object-oriented design can fulfill all requirements listed in Section 1.

2.2 What is the state of a feature?

This section will explain some basic concepts of FSM with the intent of clarifying some misunderstandings.

In making each feature a FSM, one natural question is "what is the state of a feature?" Strictly speaking, **it is meaningless to say "state of a feature"**. It is only meaningful to say "*state of a call (half-call)*" because we are describing the status of a half-call. When we say "feature *F* changes from State *S1* to State *S2* on event *E*", it should be read like "when the call is in State *S1* and is given event *E*, feature *F* suggests the system to change the state of the half-call to *S2*." Although some shorthand may be found in this document, readers should always interpret feature FSMs in this way.

One important characteristic of FSM is that it does not have any memory to memorize previous history. (The state itself can be viewed as a one-step memory that stores the net effect of all previous events applied to the call.) It doesn't remember how the current state is reached. It makes decision entirely based on the current state and the current event regardless the history of the call.

3. OBJECTIVES

The objectives of this document is to propose a feature management framework, including a state representation scheme, a computational model, and an implementation plan. This framework will facilitate:

1. easy feature management with dynamic and symbolic capability,
2. fully automatic detection for event-contention feature interactions, and
3. semi-automatic detection for data-contention feature interactions.

The basis of the proposed framework is a new way of doing "Feature-As-A-FSM" decomposition for a half-call. This framework has a universal shared state space and a single current state. The details will be discussed in the following sections. (For simplicity, we will use "call" for "half-call" throughout the rest of the document.)

4. A STATE REPRESENTATION SCHEME

A good state representation scheme is required to handle such a huge state space. Current telecommunication software development community all use the so called *Call Model* in one way or another as part of state representation. As a matter of fact, the state representation scheme used in many telecommunication system, such as 5ESS® switch, is loosely defined by a

0. Registered trade mark of AT&T.

combination of a call model, state variables, and local variables. Before a well-defined state space is formally defined, we cannot have a very clean software system. In the rest of this section, an example of currently used call model will be described first. Then, a formal state representation scheme will be presented.

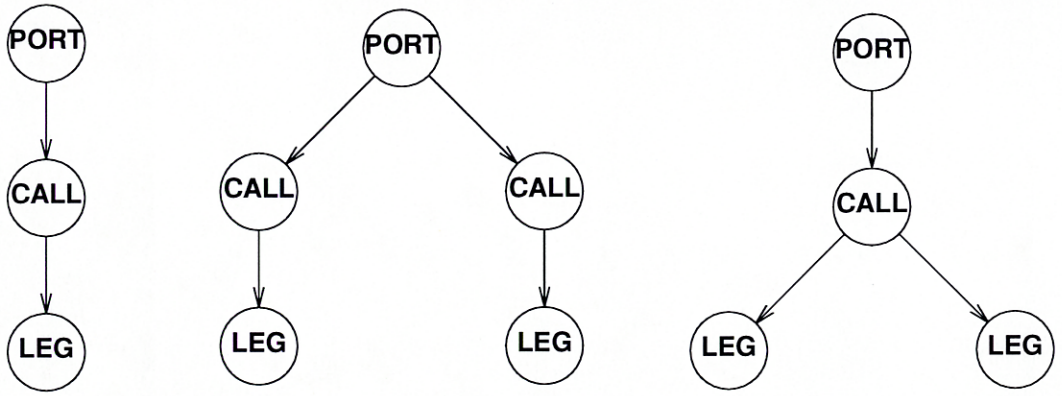
4.1 Call Model For a Half Call

Most of the call models divide a call entity into some number of structurally related logical entities each associated with its own state variables. There are many ways to construct a phone entity from these sub-entities each representing a specific type of phone calls. The following two example call models were used by ACPP [2] project in different application domains.

The first model, *PORT_CALL_LEG*, has three type of entities, PORT, CALL, and LEG.

- PORT
  This entity represents the physical CPE associated with the call. This could be a terminal, trunk group members, etc. It assumes each directory number (DN) is tied to a CPE and vise versa. This entity is the root of all other entities. Each CPE has a unique PORT entity shared by multiple calls on the same terminal.
- CALL
  This entity represents the near end of a call. It is a child entity of the PORT. There may be multiple CALL entities associated with a PORT.
- LEG
  This entity represents the far end of a call. It is a child of the CALL entity. There may be many LEG entities associated with a CALL entity as in the case of a Multiway call.

Figure 1(a) to 1(c) shows some examples using this call model. Note that the structure of a call under this model is a single rooted tree.



(a) POTS (b) Call Waiting (c) Three-Way Call

Figure 1. Some PORT-CALL-LEG call model examples.

The PORT-CALL-LEG model is not adequate for a more complicated system where an one-to-one relationship between CPEs and DNs does not exist. (One DN can be shared by two or more

CPEs and vise versa.) This led to the development of the second call model, *LEP-PEP-CALL-LEG model*, which splits the PORT entity into LEP and PEP entities.

- Physical End Point (PEP)
  This entity represents the physical device associated with the call. This could be a terminal, trunk group members, etc. This entity does not have a parent-child relationship. It may be shared by multiple calls on the same terminal.
- Logical End Point (LEP)
  This entity represents the logical device associated with a call. This could be a DN, trunk group, etc. This entity is the parent of a CALL entity. Multiple calls associated with the directory number may share this entity.
- CALL
  This entity represents the near end of a call. It is a child entity of the LEP. There may be multiple CALL entitys associated with a LEP. It is not associated directly with a PEP since a call could be on many terminals if it is a call on a shared DN. If there is a CALL entity then there must be a LEP and a PEP entity.
- LEG
  The same as PORT-CALL-LEG model.

Since a CALL entity has to associate with at least one LEP and at least one PEP, the structure of a call under this model is not a single rooted tree any more. (In fact, it could be extremely complicated.) Figure 2 shows the structure of a POTS call in this model.

