The Lightweight Flow Engine:
A Model for Rapid Development and Emulation of Telecommunication Services

Thomas L.P. Wood

Members of the dissertation committee:

Prof. A.V.	Aho, PhD	Columbia University, NY
Prof. dr ir H.J.	Bos	Vrije University, Amsterdam
dr ir H.G.P.	Bosch	Cisco
Prof. dr H.	Brinksma	University of Twente
Prof. dr ir P.J.M.	Havinga	University of Twente
Prof. dr S.J.	Mullender	University of Twente (promotor)
Prof. dr R.J.	Wieringa	University of Twente (chairman and secretary)

Copyright © 2013, Thomas L.P. Wood, Colts Neck, NJ, USA

All rights reserved. No part of this book may be reproduced or transmitted, in any form or by any means, electronic or mechanical, including photocopying, microfilming and recording, or by any information storage or retrieval system, without the prior written permission by the author.

ISBN 978-90-365-0371-6

THE LIGHTWEIGHT FLOW ENGINE:
A MODEL FOR RAPID DEVELOPMENT AND EMULATION OF TELECOMMUNICATION SERVICES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 13 september 2013 om 12:45 uur

door

Thomas Leslie Peter Wood

geboren op 13-01-1956
te Ridgewood, New Jersey, USA

Dit proefschrift is goedgekeurd door
Prof. dr S.J. Mullender (promotor)

Table of Contents

The Lightweight Flow Engine:....i
Prof. dr S.J. Mullender (promotor)....iv
Acknowledgements....ix
Summary....xi
Samenvatting....xiii
1 Introduction....1
Problem Statement....8
2 A Brief Overview of LiFE....11
2.1 Service Logic....13
2.2 Resource Discovery....16
2.3 Reliability/Availability Features....17
3 Background & Related Work....21
3.1 Linda (Gelernter)....21
3.2 Unified Modeling Language Based Approaches....22
3.2.1 UML Background....22
3.2.2 Specification & Description Language – Real Time (SDL-RT) 24
3.3 Estelle....25
3.4 Communicating Finite State Machines 26
3.5 Petri Nets....26
3.6 RoseRT....27
3.7 BPEL....28
3.8 Web Services Flow language....29
3.9 SCXML 30
3.10 CCXML....32
3.11 Opensips....33
3.12 Esper....33
3.13 Cosmogol 34
4 An informal description of LiFE....35
4.1 The LiFE model....36
4.1.1 The Flow Graph Structure....37
4.1.2 The Global Name/Value List....37
External Service Definitions....38
Internal Service Definitions....39
4.2 Graph States....42
4.2.1 Graph State Structure....43

4.3 Messages and Exceptions....46
4.4 Arc Actions....47
4.5 Arc Name/Value Pair Directives 49
4.6 String Manipulation....50
4.7 LiFE and Turing Machines....51
4.8 Influenced by Data Flow....57
4.9 Lazy Linking....58
4.10 The Plug-In Architecture....63
5 The computational model of LiFE*....67
5.1 A Model for Specifying Network Services....68
5.1.1 Informal Overview....68
5.1.2 Formal Model....69
Labeled transition graph....69
5.1.3 Asynchronous messages and concurrent threads....70
Data values and expressions....73
Timeouts and exceptions....76
5.2 Operations Expressible in the Model....77
5.2.1 Data Types....77
5.2.2 Thread Types....78
5.2.3 Dynamic Extensibility....78
5.2.4 Graph Calls and Returns....79
5.3 Conclusion....79
6 The LiFE execution engine architecture....83
6.1 Overall System Architecture....83
6.2 LiFE and failover....85
6.3 LiFE and fractured state....86
6.4 Just-in-Time Element Management....89
7 Use Cases....97
7.1 Image Processing Support....98
7.2 Emulation of Telecommunication Services....101
8 Performance Architecture....111
8.1 Design Principle....111
8.1.1 Use of Hash Functions for Lookup....111
8.1.2 Handling Timers....114
Use of quantized timers on a timer wheel....115
8.1.3 Prioritizing Message Processing....117
A Summary of the Prioritizing Message Flows Method…119
Managing Overload within the Flow Engine....119

Late everything....121
Even Later Binding....123
"Methods for Update in Place" 124
Detailed Description of the Invention....124
9 Performance Evaluation....127
9.1 Comparing Performance of Flow Engine Architectures....130
9.1.1 Timer Performance....140
9.1.2 Minimizing the number of malloc/frees – Performance Impact....142
9.2 Evaluation of graph structures....145
9.3 Comparing to Other Systems....146
9.4 Conclusions....149
10 History of LiFE....151
10.1 First evolution: workflow engine....151
10.2 Second evolution: media gateway controller....154
10.2.1 Adding Multi-protocol support....155
10.2.2 Standard yet Incompatible....157
10.2.3 Implementation Notes....160
11 Conclusions....163
12 Bibliography....167
A US Patent 20090089770....173
A.1 Abstract....173
A.2 Claims....174
A.3 Description....177

Acknowledgements

I would like to thank all of the members of the committee for their advice and support. At this point it is usual to thank the *promotor* for his or her advice and support. The words seem inadequate to express my deep appreciation and the gratitude that I feel for my promotor, Sape J. Mullender. His support, wisdom and technical prowess have not only guided me through this process, but has set an example of behavior that will serve as a personal benchmark going forward. In addition, I'd like to acknowledge the encouragement provided by Peter Bosch. His support was instrumental to my pursuit of this degree. I have received support and encouragement from my colleagues and managers at Bell Labs during the entire PhD process. I want to thank my entire Bell Labs family, and Markus Hofmann, Vassilka Kirova, John Shamilian and Ramesh Viswanathan in particular.

None of this would have been possible without the support of my family: Rob, Grant and especially my wife and eminent proof-reader, Pam. Grant and Rob continue to be a constant source of joy and inspiration, but without Pam's love, strength and gentle encouragement, I'd still be on the first chapter.

Summary

This thesis explores a method to model network services with particular emphasis on telecommunication control-plane applications. This class of applications is *stateful,* that is, the application's correct behavior is predicated upon the processing of previous events. Most telecommunication infrastructures are realized by a number of independent organizations, relying upon *standards* to ensure interoperability. By their very nature, most standards are open to some degree of interpretation, leading to issues during use. One of the primary motivations for the work described in this thesis has been to be able to rapidly realize and customize control plane applications described using models (message-charts, state-machines) contained in industry standards. These efforts have resulted in the creation of the Lightweight Flow Engine (LiFE), an execution environment which accepts a domain-specific language designed to correspond well to modern control plane applications. While LiFE provides a general and flexible solution to the development of diverse services ranging from Voice over IP (VoIP) call control to bill-reconciliation processing, the resulting implementations have proven to be highly efficient and portable; additionally, services developed using the LiFE system are dynamically extensible and reconfigurable.

In the thesis, several aspects of LiFE will be examined, including the software and system architecture of the flow engine, performance of key portions of the architecture, as well as the domain-specific language used to implement network functions. Two use cases will

be presented as examples of the range of applications that can be described and realized efficiently using LiFE.

As part of this thesis, a formal model is presented that distills the theoretical underpinnings of the domain-specific call-flow graph language described. The model, formulated as a labeled transition graph in which high-level programming constructs are ascribed to transitions, smoothly unifies a number of computational paradigms that have remained separate so far. First, it can express both the asynchronous characteristics of interactive systems as well as the synchronous characteristics of reactive systems. Second, the data flow is explicitly delineated along message send and receive operations, thus assimilating the spirit of data-flow languages. The combination of asynchronous and synchronous features, as well as an explicit structure on the control and data flow seems to correspond well with the way network services are typically specified.

Samenvatting

Dit proefschrift onderzoekt een methode om network services te modelleren met speciale nadruk op *control-plane* toepassingen in de telecommunicatie. Deze klasse van toepassingen is *stateful*, dat wil zeggen, het correcte gedrag ervan is afhankelijk van de verwerking van voorafgaande gebeurtenissen — *events*. Telecommunicatie infrastructuren worden meestal gerealiseerd door samenwerking van een aantal verschillende onafhankelijke organisaties, die gebruik maken van standaarden om succesvolle communicatie tot stand te brengen. De eigenschappen van standaarden brengen vaak een zekere mate van dubbelzinnigheid met zich mee. Een van de voornaamste drijfveren voor het werk in dit proefschrift is om in staat te zijn snel en aanpasbare control-plane applicaties te bouwen, gebaseerd op modellen (in de vorm van *message charts* en *state machines*) gedistilleerd uit *industry standards*. Het resultaat van dit werk is de Lightweight Flow Engine (LiFE), een applicatie-executie omgeving die gebruik maakt van een domein-specifieke taal ontworpen om *control-plane* toepassingen goed te kunnen beschrijven. LiFE levert niet alleen een flexibele oplossing voor het ontwikkelen van een brede klasse van services — van Voice-over-IP (VoIP) tot factuurafhandeling — de resulterende software is ook bijzonder efficiënt en overdraagbaar (*portable*); daar komt bij dat services gecreëerd met LiFE dynamisch uitbreidbaar en reconfigureerbaar zijn.

In dit proefschrift zullen een aantal aspecten van LiFE worden behandeld, waaronder de software- en systeem-architectuur van de *flow engine*, de prestaties van belangrijke onderdelen van de architectuur en de domein-specifieke taal waarmee netwerk-functies worden beschreven. Twee gebruiks-casussen zullen worden gepresenteerd als voorbeeld van de breedte die bestreken kan worden door applicaties in LiFE.

In dit proefschrift wordt een formeel model gepresenteerd waarin de theoretische fundering van de domain-specifieke call-flow-graph taal wordt beschreven. Het model, geformuleerd als een *labelled transition graph* waarin programmeerconcepten gekoppeld worden aan transities, verenigt verscheidene computationele paradigma's die tot nu toe onverenigbaar leken. Ten eerste kan het zowel zowel de asynchrone karakteristeken van interactieve systemen beschrijven als de synchrone van reactieve systemen. Ten tweede wordt het data pad expliciet neergelegd aan de hand van de ingaande en uitgaande *messages* waardoor er een *data-flow*-achtige taal ontstaat. De combinatie van asynchrone en synchrone kenmerken, plus de expliciete structuur van *control-* en *data-flow* lijkt goed overeen te komen met de manier waarop netwerk services meestal worden gespecificeerd.

1 Introduction

The implementation and development of applications and services can be significantly simplified and improved through the use of *domain-specific languages* or formalisms. Domain-specific languages provide primitives that correspond to the most important computational elements around which the particular class of applications of interest are typically structured. The domain-specific language should be designed to enable a programming paradigm that closely mirrors the style in which the target applications are designed or described. By thus allowing the developer to focus on the core logic of the application, significant reductions in development time and in implementation errors can be achieved. The tight correlation between the code and the application description also allows for easier software maintenance and tracing of any bugs and their causes. Finally, by automating the code generation or execution of the language for different platforms, applications can be easily ported to new platforms with increased likelihood of interoperability. In this monograph, a new formalism is presented that has been developed for the application domain of **network services**.

As opposed to systems that only compute data outputs from data inputs, a network-service application must interact constantly with an environment that sends it traffic packets and control messages, and is itself made of concurrent parts (e.g., multiple threads). Following the terminology introduced in [66], such systems fall into two distinct classes—*interactive systems*, and *reactive* or *reflex systems*.

In interactive systems, clients ask for accesses or resources that the system grants or allocates if and when possible. The system is the leader of the interaction and the clients wait to be served. Message handling is therefore typically asynchronous. Examples of this class are operating systems and distributed systems

In reactive or reflex systems, the system's role is to react to external stimuli by providing outputs in a timely way, with the environment as the leader of the interaction. Most often, clients cannot wait and the pace of interaction is therefore determined by the environment rather than the system. Prominent examples in this class are airplane or automobile control systems, signaling protocols, signal processing, and synchronous circuits.

Existing models and formalisms fall squarely into one of the two categories. Reactive systems are naturally modeled as finite-state automata, and extensions in the form of input-output actions such as Moore and Mealy machines [36]. Programming languages for reactive systems are synchronous languages such as Esterel [10], Lustre [30], and Signal [43]—these are based on the perfect synchrony abstraction in which processes are considered conceptually to perform their computations in zero time. Programs in such languages are then compiled into finite-state automata [11], [15]. Interactive systems can be elegantly described using process calculi such as communicating sequential processes (CSP) [33], calculus for communicating systems (CCS) [48], and the π-calculus [49], [50]—these are based on programming primitives for concurrency, sending and receiving messages, and scoping constructs for variables and names. The formalisms for the two classes differ significantly in the style of descriptions that they support. Automata-theoretical models for reactive systems are graphical and based on the explicit identification of control flows in terms of states and the flow between them. On the other hand, specifications in process calculi are closer to high-level programming languages (such as C or Java*) in that the control flow and data flow is an implicit effect of the semantics of the various programming constructs rather than being explicitly identified in the program itself. The two formalisms also differ in the extent to which they support data processing. Reactive system models are much less data intensive. Data-manipulation operations are less powerful and

the data driving the execution belongs to a simple variety of possibilities, such as, for example, a small number of event types (as in signaling protocols) or binary values (as in synchronous circuits). The data processing cannot, inherently, be very complicated because, if the data computation takes a very long time, then it would no longer be consistent with the perfect synchrony assumption.

To my knowledge, there is no existing formalism that encompasses the characteristics of both reactive and interactive systems in a smoothly unified way. Clearly, network services have components of both kinds—e.g., message processing has to be asynchronous and buffered and some management-related processing can be delayed, but certain timeouts have to be respected to ensure line-speed packet processing. Moreover, neither class of formalism is particularly well-suited to describing or programming network services.

To illustrate this, consider implementing a protocol standard which is a simple but representative example. Protocol specifications define the format of protocol data units (PDUs), which describe the packet or message formats. The coarse structure of the control plane is specified with a finite automaton identifying states corresponding to different steps, such as initializing a session, waiting for an acknowledgement, or terminating a session, and the flow between them. This has to be augmented with procedures for receiving, validating, and sending PDUs and how the contents of the PDUs influence the protocol processing. Process calculi (used for interactive systems) do not correspond well with the system description being structured around the explicitly identified flow in the control automaton. On the other hand, models for reactive systems are also ill-suited because they cannot capture the rich structure of PDU formats and their processing and influence on the execution. While the data processing can be complex, the data flow itself is simple and explicit, being primarily associated with the receipt and sending of message content. The proposed formalism mirrors this structure by supporting expressive data manipulation operations but making both the control flow and data flow explicit in the program specification. It can therefore be seen as augmenting automata-theoretical models with the paradigm of data flow languages [72].

The work presented in this thesis is based on our experience with the Portable Call Agent (PCA) tool. PCA was developed to emulate *softswitch* implementations with support for multiple Voice-over-IP (VoIP) call-control protocols (such as Media Gateway Control Protocol (MGCP) 1.0/Network-based Call Signaling (NCS) 1.0, or H.248), for the purposes of testing and demonstration. Rather than hard-coding-in the implementation of the many protocols and VoIP features, PCA defines a language in which call-flow processing and message translations are specified as a "call-flow graph;" such specifications are then dynamically loaded, instantiated, and executed using a "call-flow engine." Different call-control protocols and call-service features were then implemented by defining call-flow graph specifications for each. Because of the general nature of the call-flow graph specifications, PCA is applicable to the implementation and emulation of services beyond just VoIP call control. Our model for network services is a distillation of the theoretical underpinnings of PCA.

Communication systems must perform protocols correctly. Creating correct and complete communication systems is difficult. Organizations have invested hundreds of staff-years into designing, programming and testing mission-critical communication systems. Whether we are using the last century's plain old telephone service, watching videos, or using the latest computer tablet, communication protocols [69] are coordinating the delivery of content between end users. Modern communication systems generally have separate data and control planes, that is the content (data) and the control of the content are accomplished using distinct sets of procedures and messages [75].

The data plane is the data communication path that contains the content exchanged between communication end points, often voice or video. In contrast, the control plane is responsible for the set up, tear down, error handling and monitoring of the content being communicated. The control plane is responsible for controlling communications resources required to validate, negotiate, acquire and configure the data plane. In addition, the control plane must coordinate its activities with other systems, such as accounting and operations, so that the service provider can fix problems as they arise

and receive remuneration for their efforts.

Management of the data-plane resources, is routinely accomplished using messages exchanged between peer control plane entities [63], [29], [6]. Therefore, the control plane entity must keep track of what messages have been sent, what are possible messages that can be received from the remote/peer communication manager, and what to do if certain messages are received or not received within a certain time interval. This implies timers of some sort. If a centralized approach for management of communications sessions is used, there will be a large number of calls (or half calls) active at any time. This, in turn, suggests that the communication manager will need to keep a large number of timers and a large number of memory objects that represent call or session state.

Writing software that implements the control plane of a modern communication node, whether it is a Next Generation Network element such as a User Agent (UA) or Back-to-Back UA (B2BUA) [25] used in the wired network, or an element that is part of the wireless Enhanced Packet Core (EPC) such as an Mobility Management Entity (MME) [52], entails a significant investment of effort to realize, using procedural and/or object-oriented languages.

Audio or Video communication *features* are realized through the control plane. As more features are added to a communication system, the control plane reflects these new capabilities, and the complexity invariably rises.

This remains true even in the case of a landscape dominated by functional protocols such as Session Initiation Protocol (SIP) [63]. By its very nature the use of SIP should greatly simplify the control plane structure, as it was designed to handle call set up and tear down in a peer-to-peer fashion. For simple point-to-point scenarios, if both endpoints implement the standard in the same way, this can be true. However if the communication system has been positioned to offer so-called Custom Local Area Signaling Services or CLASS 5 features, such as call forwarding, call hold, three-way calling[1], then the situation becomes much more complex. A central control element must supplement the capability of the endpoints to deliver the advanced service, or worse, the central control may have to adapt the

1 http://morehouse.org/hin/ess/ess11.htm (Accessed June 2013)

peer control message streams in order to mesh together dissimilar end point models.