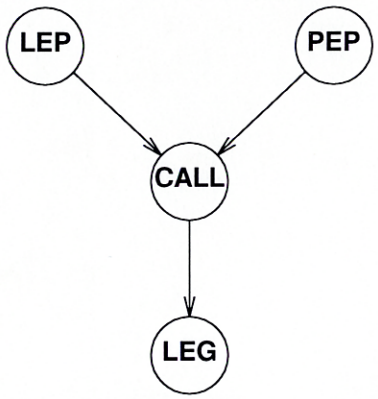


Figure 2. A LEP-PEP-CALL-LEG call model: POTS call.

Note that a complete call model not only defines state space, it also defines events and actions for a FSM.

4.2 A Structured Multi-entity Multi-dimension State Representation Scheme (SMEMD)

A structured multi-entity multi-dimension state representation scheme, *SMEMD*, is shown in Appendix A. Basically, this representation scheme can be obtained by associating some state variables to each of the entities in the call models shown in Section 4.1. The scheme doesn't necessarily assume any call model, however. This representation scheme formalizes the state space implicitly used by many telecommunication systems. This formalization will allow a better

software system to be developed.

5. COORDINATED FINITE STATE MACHINE

This section describes a new computational model that provides several advantages for better feature management as compared with the other approaches discussed in Section 2.1.2. It is an "Feature-As-A-FSM" approach.

5.1 Definition

A Coordinated Finite State Machine (CFSM), a variation of Finite State Machine (FSM), is an abstract machine whose status is represented by a finite number of states and whose behavior is controlled by a finite number of slave FSMs and a master FSM that activates a slave FSM on a given state and a given input. It is demonstrated in Appendix B that a CFSM is equivalent to a regular FSM. Consequently, the CFSM model can support the implementation of all possible features that a FSM can implement. By modeling features as slave FSMs, the CFSM provides the "notion" of features to the call processing programming environment. Therefore, the call processing logic in a feature-rich telecommunication network can be properly modeled as a CFSM. The details is described in Appendix B.

5.2 Properties of CFSM

1. Universal shared state space. (This implies that all state names must be unique.)
2. One single current state seen by all FSMs.
3. No communication between slave FSMs.
4. Limited one way communications between the master FSM and slave FSMs. The only communication between the master FSM and slave FSMs is that, at a given event, the master FSM "select" a slave FSM to serve the event.
5. Once selected, a slave FSM can only control the system within one event. In other words, the master FSM can select any feature to execute at each event regardless which features provide the service in previous events.

Based on this computational model, a call processing software that satisfies all requirements listed in Section 1.4 can be easily constructed.

1. **Symbolic feature management**
   This is contributed by the notion of feature in the model.
2. **Dynamic feature management**
   Same as above.
3. **Independent feature development**
   This is mainly contributed by the third property: no communication between slave FSMs.
4. **Multiple feature invocation**
   This is contributed by the fact that there is only one current state and each feature can only control the system within one event. The master FSM can select any slave FSM at any event regardless which slave FSM was selected in the previous event. Therefore, the call processing logic can execute multiple features interleavely within a call. Combining multiple independently developed features to form a new feature becomes possible.

6. IMPLEMENTATION

No matter how great a model is, it would be useless if a practical implementation plan does not exist. There are many constraints in the development world that may make a great idea useless. It is beyond this document's scope to discuss how difficult it is to establish a real working environment. Although the following high level implementation plan may not be the best plan, it is similar to the plan successfully experimented in the ACPP project [2].

1. Establish a clean interface between the operating system and the call processing logic.
2. Implement a kernel that translates operating system interface into a CFSM environment. In other words, it should understand the syntax and semantic of events and actions of the defined call model. It translates the operating system messages into CFSM events and translates the CFSM actions into appropriate operating system calls. (It is also possible to define a model-independent kernel by defining another interface between each call model and the kernel.)
3. Define an *Application Oriented Language* (AOL) that matches closely to the computational model (the CFSM, in this case).
4. On top of the kernel, implement a *run time execution environment* (RTE) to execute feature codes, which are written in the defined AOL language. Actually, each RTE itself works as a generic CFSM and the feature codes populate it into a specific CFSM.

To implement a specific calling process logic, the following steps should be followed:

1. Define a call model, which consists of state, event, and action spaces.
2. Implement the feature manager and feature codes based on the defined call model using the defined AOL.
3. Compile above entities together with RTE into a monolithic C or C++ program, and then compile into a monolithic executable image.
   Each of this image is a complete CFSM that can handle a half-call of all types. It can be executed as an independent process, or, a thread if there is only one call processing process in the system to handle all calls.

The structure and the relationship of these entities are illustrated in Figure 3.

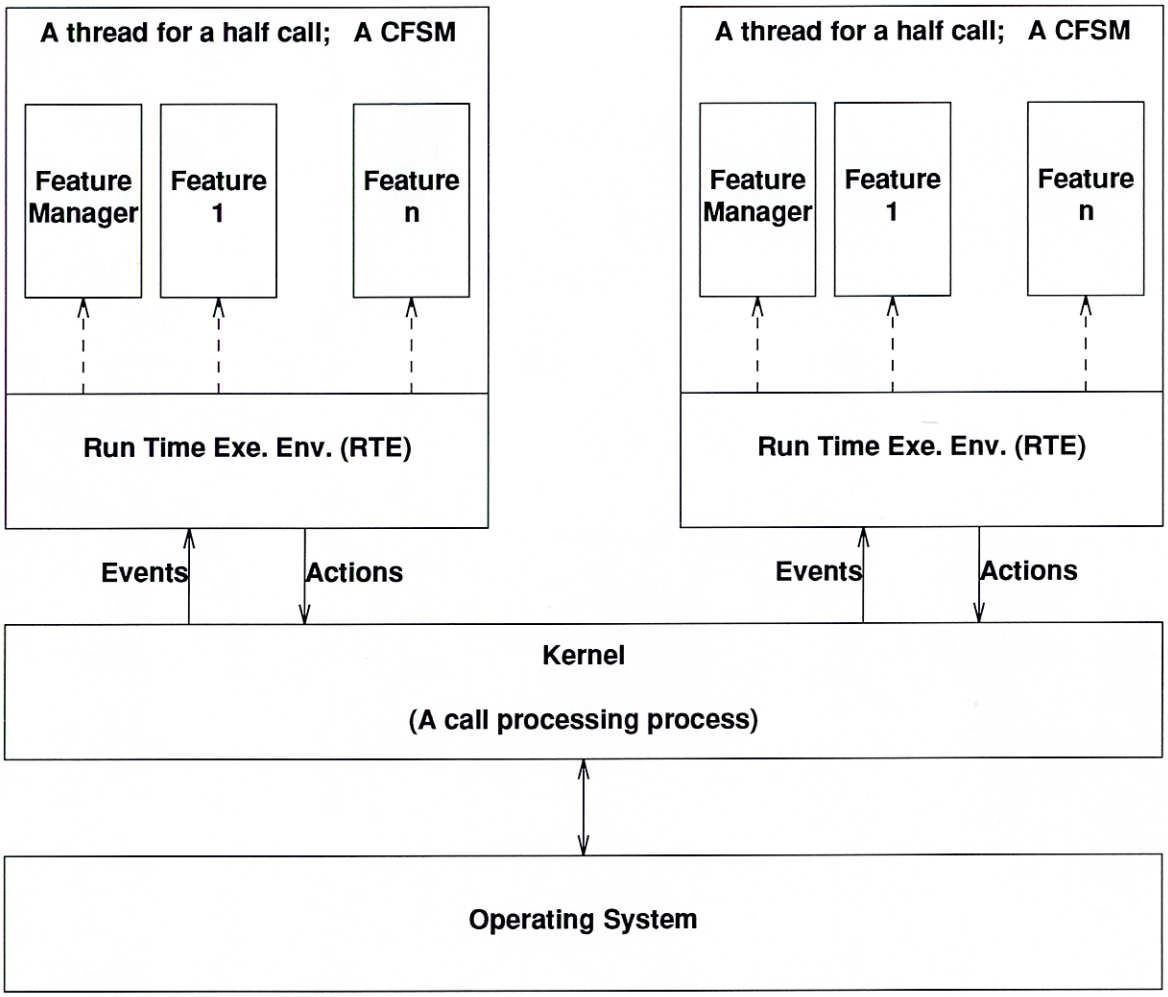


Figure 3. System structure in the implementation Plan.

Theoretically, all these entities (AOL, kernel, and RTE) can be defined and implemented independently. In reality, they had better be designed together under the same development team because no prior experience has been established in defining perfect AOL, kernel, and RTE yet. Working under a single team will have a better chance to succeed.

6.1 Application Oriented Languages (AOL)

One index that can be used to estimate the productivity of a programming environment is the semantic gap between the programming language and the computational model of the target problem. One approach is to design an AOL to directly support the model. The other is to use an object-oriented programming language to create an abstraction that is as close as possible to the computational model. It is not this document's intention to compare these two approaches.

A good AOL not only can improve the programming productivity, it also allows program semantic be easily extracted from source programs. With that capability, a development environment such as a code browser can be easily developed.

The followings are the desired properties of an AOL:

1. **It must match closely to the computational model.**

   Assuming the CFSM is chosen to be the computational model, the AOL must satisfy the following two conditions. First, **it should support the event-driven FSM type of operations** in a way similar to the followings:

   ```
   State: S1
   Event: E3
   Actions:
      Commands
      Commands
      Change_state(S2)
   ```

   Such a code block is referred to as an *action module* in this document. Theoretically, it doesn't have to carry a label. However, it will be more convenient to have a label for every action module.

   Secondly, **the AOL should support the notion of feature.** This can be fulfilled by the followings:

   - Action modules that belong to the same feature should be grouped together and should have a label (name) for the feature.

     The grouping can be done easily if each action module is given a label. The followings is a simple example:

     ```
     Feature  CW { block1, block2, block9, ...)
     ```

     The grouping and the label are the basis to support symbolic feature management.

2. **It must enforce a controlled common state space**.

   This implies that a public module should be designated for model specification including state, event, and action spaces. Any changes made to the model will impact all feature codes. Therefore, it must be well controlled.

3. **Feature codes cannot define public states**.

   This is the consequence of the above property.

4. **It must be able to manipulate features symbolically and dynamically**.
   Once each feature carry a name in the program, the AOL should further allow features be managed by their names.

5. **It must provide appropriate mechanisms to define the public call model based on a particular state representation scheme.**
   The following are basic mechanisms:

   - facilities to define entities, state variables associated with each entity, and the structures of these entities.

- facilities to manipulate the state of a call and the network.

Above properties are basic properties of an AOL. It may need other properties, such as object orientation, to make program easier to develop.

6.1.1 Examples of SMEMD State Manipulation Facilities

Assuming the SMEMD state representation scheme described in Section 4 is adopted, following example facilities are necessary:

- create and destroy entities such as ADD_CALL, ADD_LEG, DELETE_CALL, DELETE_LEG, etc.
- change structures such as MOVE_LEG, SPLIT_LEG, MERGE_LEG, etc.

The execution of these commands must be accompanied by the actual network manipulations either explicitly or implicitly. Programmers need to issue separated commands to manipulate the network if explicit methods are provided. While implicit methods allow a single command to change state and to manipulate the network together.

6.1.2 Examples of Symbolic Feature Management

The followings are examples of symbolic feature management. The first example will allow the feature manager to instruct the system to execute features in the specified order.

```
State: S0
Event: E1
Action:

    Execute_Feature(F1, F2, F3, POTS);
```

The command, *Execute_Feature()*, will execute features, *F1, F2, F3,* and *POTS* for the event *E1* at state *S0* sequentially until either a termination condition is detected (Section 6.2.3) or all features are executed.