The IP Multimedia Subsystem (IMS), originally defined in 1999 [7], is an example of a SIP-based communication system. Typical realizations of IMS contain several layers of network elements to control the authorization, configuration and setup of multimedia sessions. In the IMS model, the Application-Server (AS) node serves the function of realizing the user services including supplementary services, those beyond the basic point-to-point call model.

For any system to function correctly, every active entity has to have a well defined behavior. When the endpoint is very simple, it is straightforward to create a consistent ubiquitous solution. In the case of IMS, the endpoint is a SIP terminal, one that may be an embedded computer in the form of a physical phone, or it may be a softphone, an application running on a platform such as a tablet, smartphone or Personal Computer. IMS has been positioned as a system that works using components from many vendors. Thus any system that a vendor expects to be widely deployed must interoperate with a range of manufacturer's endpoints, potentially endpoints that are already in use. SIP is a functional protocol, that is, an endpoint executes a set of *functions* or methods that establish, modify and terminate sessions. As a functional protocol, each SIP endpoint must be tested for interoperability, not only with the IMS core, but potentially with other endpoints as well. If, rather than passing the control messages end to end, as they are received, the IMS system terminates and regenerates each session within IMS, then the network elements take over the chore of interworking two potentially incompatible SIP endpoints.

The issue of interoperability became such a large issue, that ultimately some vendors adopted multiple SIP endpoint models – so called “smart” and “dumb” endpoint models. The “smart” endpoint could perform many of the supplementary services on its own, while the “dumb” endpoint required the use of a centralized element in order to obtain behaviors such as secondary dial-tone and digit collection / dial-plan evaluation [64]. Having dissimilar models forces the network architecture to have an element that models each type of endpoint, and can convert between each of the models for all

of the needed supplementary services. Far from being a solved problem, work continues today on standards such as SIPConnect which defines interworking procedures to support advanced features [56] .

In contrast to functional protocols, a stimulus/response protocol defines a set of lower-level events, the *stimulus*, such as a user hanging up the handset, or pressing a button on the phone, to inform a centralized controller about the state of a users line. The controller then issues commands, the *responses*, to the endpoint to create, modify or terminate objects that generate or consume multimedia streams, play out tones to the user or set values on a user's display. In the case of stimulus/response protocols, such as H.248/MEGACO, a central controller is directly responsible for the behavior of the communication system. In this case, the Media Gateway Controller element will become increasingly complex with each added feature.

Although the data plane poses its own requirements, this thesis will focus on the challenges that surround the realization of a modern communications control plane. In some cases, the control element must alter resource description data not associated with a normal call setup, perhaps to achieve interoperability as mentioned above.

Translating a control plane specification into a working system is challenging because:

- Errors can occur in the translation of requirements into code, as there is a large difference between the level and expression of a specification and that of the actual code needed to realize the system;
- Current models of computation provide only partial support for the creation of a working system, leaving many difficult aspects to the programmer;
- System-behavior requirements frequently change during the course of a project;
- The system specification may be incomplete or a system may need to interwork with other systems that deviate from the published specification; or the specification may contain a model that, while correct, does not translate into an efficient (or feasible) implementation;
- Reliability and performance features are left to the application

writers outside of the model framework;

- Management subsystem (OA&M) development routinely lags behind the development of the main functions and are difficult to keep in sync with both the schedule and feature set of the system.

Problem Statement

Many communication systems, especially the control-plane portion, are described by a state model. Examples range from early systems such as fax (T.30) [58], to modern systems which must function in a very predictable manner include such as the MIL-STD 188-200, a tactical radio system.

In addition message-chart diagrams are routinely used as part of the specification of a communication system. The message- chart definition is often used to illustrate either a specific feature along with other specifications, or how the behavior uses one or more of the optional aspects of a protocol.

The requirements suggest that what is needed is a flexible system that has the following characteristics:

i. Accepts a human-editable description which can be easily modified to reflect new features or changing system requirements;
ii. Executes the description with sufficient speed and efficiency to create feasible instances of a variety of telecommunication services;
iii. Presents the programmer with a computational model that corresponds to the ways many communication-centric problems are posed;
iv. Contains support for high reliability and availability as an integral part of the environment, without requiring additional programming effort;
v. Is general enough to describe applications beyond those considered to be traditional communications (mixing commerce, communications, and vertical market features);
vi. Is concise and easily readable without requiring specialized authoring tools

This thesis describes an alternative way to describe and realize *stateful protocols*. It is called the ***Lightweight Flow Engine*** (**LiFE**). Specific use cases will be used to demonstrate that "LiFE" is general enough to be used to describe a range of processing functions, including those associated with telecommunications, business process, and as well as mixture of web and traditional IT tasks. In addition the performance of "LiFE" will be examined to show that the execution framework is efficient enough to execute a wide range of functions on off-the-shelf commodity hardware. The thesis will present the programming model of "LiFE" to show that the abstraction provided by the system corresponds well to the ways many communication-centric problems are described. Finally, the thesis will describe reliability and availability features which naturally arise from the way the system executes.

Throughout the discussion, LiFE is also referred to as "the flow engine", or in some cases the "work-flow router". All of these terms should be treated by the reader as synonyms for LiFE.

During the course of this thesis, the computational model of LiFE will be described using a variety of means. LiFE will be described directly through the presentation of a model, and will be further illustrated though the use of two examples: a business process, and a portion of a VoIP service. As the latter is comprised of about 1000 LiFE state declarations and is over 29,000 lines of flow language, the full example will not be included in this monograph; however, a few key portions of that graph will be described and illustrated to convince the reader of the expressive power of the language.

2 A Brief Overview of LiFE

In the following section, I will give an overview of LiFE. Conceived and executed as a system, LiFE consists of a *language* to describe service logic, *transformation languages* that describe conversions between internal state of LiFE elements and external messages, an *execution environment* for the computations, a dynamically configurable *management process* and a set of external *helper programs*. Central to the design of LiFE is the decomposition of the problem space into: 1) service logic; 2) transformation of messages between external and internal forms; 3) resource discovery; and 4) reliability/availability mechanisms. Referring to Figure 2.1, the Flow Graph Description contains the service logic – that is the definition of what actions are carried out in response to input events.

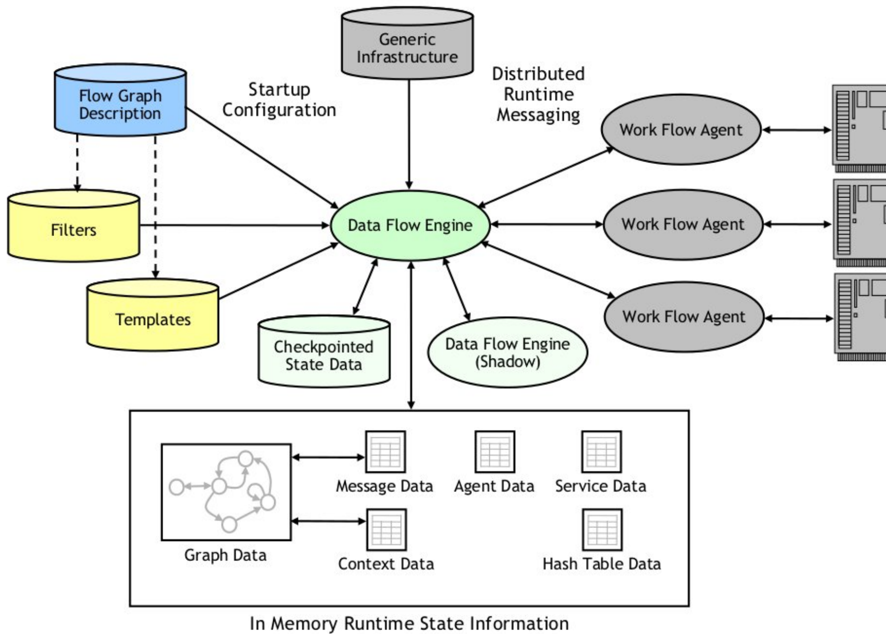


Figure 2.1: Flow Engine components

The Filters transform system input, converting input messages into the common internal representation used by the service logic. Templates perform the output transformation, converting the internal representation of data elements into messages appropriate for their destination. The combination of the Filter and Template mechanisms perform the majority of the interfacing duties of many systems. A number of applications, such as interfacing to 3rd party or homegrown data bases that have existing interfacing programs, these programs need special consideration. The WorkFlow Agents are used to wrap these programs to provide a control and data interface to the system without requiring the creation of additional interfacing code. This is a convenient way to connect the system to legacy applications and existing systems.

An application is expressed using the flow graph language. The basic model roughly follows that of a Mealy machine (See Figure 2.2); a formal discussion of the computation model follows in a later

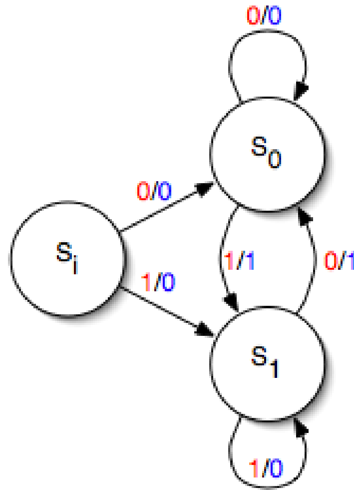


Figure 2.2: A simple Mealy machine

chapter, this discussion will be an informal one.

2.1 Service Logic

In the LiFE model, a set of nodes, called **states**, are connected by a set of edges called **arcs**. One or more tokens, called **contexts**, traverse edges from one state to another. A context traverses from one state to another in response to an **event** that has been routed to a particular context. Events are generated as a result of a message or a timer. Traversal of an arc by a context may result in the invocation of the **arc action**, a method definition contained in the arc declaration. There is a fixed set of arc actions, each of which may use arguments contained in the declaration of the arc. An arc action may be a **GRAPH_CALL**, which pushes the current location of the context onto an internal

calling stack before the context traverses to its next state. When the context encounters a **GRAPH_RETURN** as part of its next state processing, rather than interpreting the next state value directly as the next state for the context, the system uses the internal calling stack to identify the arc with the last **GRAPH_CALL** directive, and selects the next state array located at that calling arc as the array of potential next states. The system will then use the next state value as an index into the calling arc's next state array to choose the next state for the context. In this way, portions of the flow graph can be used as subroutines (or subgraphs), for often repeated procedures.

Each context owns a name/value pair (NVP) list, which along with the current place of the context constitutes its state. While this is discussed in more detail below, the name/value pair list is used in conjunction with the various arc actions to perform actions, including sending a message to an external client, or selecting the next state of the context. Later in the text each of the arc actions will each be covered in detail.

Each arc may contain zero or more name/value pair directives. These constructs are defined as part of a particular arc, and triggered only if an event triggers the traversal of a context from one state to another. The name/value pair directives are applied only to the current context's name/value pair list. Directives may create, delete, or modify the local context's name/value pair elements. The primary use of these directives is to update the local context's name/value pair list. In addition, there are specific arc directives that can send a message between contexts, while others can be used to provide read access by one context to another's name/value pair list.

In summary: *Service Logic + Message Transformation + State = LiFE*[2].

At the heart of LiFE is the representation of state. As each message is received and processed by the flow engine, the state of the system changes. There are certainly a multitude of ways to represent state; the approach taken in the construction of LiFE was influenced by the characteristics of the applications at hand. The common model that arose was one of a very large number of elements with relatively

2 With apologies to Niklaus Wirth.

independent behavior. Some of the problems of interest (soft-switch, order transaction systems) included grouping of elements, parent-child relationships, and inter-element messaging. Largely, however, the elements could be viewed as independent elements with very limited shared state. Using this model for inspiration, LiFE defines an object called a **context**. Each context owns a local name/value pair list that contains the state of the context. The context's name/value pair list is private, with very limited direct access by other contexts (even limited direct access may be a mistake – but it is useful).

Contexts communicate state using messages. These may be internal messages, from context to context, or may be external messages, from a context to an external server or system. Contexts gather state by processing incoming messages. As mentioned above incoming messages are converted into a common form, which is the name/value pair sets. Incoming messages present to a context as an event. Depending upon "where" the context is currently located within the flow graph, (its place), the message may be converted in a specific way to a name/value pair set. That set is then added to the context's private name/value pair list, creating new pairs or replacing existing pairs as appropriate.

There are two other repositories of state in LiFE, one explicit and another implicit. Declarations contained in the flow graph can cause the execution environment to create server sockets using Streaming Control Transmission Protocol (SCTP), Transmission Control Protocol (TCP) or User Datagram Protocol (UDP). In addition, there are flow language directives that can cause a TCP connection to be created and managed on behalf of a specific context. These objects clearly contain state, but not all of that state is visible to the engine. In practice this has not been a problem, as application level protocols are used to capture the state that would otherwise be invisible to the service logic. The other repository of state is a global name/value pair list. This list can be accessed from any of the contexts in the system. The existing design of the flow engine permits a context to read and write the global name/value pair list. In practice, most contexts only read from the global list.

2.2 Resource Discovery

As part of the flow graph declaration, there is a section that lists external resources that are needed for the complete execution of the computation. The declaration of these resources includes a set of Open Network Computing Remote Procedure Call (ONC RPC, here after called RPC) [12] program and version number ranges. Each range is named. The flow engine interprets each named range as a potential pool of servers that may be used to perform actions on behalf of a specific context. At start up the system sends out broadcast messages to each defined server pool. Any currently listening servers respond to the broadcast “chirp” and will be subsequently used by the flow engine when requested by a context. The collection of server resources found as part of this discovery mechanism are referred to as “Workflow Agents”. These standalone servers may be located anywhere on the local LAN segment, limited by the reach of a broadcast UDP packet. The addressing scheme of the Workflow Agents supports multiple instances of the same Agent class co-existing on a single machine. Thus the flow engine may discover a large set of Agents distributed over a set of machines within the LAN segment. Since the RPC mechanism uses an architecture independent messaging presentation protocol (eXternal Data Representation or XDR) [68], the Workflow Agent pool can be defined across sets of heterogeneous machines (ARM, MIPS, x86, PowerPC).

In a similar way, flow engines can discover and be discovered by peer engines. This mechanism is used as a first check when an instance of a flow engine is started. Each flow engine is required to have a unique address assigned to the flow engine at run time. Beyond the obvious use as a mechanism for directing messages to a particular flow engine, the address is used to determine if a context is native (has been created locally) or if a context is “foreign”, (has been created on a different flow engine instance). The flow engine address is also used to communicate to external monitoring and administration processes

To enforce the uniqueness of address; at start up time, the flow engine will attempt to discover if another instance is running on the

LAN segment with the same address. This is done by making a broadcast RPC request to Procedure 0. If another instance of the flow engine responds to the broadcast, it means that there is an active instance of the flow engine already running on that address, and the new flow engine will provide a diagnostic message and exit.

There are two variations of monitoring programs available for the flow engine: the Workflow Tracker and the Flow Engine Element Management System. The Workflow Tracker is used as a diagnostic aid during the creation and debugging of a flow graph (computation), while the EMS is used as general tool for controlling features defined in a specific flow graph. Since the computation is only defined at run time, and therefore the managed features may be dramatically different from one graph to another, the EMS was designed to have graph-specific management functions instantiated when the EMS is first connected to a running flow engine.

2.3 Reliability/Availability Features

The system finds external resources at start up and then looks for additional resources on a server-pool basis either when it has detected that work has become queued waiting for a server, or when instructed via an external command. If a request has been queued to an Agent, and that Agent becomes unresponsive, (after a number of retries), the request is dequeued from the now unresponsive Agent, and queued to the next available instance. If there are no further free Agents, the request is held by the system until additional Agents have been discovered. The unresponsive Agent(s) are internally marked as “down”, and no further work is queued to those instances. In the case just described, the flow engine will automatically launch a periodic discovery “chirp” to find appropriate Agents. In this manner the flow engine will perform correctly in the face of temporary network partitioning.

The flow engine contains a checkpoint capability. Just as the engine finds Agents on the network, it can also utilize the portmapper/rpcbinder [67] registration of each flow engine to contact peer flow engines. If directed at run time, a flow engine may advertise its ability to serve as a standby node for another flow

engine. If the offer of backup is accepted by the peer referred to as the primary, inactive copies of contexts are created or updated on the standby node when the context enters a STATE (place) on the primary that contains a DO_CHECKPOINT directive. The checkpoint directive is part of the flow language, and can be included in any of the defined graph states. If the standby node receives checkpoint information about a new context, a copy is created and put at the current STATE in the standby graph. When created on the standby engine, the address of the active engine is also transferred. This information captures the affinity of the context, so that origin of the context is not lost. The current name/value pair list is sent from the active engine to the standby engine along with the context's current place within the graph. The set of contexts created in this manner are considered “foreign” contexts are kept in an inactive state, that is although the contexts may be occupying an arbitrary place in the standby graph, no timer functions operate, nor will any message events be queued to these contexts. They may be moved from place to place within the graph, have their name/value pair lists updated, or destroyed in response to checkpoint messages from the originating flow engine instance.

When the connection between the active and standby flow engines has been lost for a predetermined amount of time, all of the standby contexts associated with now disconnected flow engine are sent a named exception. The exception causes the contexts to perform an initialization procedure as defined by the flow graph, and then the contexts continue processing from the last checkpointed state, using its current name/value pair list. At this point the contexts traverse the flow graph in the same manner as the “local” contexts. The contexts retain their original affinity, so the flow engine has not lost the identification of the originating flow engine. If at a later time, the flow engine becomes aware of another flow engine instance with the identification that corresponds to the now active “foreign” contexts, the engine will offer the contexts back to their “parent”. If accepted, the contexts will be suspended and transferred back to the originating flow engine, and will be deleted from the standby engine. If the previous parent refuses the contexts, then the flow engine changes the affinity of the contexts to correspond to its own identifier, thus

changing them from “foreign” to “local”. A flow engine may serve as a standby node for more than one flow engine. However, a flow engine may only back up its contexts to a single other flow engine. In addition, a flow engine may have its own set of active contexts running at the same time it is hosting “inactive” contexts. It is possible for two flow engines to back each other up, running in an active-active configuration.