We can also implement a *list* data type and store feature names in that list, called the *feature list*. The dynamic feature management property can be achieved by manipulating this feature list. This will be shown in the second example as follows.

```
1. feature_list=(F1, F2, F3, POTS);
2. if ( c1 )
3.    then  feature_list -= F3;
4. endif
5. Execute_Feature(feature_list);
```

In the above example, line 1 defines a feature list; line 3 removes feature *F3* from the feature list; and line 5 executes features according to the order specified in the feature list. In this way, feature *F3* is symbolically disabled in this call.

As we can see from these two examples, features are actually managed symbolically.

6.1.3 FSM Actions and Non-FSM Statements

An action is called a *FSM action* if it is defined in the action space in the call model, a *non-FSM action* otherwise. Examples of FSM actions are: change_state, collect_digit, and send_dial_tone. These actions either change the state of a call or manipulate the network. The definition of FSM imposes some restrictions on the use of these actions. For example, no more than one change_state can be executed in the processing of one event. All other C/C++ instructions and functions are non-FSM actions. FSM definition does not imposed any restriction on their usage.

6.1.4 AOL Design Approaches

There are two approaches to design a language: either design from scratch a complete self-contained language, or extend an existing language such as C++.

The design of system software involves many nitty-gritty things to do. To design a complete self-contained language that can support system software development is nontrivial. The second approach is more practical when time is limited. This document assumes the second approach is taken. However, a careful comparison should be made to determine which approach is better for each particular case.

Designing a language like this is not trivial. A typical example that illustrates the complexity of this task can be found in [3].

6.1.5 Organization of AOL Programs

As mentioned in the previous section, the basic code block of an AOL program is the *action modules* each associated with a triggering condition, (state, event), such as follows:

```
State: S1
Event: E2
Action:
   (action codes)
```

Note that the state will be in the form of boolean combination of state variables if the SMEMD state representation scheme is used. The action codes can be expressed in a conventional language such as C or C++.

The structure of an AOL program that is to handle a half-call with all possible features may be as follows:

1. A call model specification defining state, event, and action spaces.
2. A feature manager (master FSM) containing a group of action modules.
3. Some features (slave FSMs) each containing a group of action modules.

The triggering conditions of action modules must be unique within a feature, and can be the same in different features (where a feature interaction might occur). The syntax of action modules in the feature manager is similar to that in feature codes. The only difference is that the feature manager is always executed first in the processing of each event and it can execute some privileged commands such as *Execute_Feature()*.

One major responsibility of the feature manager is to specify the logical execution of features and get it executed. For instance, the first example in Section 6.1.2 will cause all action modules in features *F1, F2, F3,* and *POTS* that have a triggering condition of (*S0,E1*) be executed by the RTE one after another until either a termination condition is detected (Section 6.2.3) or all modules are executed.

The name of an action module is not an absolutely required information. However, it makes the program organization much easier. Once all action modules are designed, the structure of the entire package can be defined symbolically. For instance, feature packages can be defined in the following hypothetic syntax:

```
Package P1 { F1, F2, F3, ..., Fn }
Package P2 { F1, F4, F5, ..., Fm }
Feature F1 { M1, M2, M3, ..., Mj }
Feature F2 { M2, M4, M6, ..., Mk }
```

Further, as shown in the above example, code reuse becomes very trivial.

6.1.6 Summary

Simply speaking, the AOL language provides a model for specific operations as well as reorganizing feature codes, within the structure of a conventional programming language, into a more controlled and organized fashion. It provides the following advantages:

1. the state space can be easily defined and manipulated;
2. the state space is more visible and sharable;
3. the state space is protected from arbitrary modifications; and
4. the notion of features is realized so that symbolic and dynamic feature management is possible.

6.2 Run Time Execution Environment (RTE)

There is a run time execution environment, which could be implemented as a process, handling each half call. The run time execution environment works like an operating system receiving all external events from the underlying system, and invokes/calls appropriate action modules to execute. It also has to take the commands from action modules and deliver to the system to execute. It has to memorize and update the "current state" of the half call according to the "change_state" commands given by features. Based on the combined condition of state and event, it can select appropriate action modules to execute. Note that each feature is not implemented as a process. It only offers suggestions to the RTE what to do. RTE can make its own choice to decide which features to be included in the feature list and in what order.

In a nutshell, the RTE works like a generic Coordinated FSM and the AOL program actually specializes it.

Like a typical operating system, a RTE has many nitty-gritty things to do. Many of them are implementation dependent. The complete details can only be figured out in the implementation. The following section will list some properties that are implementation independent.

6.2.1 Essential Properties

A RTE must have following properties to carry out a CFSM implementation:

- The master FSM will always be executed before any slave FSM.
- At any given event, only one slave FSM is selected as the serving FSM by the master FSM.
- The serving FSM can only control the system for the current event.
- Facilitate independent feature development

6.2.2 Other Desired Properties

For practical purpose, the following properties are desirable in implementing a RTE. They are not essentially required by the CFSM. However, they will make the system easier to implement.

- Simple and stupid

  For the following reasons, the most popular common wisdom in designing a complex system software is to make the logic *simple and stupid*. A system software usually have to supports a very "dirty" environment that it is very difficult, if not impossible, to have a "clean" design. The designers usually have to wait until very late in its life cycle to get a very clear idea how to make it clean although it may not have that luxury to actually clean it up. It is doomed to fail if a system software is designed to be "smart" at the very beginning. For example, the most popular scheduling policy in an operating system is a very simple round-robin algorithm, instead of more efficient but more complicated algorithms invented by scheduling experts. The most popular protocol for database concurrency control is the strict two-phase locking protocol, instead of more efficient optimistic based or time-stamp based protocols. Nobody dares to adopt any "advanced" protocols invented by database researchers. (There are thousands of such protocols).

  Therefore, the most important rule-of-thumb in an operating system design is "keep it simple and stupid". The run time execution environment for CFSM should follow this common wisdom.

- Provide execution status monitoring facilities to the feature manager

  The feature manager has to watch for the execution of each feature and makes decisions based on the status of their execution. An example will be shown in Section 6.2.3.