Using this approach, flow engine updates and minor graph updates can be made on a running system, by bouncing active contexts from the active to the backup, restarting with the new executable and/or graph and allowing the system to re-balance itself. Then repeating the same procedure for the other instance. The flow engine will not accept a checkpoint unless both flow graphs have the checkpointed graph location in common. So if the graph updates mentioned above do not share common quiescent states, then the strategy mentioned above will not succeed. In this case another mechanism, described in US patent # 8146069 & US patent # 8141065, “Methods for Update in Place” (reproduced in Appendix A) can be used to perform updates in the graph. The combination of these two approaches provides the capability to provide a high degree of system availability.

3 Background & Related Work

LiFE is used to describe interactive systems [30] rather than reactive systems. A reactive system is one that is used to describe automatic process control and monitoring. Examples of reactive systems include the processing that encode high-data-rate, low-latency inputs such as real-time video transcoding, radio-waveform processing, and audio mixing. Systems that are suitable candidates for realization using LiFE are interactive applications that define a return control path for synchronization such as telecommunications control plane (e.g. soft-switches) .

3.1 Linda (Gelernter)

At the heart of the flow engine is a name-value pair list that, along with the context's place, represents the current state of the context. There are similarities between flow engine primitives and those described by David Gelernter as part of the Linda coordination language [26]. While Linda is used to describe a general mechanism to coordinate processes, and defines a set of operators for the name-value pairs (tuples), the flow engine defines a set of specific operations that can operate on the name-value pairs (tuples) which integrate the operations defined in Linda, and perform them largely

as side-effects of the execution of a name-value pair directive. In contrast to Linda, the name-value pair list in the flow engine is never consumed by an operation. The central operations of the flow engine are based upon name-value pairs (referred to as *tuples* by Gelernter). Most of the operations in the flow engine are the closest to the “rd” operation in Linda, that is, the tuples are non-destructively read. This is the case when messages are generated by the flow engine using the combination of a “template file” and the name-value pair list (or tuple-space). The only time a tuple is removed by the flow engine, is as a result of a “DEL” directive, this is the closest to the Linda “in” operation. A flow engine tuple can be updated however, and in this case, the old tuple is destroyed and replaced with the new tuple. Any of the set directives in the flow engine (e.g. SV or the output resulting from incoming message that is processed by a call flow filter), produces tuples that are applied to tuple-space just as Linda's “out” operation. The Linda operation “eval” roughly corresponds to the “EVAL” call flow directive. In the case of the “EVAL”, the ways in which the tuples can be evaluated are limited to a specific set of string sorting order comparisons. The results of the EVAL are used as a control mechanism, and does not update the tuple name-space, but rather is used to set the next place for the context. In the flow language the Linda “eval” operator more closely corresponds to the output of the set-value (SV) or conditional set-value (COND) directive. In these flow engine directives, the tuple-space is updated based upon a combination of constants, existing tuples, and functions of existing tuples.

3.2 Unified Modeling Language Based Approaches

3.2.1 UML Background

Unified Modeling Language or UML [24], goes beyond what the name suggests. UML certainly is a language that can be used to describe a software system, but as described by the Object Management Group, UML is a complete software ecosystem that includes creation, visualization, analysis and code-generation tools. A

number of efforts to describe state based computations either exist as extensions to UML or have been connected to UML. The version of Unified Modeling Language (UML) defined by the Object Management Group is specified in detail, in documents that are publicly available. UML is only one of the technologies that is championed by OMG.

"The OMG's Unified Modeling Language™ (**UML**®) *helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements. (You can use UML for business modeling and modeling of other non-software systems too.) Using any one of the large number of UML-based tools on the market, you can analyze your future application's requirements and design a solution that meets them, representing the results using* UML *2.0's thirteen standard diagram types*.

> *You can model just about any type of application, running on any type and combination of hardware, operating system, programming language, and network, in UML. Its flexibility lets you model distributed applications that use just about any middleware on the market. Built upon fundamental OO concepts including class and operation, it's a natural fit for object-oriented languages and environments such as C++, Java, and the recent C#, but you can use it to model non-OO applications as well in, for example, Fortran, VB, or COBOL. UML Profiles (that is, subsets of UML tailored for specific purposes) help you model Transactional, Real-time, and Fault-Tolerant systems in a natural way. "*[3]

As indicated by the authors of OMG-UML, it is a general purpose way to model any software system, including, of course, the same types of applications that have been modeled and implemented with LiFE. The real-time aspect of software implementations is addressed by an extension called the Modeling and Analysis of Real-Time and Embedded Systems or MARTE. The goals of MARTE are:

> *"Among others, the benefits of using this profile are:*
>
> - *Providing a common way of modeling both hardware and software aspects of a RTES in order to improve communication between developers.*

3 http://www.omg.org/gettingstarted/what_is_uml.htm (Accessed June 2013)

- *Enabling interoperability between development tools used for specification, design, verification, code generation, etc.*

- *Fostering the construction of models that may be used to make quantitative predictions regarding real-time and embedded features of systems taking into account both hardware and software characteristics."*[4]

OMG UML-RT provides a rich modeling environment, with sets of tools available to support a developer through the QoS Modeling, Architecture Modeling, Platform-based Modeling and Model-based Analysis parts of the design. Bridge programs have been developed to link MARTE based models for various development tools, including those built upon the Eclipse platform.

In contrast, LiFE uses a far more restrictive model to describe a computation, but through that model includes enough built-in facilities to describe the application enough to execute it directly.

There have been other efforts, under the UML umbrella designed to describe state machines, including SMDL[5] a project that accepts a state description language and emits Java code. Most of the UML approaches involve code generation and integration to realize an application.

3.2.2 Specification & Description Language – Real Time (SDL-RT)

The SDL-RT organization states that: *"As its name states, SDL-RT is based on SDL standard from ITU extended with real time concepts. V2.0 has introduced support of UML from OMG in order to extend SDL-RT usage to static part of the embedded software and distributed systems."*[6] Chiefly, SDL-RT is a graphical language that allows a designer to define an event-driven application, using a set of states with embedded C language code and messages that send in and out of blocks of code. SDL-RT goes beyond modeling the system, with a specification that can generate code. Although, SDL-RT has been

4 http://www.omgmarte.org/ (Accessed June 2013)

5 http://smdl.sourceforge.net/ (Accessed June 2013)

6 http://www.sdl-rt.org/standard/V2.3/html/index.htm (Accessed June 2013)

designed to describe a wide variety of applications, the state of the application is held internally to the system. In contrast, the flow-engine has its state explictly described in the combination of name/value pair lists and the current graph. In addition, unlike LiFE, there is no provision in SDL-RT for updating the computation during execution.

3.3 Estelle

Estelle is a specification language that can be used to describe protocols and distributed systems[7]. Created during the 1980's, Estelle [16] has been adopted as an ISO standard 9074:1989[8]. Estelle is presented as a technique based on an extended state transition model. The language is described by the authors as an extension of ISO Pascal. Informally, the Estelle models a system as a collection of possibly interconnected *modules* that communicate through a set of channels. Structured, strongly typed messages pass through specific, declared channels. Estelle specifies module interfaces and messages exchanged between modules with enough rigor to be able to analyze, simulate, create code and generate test scenarios for a complex system such as the US military's mobile combat radio Data Link Layer[9]

There are a number of common elements between LiFE and Estelle. Both build upon a similar state transition model and primarily use messages as the communication mechanism. Estelle, based upon ISO Pascal, is a far more general way to describe algorithms, with few restrictions placed on the facilities found in ISO Pascal.

Managing internal computational state appears to be more straightforward using the flow engine. LiFE uses the combination of the graph state and a set of name/value pairs to define the state of the computation, and LiFE includes a primitive within the execution

7 www.cs.uga.edu/~kochut/Teaching/8060/presentations/.../Estelle.pdf (Accessed June 2013)

8 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=16659 (Accessed June 2013)

9 http://dx.doi.org/10.1016/S0140-3664(99)00246-7

engine that enables two instances to communicate checkpoint information. With Estelle, the state of the system is contained inside of the modules themselves. Using the structured messages in Estelle, one could create a representation of the module's internal state, but this would be left to the programmer. Using the “Updates in Place” method for the flow engine, outlined later in the thesis, LiFE can support adding or modification the function of a live system.

3.4 Communicating Finite State Machines

Communicating Finite State Machines is a model of communications protocols [14]. The model is chosen to be both powerful enough to represent protocols, and defined in a manner that allows analysis. Many of the constructs used to explain CFSM in the paper have a direct analogy to concepts found in the flow engine. A central assumption in the CFSM model is a set of processes that traverse a set of states in responds to messages arriving on a queue. While reception of messages can cause the process to move from state to state, other transitions cause the CFSM to emit a message. Processes in CFSM correspond to contexts in LiFE, with messages playing the same role in both models.

LiFE defines a manner to describe logic that can control how long, and under what conditions, the context will transition between states. A context in LiFE holds its overall state as a combination of the context's name/value pair list plus its current occupied state. Even though this theoretically is at odds with the CFSM model, a graph designer could create flow definition that does not use the name/value pairs to control state transitions, and have a result that corresponds closely to the CFSM model.