6.2.3 An Implementation Example

When an event arrives, the RTE will identify all action modules whose triggering conditions are satisfied and execute them one by one. The feature manager, which controls the execution order of features, will be always executed first.

- One major responsibility of the master FSM is to determine the order of feature execution.
- All slave FSMs (features) in a given event are executed by the master FSM in a daisy chain fashion. The preceding FSMs have higher priority than succeeding FSMs to become the serving FSM.

(In this way, one slave FSM is "selected" by the master FSM, which actually only sets their precedent order.)

- All FSMs can execute non-FSM actions. Only the selected FSM (the serving FSM) can execute FSM actions.

Based on the execution status, RTE will make the decision out of the following choices after the execution of each action module:

1. **stop**, if some FSM action is executed;
2. **continue**, otherwise. This includes the situation where no action module is actually executed.

In this way, only one slave FSM can be selected as the serving FSM for each event. This characteristic allows a RTE to overcome the dilemma that it must be stupid by not knowing the detailed implementation of all features and must be smart enough to "select" a serving feature.

The following example, which has one action module in a CWC feature and one in the feature manager, shows how a feature can "control" another feature without knowing the implementation details. Specifically, Call-Waiting-Cancel is used to disable Call-Waiting feature temporarily. This is a very desirable feature when a data call is made over an analog line and the invocation of the CW feature may corrupt the transmission of data.

```
Feature_Manager_Module_CWC {
      State: S0
      Event: Feature_Request
      Action:

        feature_list = (CWC, CW, POTS);
        Execute_Feature(feature_list);
        if ( CWC is executed)
        then
           feature_list -= (CW);
        endif
}

Feature_CWC_Module_1 {
      State: S0
      Event: Feature_Request with feature_type == CWC
      Action:
            ......
            state_change(....)
}
```

When an event *Feature_Request* arrives (the event must carry the feature type information), the RTE executes the module *Feature_Manager_Module_CWC* in the feature manager first, it in turn sets up the feature list and executes the *Execute_Feature* command. The module *Feature_CWC_Module_1* in feature *CWC* will be executed first since it's triggering condition is true. The RTE will temporarily exit the execution of *Execute_Feature* and examine the status. Since it makes a state change, the RTE will really stop the execution of *Execute_Feature*. It continues to execute the rest of the commands in *Feature_Manager_Module_CWC*. In the

execution of

```
if ( CWC is executed),
```

the RTE will find that feature CWC is executed so that it will execute the *then* segment

```
feature_list -= (CW).
```

In this way, Feature *CW* is removed from the feature list and will not be invoked in the succeeding events unless it is put back into the list. In this way, a feature can actually "control" another feature without knowing the implementation details.

Figure 4 depicts an example to show how a compiler translates a call processing logic written in an AOL into a corresponding C program. The control flow can be easily seen from the picture.

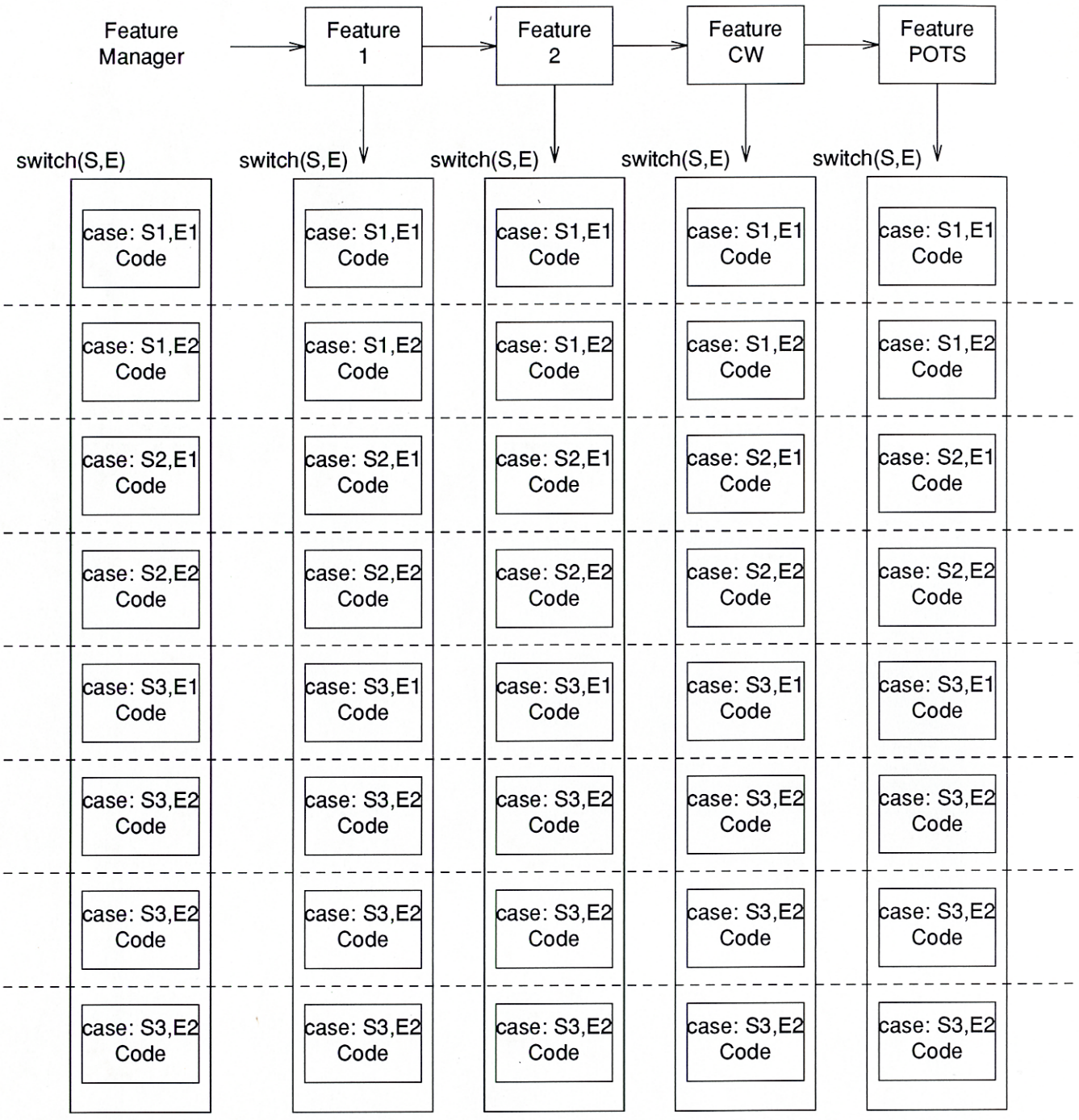


Figure 4. An example of compiled AOL program showing the control flow.