3.5 Petri Nets

Petri nets [57] define a process model that is comprised of three elements: *places, transitions* and *arcs.* A nice overview is located here[10]. The model has formal semantics and is used to describe distributed

10 wwwis.win.tue.nl/~wvdaalst/workflowcourse/.../pn2refresher_long.ppt

systems. There is a large number of tools to support the creation, visualization and analysis of systems modeled as Petri nets. A number of the tools include support for useful extensions to the basic model, adding support for time elements, colored nets and hierarchical modeling.

A number of the basic constructs in LiFE correspond closely to those found in Petri nets. LiFE is built around the notion of a multiplicity of tokens (called contexts) that proceed through a series of places (called states), controlled by transitions (called Transition Arcs or simply Arcs). A context in LiFE has a type or color, that corresponds closely to the color extension in Petri nets. In addition the LiFE context contains a name/value pair list, that contains a collection of data elements. In addition, notions like supplemental states, graph call/returns used in LiFE to reduce the presentation complexity of the graph, do not directly align with concepts in Petri nets. A big difference in LiFE is the inclusion of timing into the model. Another difference is the ability of a LiFE context to check for state occupancy in the graph. As will be explored in Chapter 4, LiFE, unlike the basic Petri net model, is Turing complete.

3.6 RoseRT

A commercial offering of IBM, Rational Rose RT[11] is a model-driven development platform that is used to capture system requirements. RoseRT than uses the model entered into the system to create executable code in a number of target languages. The generated code, along with supplied libraries forms a framework for the target software solution. In many cases, the framework is then populated with additional code that is created by software developers. While both RoseRT and the flow engine use a state-machine model, RoseRT emits code, to which the users often add additional software, and is subsequently compiled into the executable. The code base is fixed at this point. The design decision of executing the flow engine language directly, rather than using a separate emit/compile step, has enabled an update-in-place capability – that is, logic that is expressed in the

11 http://www-03.ibm.com/software/products/us/en/rosetechdev (Accessed June 2013)

flow-engine language can be updated while the flow engine is running, without requiring a restart of, or pause in processing. Clearly the engine code itself cannot be updated in this manner (one would use a primary backup strategy for that and causing the system to fail over would result in a delay in processing). In addition, the structure of the flow engine's dynamically linked input/output transform libraries could also unlinked and re-linked under control of the flow engine, to accomplish an update of individual libraries. This provides a fairly flexible capability to update a system without requiring a complete restart and the resulting loss of availability.

3.7 BPEL

The Web Standards Business Process Execution Language, WS-BPEL[12], uses web standards, WSDL, XML, SOAP and Universal Description Discovery and Integration (UDDI) to describe a business process. The goal of WS-BPEL is use a set of web services in a coordinated fashion to execute a specific business process. A WS-BPEL process takes the form of a document with sections that define types, messages, interfaces, and the sequence of steps to take in response to an input. A WS-BPEL definition is *call back* in nature.

The flow-engine language has similar control-flow constructs as WS-BPEL, such as structured sequencing, conditional behavior, repetitive behavior and selective event processing. In WS-BPEL a business process is defined using a set of declarations that include data variables, interfaces, exception handling and finally the business logic the describes the expected normal process flow, the analogous construct in LiFE is a collection of reachable states in the graph. Just as in BPEL, multiple processes can be created.

Parallelization can be defined both in BPEL and the flow-engine language in a natural way, with the system creating new resources in response to input data. WS-BPEL has an explicit “flow” concept in the language, using a *join* construct to synchronize different events. The flow engine has a *rendezvous* function that allows a set of contexts

12 http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-bpel/ws-bpel.pdf (Accessed June 2013)

to wait at a specific place for all of the contexts. In both WS-BPEL and the flow engine, the rendezvous can be timed to prevent indefinite wait times. Exceptions are likewise supported in both systems. Variables are defined explicitly in WS-BPEL, while the flow engine will create a variable upon first reference. The inspiration for this feature came principally from “awk” [1]. There is no explicit variable-definition section in the flow engine language. In keeping with the document and XML nature of WS-BPEL, the process is statically defined in BPEL, while the flow engine has the capability of updating the process during execution. Vendors such as Oracle have introduced capabilities to update a running BPEL process (SOA Suite 11g) (This seems to be limited in allowing only updates for specific verbs[13]). In addition, BPEL does not have a capability to express a secondary source for state (variables, business processes in progress) in the language while the flow engine has facilities that allow a secondary instance to hold the primary's state and take over operation automatically when the primary fails. The failover strategy in BPEL falls to the BPEL-engine vendor (Oracle ESB). While a finite-state construct can be created in WS-BPEL, the structure of the language does not lend itself to expressing a problem in this manner (Translating WF-Nets into BPEL). BPEL, using XML, is verbose while the flow engine uses a much more concise language. .

3.8 Web Services Flow language

This language is designed as way to describe business processes. Similar to BPEL, WSFL [44] breaks down the process into a series of single steps and models the steps as nodes in a graph.[14] Edges between the nodes, called “control links” are directed edges which have associated events. WSFL supports a join condition, which requires a set of conditions to be satisfied before an activity associated with the node will be performed.

LiFE has been used for a number of applications, at distinct levels of abstractions, from business-process modeling like WSFL to control

13 See http://www.linkedin.com/answers/technology/enterprise-software/TCH_ENT/878964-527202

14 http://xml.coverpages.org/wsfl.html

layer signaling in the case of softswitch emulation. LiFE supports all of the abstractions contained in WSFL, including the join, which is called a rendezvous in LiFE. WSFL uses an XML based language called WSFL to describe the computation. At one point, the flow engine was modified to accept an XML based graph language. Without an language specific editor, maintaining a graph of any size, let alone the 900 defined state VoIP controller graph became extremely unwieldy, and the effort was curtailed. Recently an effort was started to use JSON as a syntax for the graph language, as that may result in a form of the specification that is still suitable for maintenance without a specialized application.

3.9 SCXML

State Chart XML (SCXML) [7], initially released in 2005 by W3C, currently on working Draft 16 dated February 2012, defines a structure for computation that is very similar to LiFE. As its name suggests, SCXML is based upon XML, a language that combines concepts of Call Control XML (CCXML) with the concepts of Harel State tables (reference here). SCXML has much in common with LiFE, including the idea of a multiplicity of entities (SCXML sessions, LiFE contexts) that proceed from state to state, driven by events. Another key similarity is the use of name/value pairs to define data elements. LiFE allows the values in the name/value pair list to be used in a flexible way in the definition and/or update of name/value pairs. SCXML calls out Xpath 2.0 to provide flexible assignment of variables in an SCXML. SCXML uses Event I/O Processors to transport messages to and from other SCXML sessions. In LiFE, the SEND verb is used to send messages to either other LiFE contexts (internal) or external destinations. In LiFE, an argument of the SEND verb will reference the template file which corresponds closely to the datamodel in SCXML. All messages in SCXML are sent as XML messages, while in LiFE the messages can be sent as octets, allowing a large amount of flexibility in output format and allowing a direct interface between LiFE and external systems (such as SIP or MGCP entities). In a similar way, LiFE has been designed to accept messages

from external sources that have variations in syntax, primarily to support direct interconnect to protocol elements. Practice has shown that there are real variations in various vendors' implementations of so-called standard protocols (e.g. SIP). A construct in LiFE that has been useful in reducing the number of explicitly defined absolute states, is the GRAPH_CALL construct. In this mechanism, the current active state/transition is pushed onto a stack, before the target state is visited. The target state can be thought of as the entry point of a graph subroutine. Once control has been passed to the target state, processing continues as the context proceeds from state to state within the graph subroutine. At some point, one of the transitions has as its action a graph return verb. Control will then resume at the most recent state/transition GRAPH_CALL. LiFE will choose the target state (NEXT_STATE) by indexing into the array of possible NEXT_STATES using the index value defined by the graph return. This construct has been particularly useful handling the "Notify" event that is asynchronous as stimulus/response protocols such as MGCP and H.248. In the softswitch emulation graph, the construct saved explicitly defining several hundred additional states. More importantly, it saved the errors that could have crept into the design as well as simplified debugging the resulting behavior of the system. It is clear that SCXML could provide the equivalent behavior to the GRAPH_CALL, simply by inserting the explicit set of states in the graph subroutine in place of the GRAPH_CALL. The LiFE construct of a "Supplementary State" is not available in SCXML. In practice, the Supplementary State is used as a dynamically assignable exception handler. A context may or may have a Supplementary State assigned. If such a state is assigned to context, that state is carried by the context as it proceeds through the graph. If an event is delivered to the context that does not match any of the transitions defined at the current state, the context's supplementary state will be then checked for a match. If there is a match, then that transition will be taken. Since the Supplementary State assignment can be changed during the execution, it becomes very useful as a context sensitive exception or default handler for events. In SCXML, the same behavior could be defined by adding a child state to each of the states along the session's execution path. Paths for different types of sessions that required

distinct exception or default handling, would be disjoint. Based upon experience with LiFE, replicating these states could significantly increase the number of total states in a complex graph.

3.10 CCXML

Call Control Extensible Markup Language (CCXML) [8],[61], first defined in February 2002, was designed to provide telephony call control. Although CCXML is closely associated with VoiceXML, the two languages are separate. CCXML is a domain specific language, with language features tied closely to telephony sessions, and an event model with hides underlying state by declaring CCXML stateful objects such as call.ACTIVE and call.INVALID. In addition, looking at the main methods: "accept, createcall, createconference, disconnect, join, unjoin and reject", it is clear that significant call state is hidden in these procedures. These heavier weight methods make managing calls much simpler, as the user defined logic can be much simpler than if the state machine were forced to implement the low level steps needed to model an endpoint, perhaps using a stimulus/response telephony protocol from on-hook to an active call state. As stated as one of the design goals, CCXML is used to define a static call control definitions in order to, among other things, to allow an efficient version of the behavior to be realized. A current implementation of CCXML (Voxeo Prophecy 11) is rated at 25 call setups per second (half call setups, it seems from the example provided) on a standard Intel machine.

A class of description languages that describe frameworks rather than complete programs. Many of these state description languages are intended to be used within Integrated Development Environments (IDE) such as Eclipse. Examples include State Machine Description Language (SMDL). An effort that seems to have stopped development in 2009, SMDL is very similar but simpler than State Chart XML, but the language is focused upon describing a set of nodes (states) that will form the framework for a program, rather than describing a complete program.

3.11 Opensips

The open source project Opensips[15] [27] has grown from a SIP proxy to an extensible system that embodies many of the traits of LiFE. Based upon a scalable, high-performance SIP stack, written in the C language, Opensips is extensible by writing C language modules. When configured as a stateless SIP proxy, Opensips documentation claims rate of 13,000 call-attempts/second. In addition, modules exist to provide Back-to-Back User Agent behavior for the system, although no performance benchmark is published to date. In addition to C language modules, which add logic for specific aspects of the SIP protocol, the system uses scenarios that include those defined in the Call Processing Language [RFC 3880]. This high level XML based language, is primarily focused upon SIP and H.323. In addition, there is a provision in the XML for defining explicit states to describe high-level behavior. Opensips is designed around the SIP protocol for a world that is moving towards a single-protocol telephony culture. The system has high performance, a large user base, with software that is released under GPL.

3.12 Esper

Esper[16] is a complex event processing system, built in the Java language [28]. The system accepts streams of events and using a set of triggering patterns, executes stored programs. Esper is considered to be the highest layer of abstraction in a stream processing system, positioned above an event stream processing layer, typified by system such as Storm, which may itself be positioned above an event driven architecture, such as Communication Finite State Machines or Estelle. Esper features a domain specific language called EPL, used to describe events and actions to be taken when the event has been received by the system. This is similar to the function of a traditional Rational Data Base Management System, however, the approach taken by Esper is almost the inverse, since the event patterns (queries) are stored while the data is contained in the incoming streams, so the

15 http://www.opensips.org/ (Accessed June 2013)
16 http://esper.codehaus.org/ (Accessed June 2013)

data comes to the queries rather than the traditional notion of a query applied to the data. As a result of this architecture, there is a natural temporal capability conferred to the system, the system natuarly handles event patterns that include a time windows, such as alerting when a certain number or type of event occurs within a set amount of time.

While the focus of Esper is on streaming analytics, it is an example of a modern high performance Java based system. Using a minimal event pattern, the Esper engine, which has been released as open source, has been benchmarked at 408k events/s.[17] In this case the system was run with no logic statements, but with incoming message traffic that contained 1000, 224 bit symbols. This benchmark was performed on a machine with 4 cores (2 x Dual core Xeon processors).

3.13 Cosmogol

A language that is used to describe the structure of state machines, Cosmogol [13] was submitted as an Internet-Draft in November 2006. Cosmogol consists of defining states, messages and actions. The goal of the language was to capture a finite state machines within a larger specification, rather than to create an executable representation of a state machine. The motivation for the creation of Cosmogol came from the IETF RFC writer's guidelines themselves, as there was (and still is) no standard way to describe state machines in an RFC. The language included a way to describe states. The draft specification included a definition of Cosmogol using ABNF. Since the expiration of the Cosmogol proposed standard, according the the IETF[18]there is not standard language to describe state machines in an RFC.

17 http://docs.codehaus.org/display/ESPER/Esper+performance
18 https://www1.ietf.org/mailman/listinfo/cosmogol

4 An informal description of LiFE

LiFE was originally designed as a transition graph (TG) machine[19] [17]. As the design evolved, capability was added so that the flow engine now corresponds closely to a Mealy machine [36]. The addition of a settable name/value pair list to the engine that can be used to direct state transitions, the flow engine was transformed from a *finite-* to a *"countable"*-state machine [37].

The evolution of the computational model of the flow engine from a TG machine to its present form was driven by the application of the system to real-world problems. The state-machine structure of the flow engine's model corresponds well to many protocol-centric telecommunication services. Enhancements to the basic state-machine model permits the system to describe sophisticated behaviors that depend on a combination of configuration, logic and message exchange. The final form of the flow engine was shaped by the desire to retain a high-level state-machine model, but add enough general processing primitives to solve a wide range of problems.

19 This is due to: J. *Myhill*. Finite automata and the representation of events. *Technical Report WADD TR-57-624,* Wright Patterson Air Force Base, Ohio, 1957, however, although widely cited, obtaining the original report has proven to be elusive.

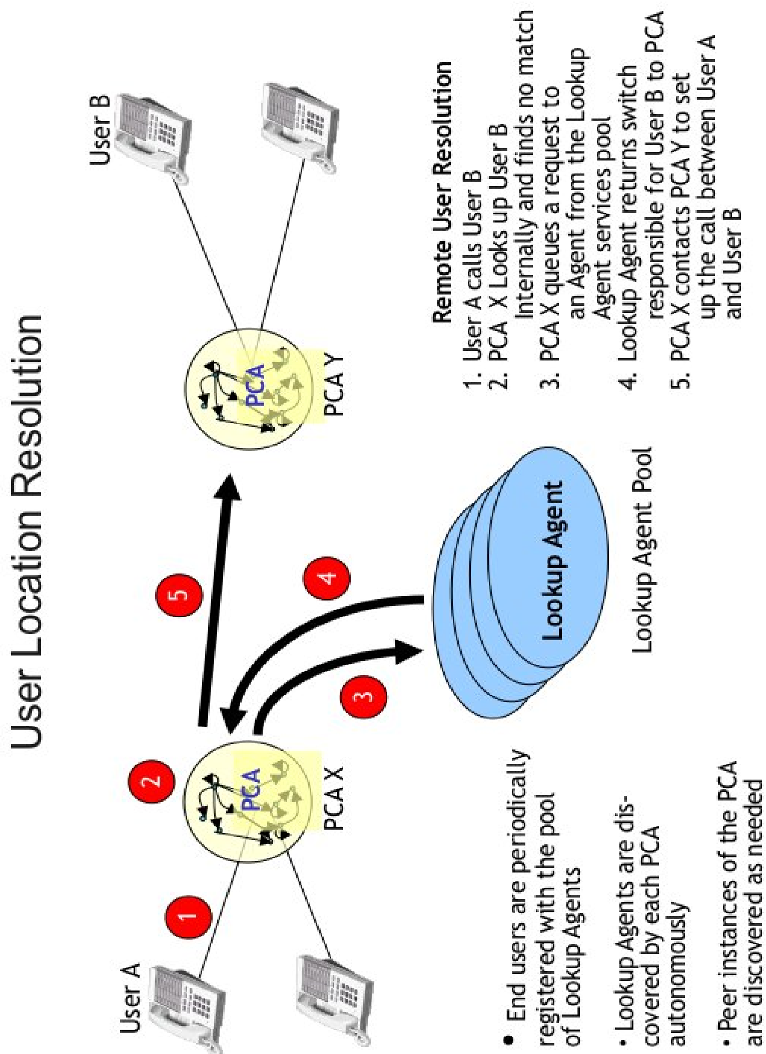


Figure 4.1: User location resolution

4.1 The LiFE model

The flow engine begins execution by reading in a textual graph-definition file specified by a command-line argument. The next sections provide an overview of the graph-file structure and the

elements that are used by the engine to create the internal executable representation of the flow graph. There are two logical parts of any flow graph: a global *name/value pair list*, and a set of one or more *flow-graph nodes*. The convention used in the flow language is to refer to flow-graph nodes as *states*. This is the convention that will be using for the remainder of the informal discussion. Of course, the "state" of a particular context is comprised of the context's place, and the current value of the context's complete name/value pair list.

4.1.1 The Flow Graph Structure

A flow graph may be defined by a collection of files. These files contain the definition of two sections: the global name/value pair list, and the graph states. Just as in languages such as C, a graph file may include other graph files. In addition, the inclusion can be conditional, so that a single file can provide a large degree of configuration over the make up of the final logical graph definition. The basic data structure in LiFE is the name/value pair list. This construct is used in almost every aspect of the system. The conditional inclusion is no exception. Values from the global name/value pair list are evaluated to determine if a file or section of a file will be included in the flow-graph definition. Included files may be nested to an arbitrary depth.