Each block is an action module. All action modules of a feature are grouped together to form a C/C++ function, referred to as as a *feature function*, pointed by a function pointer. Within each feature function, there is a switch statement switched on the (state, event) pair. Each module is

compiled into a case in the switch statement. All modules of the feature manager are also grouped into a switch statement in the same way. However, it is always executed by RTE first in each event. In the execution of the statement *Execute_Feature(feature_list)*, RTE will have to set up a list of pointers pointing to the feature functions. The content of the list is specified in the *feature_list*. In this way, features are executed one by one in the specified order. **The feature manager can manage features by manipulating the feature list. Therefore, the feature management is symbolic and dynamic.**

It is also very easy to implement customer specific feature interaction resolution. Each customer can save his/her interaction resolution decision on the database and the feature manager can populate the feature list in real-time based on that information.

From the above description, we can see that **a RTE can be implemented using a very simple and stupid logic**. The RTE in the example doesn't have much intelligence other than taking the event and following a very simple execution sequence to execute feature codes.

6.3 Feature Interaction Detection for Event Contention

Once the call processing code is written in an AOL language, it is very easy to design a software to sort these action modules by their triggering conditions. All modules with the same triggering condition will actually contend for the same event at the same state. There is a very high possibility that these features interact to each other. Therefore, the feature interaction of this type can be automatically detected by the software.

6.4 Feature Interaction Detection for Data Contention

The CFSM model does not automatically provide any detection capability for the data-contention type of feature interactions. However, the following *interaction variable* technique can be easily incorporated into an AOL such that it can provide a semi-automatic detection capability.

Some special shared variables, called *interaction variables*, can be **designated** for information repository; and a feature can save shared information in these variables. The system can then detect any interference made by other features on these variables based on their read-write relationship.

In the case of OCS-CF interaction, one can force designers to use an interaction variable to store the destination number and have all programs that need to access or update the destination number use that variable. A potential feature interaction can be detected by checking the read-write relationship among these features.

This technique is considered semi-automatic rather than fully automatic because there must have some coordination between feature developers. A software development process must be established to enforce features to use the same variable to save the same information such that their interactions can be detected.

Note that the encapsulation concept in object oriented programming paradigm is not adequate for this purpose. The interaction variable concept encourages (enforces) variable sharing. On the contrary, the encapsulation concept prevents the data items in an object from being direct updated by other objects. It actually discourages variable sharing.

7. FUTURE WORK

Following is a list of additional work that would help in better understanding the scope of the feature management problem and the feasibility of the proposed framework.

- Expand the classification of the different types of feature interaction problems and develop an approach for systematically addressing them.
- The implications of the proposed model in a distributed environment needs to be explored. There are potential synergies between this work and the Platform Operating System work.
- Since the Coordinated FSM model relies on the definition of a state space, it is important to address the stability of the model or other implications when new states need to be incorporated or a modification to the call model is required.
- Although the examples presented are call processing oriented, this model may be applicable in other domains such as operations. In general, event-driven systems could potentially take advantages in using this model.
- Additional work to identify an automatic detection for data contention is required.
- Performance analysis of the proposed model and comparison with other competing models is required.
- Assess the implications to the users of a development environment in terms of the required tools, user interface, level of programming knowledge, and other factors that would help determine the classes of users that could effectively make use of it.
- Mechanism for automatic detection of feature interaction and the generation of rules for resolution based on the proposed model.
- Refine the plan for implementation and assess the magnitude of the work required.
- Applicability for use in call models of emerging services.

8. ACKNOWLEDGEMENTS

The author would like to thank many people who give very precious comments. A special thank goes to Adolfo Lagomasino who not only provided the leadership to the project, but also provided many good ideas that are incorporated in this document.

Reference

[1]. E. J. Cameron, Nancy Griffeth, Yow-Jian Lin and et al, "A Feature-interaction Benchmark for IN and Beyond," *IEEE Communication Magazine*, Vol. 31, No. 3, pp. 64-68, March 1993.

[2] J.J. Driscoll, E. F. Knappe, and et al, "Advanced Call Processing Platform (ACPP) Architecture DS - DS5U0011.00A", *AT&T IM*, 55661-920821-02IM.

[3] Francis Leung and et al, SAIL User's Manual.

[4] Pamela Zave, "Feature Interaction and Formal Specification in Telecommunications", *AT&T TM*, BL011261-930415-17TM.

Appendix A: A State Representation Scheme (SMEMD) for Call Processing

1. Introduction

Modern Electronic Telecommunication Network usually can support a wide variety of service features such as Call Waiting, Call Forwarding and Multiway Call. A good state representation of a phone call that is rich in state descriptive power, easy to manipulate, and easy to understand becomes a necessity for switching system software development. A structured multi-dimensional multi-entity state representation scheme is presented here. The same scheme is adopted in SAIL Programming Language [3], an Application Oriented Language for 5ESS call processing.

A telephone (CPE) in a complicated telephone switch such as 5ESS may involve in a very complicated situation that cannot be represented by a simple state representation scheme. For example, a CPE may be talking to more than one other CPE (Multiway Call) while some other incoming phone calls are held on the same phone line waiting to be answered (Call Waiting). It is further complicated in an ISDN environment that a CPE may associated with more than one DN (phone number) or a DN may be associated with more than one CPE. It may require numerous different states to represent all possible situations a phone call may be involved. The traditional finite state machine model used in a switch such as 5ESS may have the following overly simplified state space:

```
NULL
ALERTING
COLLECTING_INFORMATION
ANALYZING_INFORMATION
SELECTING_ROUTE
CALL_PROCEEDING
WAITING_FOR_ANSWER
HUNTING_FACILITY
T_ACTIVE
O_ACTIVE
```

For the sake of discussion, these states are referred to as PIC (Point In Call) states. Obviously, this small state space is far from sufficient to cover all the situation mentioned above. To cope with this problem, many implicit substates are created in the forms of local/global variables, data items in database, etc. The software system becomes very difficult to understand and maintain. A better structured state representation scheme will improve the software quality significantly.

2. Multi-dimensional State Representation

The state space that can cover all possible situations may consists of millions of states. (Actually, it can be easily shown that 2^{25} is a very trivial lower bound for 5ESS.) Multi-dimensional state representation becomes a necessity.

As a matter of fact, those local variables and database items that split a PIC state into many substates in 5ESS can be viewed as implicit dimensions and PIC states can be viewed as another dimension. Thus, a multidimensional state representation is already in use in 5ESS system software only that the state declaration is informal, implicited, uncontrolled, and undocumented. A better software framework should make state declaration a well controlled and explicit component of the system software.

3. Multi-entity State Representation

In general, a phone call refers to that two (or more) parties, the calling party (near leg) and the called party (far leg), are communicating to each other over a communication channel. The status of a call must constitutes of the status of all involved parties. One possible way is using multi-entity state representation scheme, in which, each involved party is represented as an entity in the state. The entities are not necessarily representing a physical device or a process, however. For instance, a DN (phone number) can be an entity. An entity to a state is a process to a computer.

4. Structured State Representation

Once entities are identified in a state representation, relationship among entities also carry important information. A state representation must be able to reflect these relationships in the state. For example, the physical parties involve in a three-way call are the same as those involve in a call waiting call when only one incoming call is on held. However, these two situations are quite different. The state representation must be able to distinguish these two situations in clearly different states. This can be achieved by adding the relationships into the state representation scheme. Therefore, a structured state representation scheme is necessary.

5. Example: PORT-CALL-LEG Model

The following example assumes PORT-CALL-LEG call model is used where there is a one-to-one correspondence between a CPE and DN. A CPE (and the associated DN) can be represented as a "PORT" entity. Each far leg is represented by a "LEG" entity. Each near leg is represented by a "CALL" entity. A line between a CALL and a LEG means they are in the same communication channel. A line between a PORT and a CALL means the call is occurring on that PORT.

Each type of entities can be associated with any number of state dimensions. In Figure A.1, dimensions cwtPIC and cwoPIC are associated with the PORT entity denoting the active/inactive of call-waiting-originating and call-waiting-terminating features. Dimensions numLEG, mwPIC, and on_hold are associated with a CALL entity denoting the number of LEGs associating with it, the status of the Multiway feature, and the oh_hold status of a call. A dimension "PIC" is associated with a LEG denoting its PIC status.

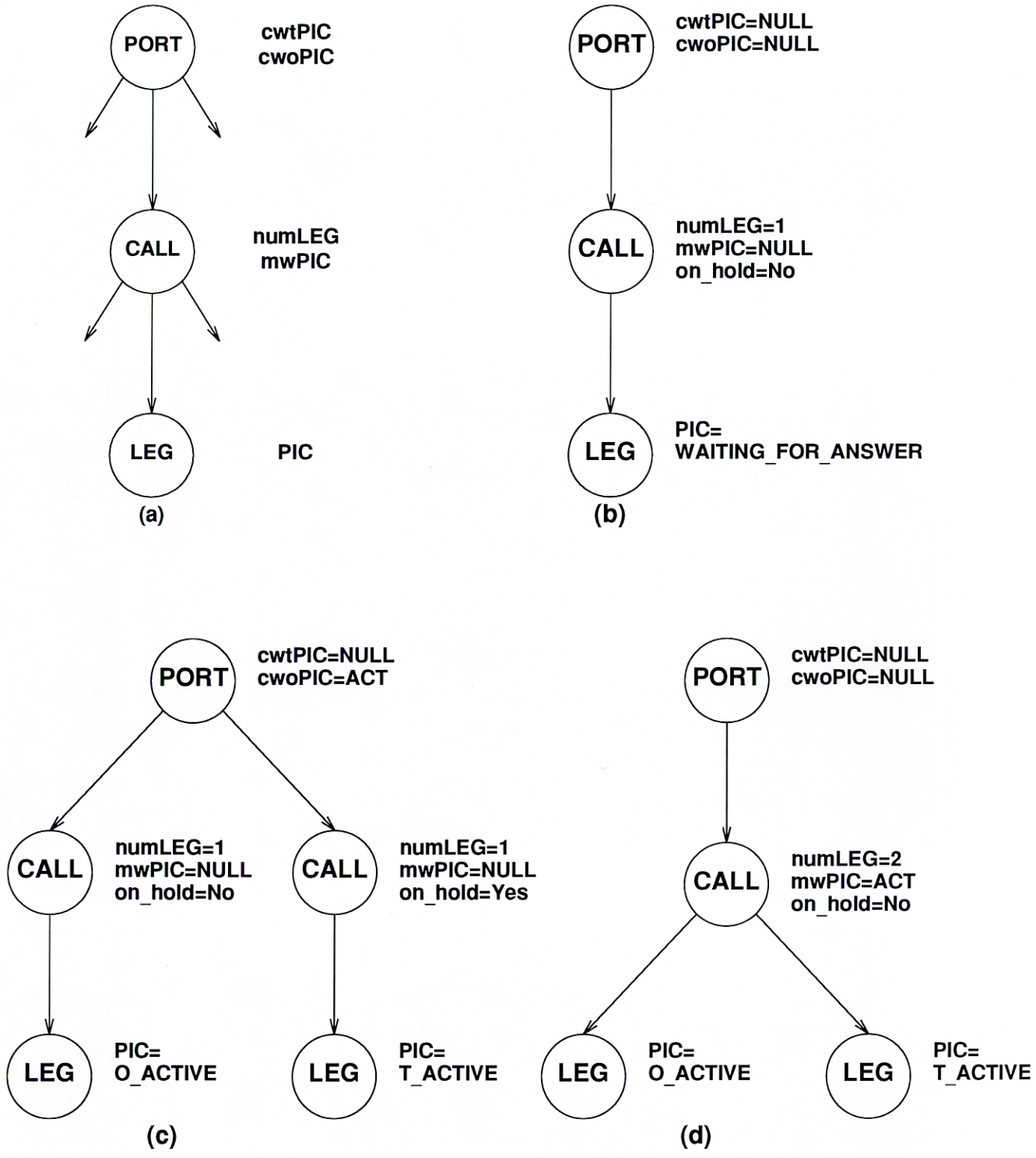


Figure A1: (a) PORT-CALL-LEG model; (b) POTS; (c) Call Waiting; (d) Three-Way

Appendix B: A Coordinated Finite State Machine Model for Switching System Call Processing Logic

1. Introduction

Modern switching systems are required to offer the flexibility of adding new service features, such as the Call Waiting and Call Forwarding features, in a timely and cost-effective fashion. Due to the complexity of call processing logic, this requirement presents a challenge to the switching system software design and development. In the early days, when a switching system supports only a handful of features, the call processing logic can be adequately modeled as a finite state machine; that is, if a "call" is waiting in some state and a signal is received, a relevant transition is fired according to the current state and the current input signal. Nowadays, a telephone switch may have to support several hundreds of features, (although a specific switch seldomly requires that many features,) each trying to control the switch in its own way. It is nontrivial to manage these features including addition, deletion, activation, deactivation, and feature interaction resolution. Feature interaction problem refers to the possible conflict between different features that are interested in the same incoming signal. Unfortunately, the traditional FSM model is inadequate for feature management since a FSM is atomic and the notion of features has to be represented implicitly within a FSM. It requires a surgical treatment to the logic at the low level software code to manage this several hundreds of features. This article proposes a variation of FSM, Coordinated Finite State Machine, to model the call processing logic for a better feature management.

A good software design framework that allows features to be easily managed can be created by using a proper CFSM model together with an application oriented programming language which matches closely to the desired CFSM model.

Nevertheless, CFSM does not limit itself to call processing only. It can be used in a variety of applications.

2. Finite State Machine Model

A Finite State Machine saves the status of a machine, which may be an abstraction of a control logic, in the form of "state" and responses to the external input (events) according to the current state and the logic embedded in the machine. Once the logic is implemented, the response of a FSM is completely determined by the current state and the current input. The behavior of a switching system required by the most basic telephone service, the Plain Old Telephone Service (POTS), can be easily modeled as a FSM. For every event generated by the user, such as off-hook, there should be a proper response made by the switch.

The formal definition of a FSM is as follows:

A *finite state machine* is a sextuple $<I, O, S, s_0, f, g>$ where

I = a finite set of input symbols
O = a finite set of output symbols
S = a finite set of states
$s_0 \in S$ = the initial state s_0

and f and g are transition function and output function respectively:

f: $I \times S \rightarrow S$
g: $I \times S \rightarrow O$

3. A Coordinated FSM Model

A CFSM is an abstract machine whose status is represented by a finite number of states and whose behavior is controlled by a finite number of slave FSMs and a master FSM that activates a slave FSM on a given input. A CFSM actually decomposes a FSM that models a call processing logic into a master FSM corresponding to feature management logic and a finite number of slave FSMs with each slave FSM corresponding to a feature. The formal definition is as follows:

A *coordinated finite state machine*, an eight tuple $<I, O, S, s_0, M, m, F, G>$,
is a finite state machine $<I, O, S, s_0, f, g>$, where

I = a finite set of input symbols
O = a finite set of output symbols
S = a finite set of states
$s_0 \in S$ = the initial state s_0
$M = \{1, 2, \ldots, n\}$
m = a select function
$F = \{f_1, f_2, \ldots, f_n\}$, a finite set of transition functions
$G = \{g_1, g_2, \ldots, g_n\}$, a finite set of output functions

and m, F and G are:

$$m : I \times S \rightarrow M$$
$$f_i : I \times S \rightarrow S, \quad i = 1, 2, \ldots, n$$
$$g_i : I \times S \rightarrow O, \quad i = 1, 2, \ldots, n$$

and the relationship between f and F (g and G), for input i and state s, is:

$$f(i, s) = f_{m(i,s)}(i, s)$$
$$g(i, s) = g_{m(i,s)}(i, s)$$

f can be viewed as the aggregated transition function and g as the aggregated output function defined by F and G respectively. The master FSM is $<I, M, S, s_0, \varnothing, m>$, and the slave FSMs are $<I, O, S, s_0, f_i, g_i>$, $i = 1, 2, \ldots, n$. All FSMs share the same state space and the same current state. Given any input at any state, the master FSM activates a slave FSM, controlled by m, and the activated slave FSM determines the current output and next state, which is shared by all FSMs.

It can be easily proved that a FSM can be emulated by a CFSM and vise versa. Therefore, the class of CFSM is equivalent to the class of FSM, where the class of a machine is its all possible instances.

Theorem 1: *The class of CFSM and the class of FSM are equivalent.*

proof:

By definition, a CFSM can be emulated by a FSM. A FSM can be emulated by a CFSM by letting $M = \{1\}$, $m(*, *) = 1$, $f_1 = f$, and $g_1 = g$.

□

Nevertheless, the CFSM allows features to be modeled independently from each others, and feature interactions to be managed at a higher level (symbolic feature management).

CONTENTS