4.1.2 The Global Name/Value List

The first logical section of a flow graph is the global name/value pair list. As the name suggests, this name/value pair list is accessible to all contexts in the system. There is a specific access function used to specify that a name refers to the global list, rather than the context's private name/value pair list. As indicated, the global list is comprised of a concatenation of all of the global lists contained in the collection of included graph files. Usually the graph file supplied as an argument to the flow engine contains a top-level file that contains configuration specific definitions in a global name/value pair list. This file may not even contain any state definitions on its own. The configuration information presents itself as name/value pairs which are either used to include specific portions of other files, such as a

range of states/ subgraph definitions or are referenced directly by the context's as data elements.

A LiFE system can function as a client, a server, or as a mixture of both as part of a defined computation. The global name/value pair list is used to hold the definitions of servers that are to be created, as well as to define external server resources that are part of the current graph definition.

External Service Definitions

Some computations may require external servers to be accessed. These could include access to external databases, or other processes that has been defined outside of the scope of LiFE. To access these resources, the flow engine uses helper programs called Workflow Agents as programmable servers. Earlier it was discussed that multiple Agents may be found by the engine, and that each Agent registers itself upon start up using the portmapper/rpcbind (ONC RPC) [67] mechanism. The address used by the Agent during registration is associated with a service name included as part of the Agent definition. An RPC broadcast is launched for each defined service. Responses are gathered by the engine and marshalled together to create a set of named pools composed of available servers. Computations use the service name, rather than using the address of a specific instance of the service (Agent). Using the pool rather than requiring specific addresses of servers allows the system to support adding and removing resources to and from a running system without the need to modify or manage configuration files.

As discussed, the flow engine can use ONC RPC to perform actions on its behalf. One might wonder if the state machine is running while the RPC call is underway. In fact, the flow engine is still running while the rpc call is underway: timers are running, at the flow engine level. The RPC code was modified to make it non-blocking. RPC requests are launched by the flow engine, and then the results are gathered at a later time, so the state machine that would be normally implemented internal to the RPC routines is exposed in the flow engine model.

An example of how workflow Agents have been used to perform a service, in this case a user lookup service, is shown in Figure 4.1.

Internal Service Definitions

Just as external services are specified using a declaration in the global name/value pair list, LiFE uses a similar mechanism to declare services that can be used by external client programs, such VoIP gateways and client programs. The internal service definitions contain the parameters used by the flow engine to create bound server sockets, supporting TCP, UDP and, to a limited extent, SCTP [55]. In order to make a server useful, the system must:

- have a way to specify how to process messages that come in via the declared sockets;
- determine how to route messages to specific contexts,
- support the association of a set of messages with a context,
- route messages that have not yet been associated with a session; and
- correcly handle retransmissions.

Figure 4.2 shows an example of a declaration of an internal service.

```
"WFA_BUILTIN_SERVICE_CLASS",
      "sip#text#BaseFilter.sip:UDP:SR:5060"
```

Figure 4.2: Example of an Internal Service Declaration

The keyword “WFA_BUILTIN_SERVICE_CLASS” indicates that this line will create a server socket. The remainder of the declaration contains the service name (“sip”), the expected type of input text or binary (“text” in this case). In addition the declaration contains the bootstrap parsing filter, “BaseFilter.sip”, the type of socket (TCP, SCTP or UDP, UDP in this example), that the server will be both sending and receiving messages, and finally the binding port of the service. One might wonder why there is a send/receive option in the declaration. In fact there is at least one service that is “write-only”, and that is the multicast service used to advertise to local resources that a particular endpoint is currently associated with a flow engine instance.

This discussion will begin with datagram services, and then cover

the differences between datagram and stream services. Finally, the rationale and structure of services that use a "plug-in" architecture will be discussed.

Incoming messages are transformed by the flow engine into name/value pairs (tuples). The originating message may have started as readable strings or binary data structures. The input processing of the flow engine will convert these messages into name/value pairs and attempt to route them the appropriate context. A line-oriented printable-string protocol such as SIP will be used to illustrate the processing steps used to transform input messages into a form that consistent with the flow engine's internal model.

When messages that are read from a server socket, each is labeled with the service name associated with the socket declaration. The flow engine then processes the messages to determine if they should be associated with an existing context. In this case, it is convenient to think of the context as a container for the state of the communication session. The engine processes the message to synthesize a predetermined identifier suitable for routing the message. Each declaration of a service contains something called a "basefilter". This file contains the parsing and synthesis rules used by the engine to create the session fingerprint and, just as importantly, assigns the event name for the message. After this stage, each message can be viewed as an event that has a session identifier.

With the session identifier constructed, the engine checks a global hash list to determine if the message is part of an existing session and therefore should be routed to an existing context. But what if this is the first message of a new session? In the case that the hash lookup does not return an existing context, the default is to route the message to context 0. This means that context 0 must be at a state (or have a supplemental state) that can handle the event. In many cases this approach works fine, but in situations when there are a large number of short-lived sessions, the queue length at context 0 can grow very large. As an alternative to routing unassociated messages to context 0, a service may have a specific entry point specified in the global name value pair list for unassociated messages. This directive will create a new context for each new unassociated messages, queue the message to the context, and populate the context's name/value pair

list with the results of the basefilter parse. For some protocols the basefilter parse may be complete enough for the flow engine to construct a reply without the need for more thorough parsing of the input message. However for some protocols, there are certain exchanges that have tighter timing. SIP, for instance, has a very short interval expected between the INVITE request and 100 response[20], usually in the hundreds of milliseconds. In contrast, the time between the INVITE request and the 180 or 183 response time can be much longer, on the order of many seconds. Providing a fast turnaround for the 100 response can eliminate unnecessary retransmissions of the INVITE message. For this situation using a minimal basefilter will minimize the delay between the receipt of the INVITE message and the transmission of the 100 response by the engine.

Once the event has been routed to the context, further parsing of the message may be required. This approach of a two-phase parse of the message allows the message to be processed in a context-sensitive manner. This second parse is controlled by a named filter file at the current state of the context. This facility can be used to avoid name clashes. If a message of the same type, such as an UPDATE, is received at different points during a message exchange, there may be reasons to preserve the data elements of the previous message while also capturing the current values contained in the newly arrived message. An example is a re-INVITE message that may come in during the course of a call. If that message belongs to a hitherto unused Session Description Protocol (SDP), the engine may need to perform an action to renegotiate with the far end. Having a convenient way to determine the changed SDP is especially important if the engine is bridging two dissimilar protocols, such as having one call leg running SIP and the other using H.248/Megaco.

Routing messages from stream protocols poses different challenges from those faced when routing datagram messages. When a stream client using a protocol such as TCP connects to the engine, a new file descriptor is created internally to carry that stream. The creation of a new file descriptor as a transport connection to a client

20 The “100 Trying” SIP message is used as the application acknowledgement to the “INVITE” session initiation message. The exchange is used to illustrate a class of protocol behaviors without requiring a full knowledge of SIP.

removes the need to perform message routing functions. The situation is different for the engine-to-engine connection, as that is used to multiplex messages between any of the contexts that might exist on the flow engine peers. For clients, the simpler condition holds; the file descriptor returned from the connect is associated with a particular context and all of the messages received over that socket are transformed and delivered as events to the associated context.

To aid in the processing of a stream socket, the flow engine supports the use of transport services on top of TCP (TPKT) [62] to delineate the extent of messages. Stream protocols such as SCTP have mechanisms to provide datagrams over its reliable stream, so that mechanisms such as TPKT are not necessary for SCTP connections.

4.2 Graph States

The second logical section of a flow graph is the definition of the individual graph states. Just as in the case of the global name/value pair list, graph states may be defined in multiple files. The conditional inclusion may include portions of a file, so the resulting flow graph will be comprised of all of the states defined in the included portions of each of the included files. As mentioned earlier, graph states may be organized along the lines of traditional finite-state machines, or they may be structured as a set of callable graph segments. In this latter case, a calling stack is added to the traditional DFE organization, and the context may traverse to a callable graph segment, which is called a subgraph, and then encounter a graph return statement that causes the next state to be popped off the context's graph stack. All of the states are labeled. The combination of a state label and and an offset is used to define a context's next state. Once the complete graph has been read into the engine, any state labels referenced are checked to make sure that each can be resolved to a state in the graph.

```
#
DEF_STATE TNTFY(1)
   STATE_NVP=
      "STATE_NAME", "Send\nNotify Ack",
      NULL
   STATE_ACTION=NULL
   STATE_ARCS=
      IS_TIMEOUT=FALSE, TIMEOUT_VAL={0,0},   MATCH_STR="SendAck",
ARC_ACTION =SEND_MESSAGE,
         NVP_LIST=
         "RULE:default", "template.ack",
         "SEND_MESSAGE_OPTION", "SaveReplyInfo",
         NULL
         OUT_STR="St8: OK\r\n", NEXT_STATE={TNTFY(2)}
      IS_TIMEOUT=TRUE, TIMEOUT_VAL={0,750000},  MATCH_STR=NULL,
ARC_ACTION=NULL,
          OUT_STR=NULL, NEXT_STATE={TOP(0)}
      END_ARCS
END_DEF_STATE
```

Figure 4.3: Example of a simple state

4.2.1 Graph State Structure

The example in Figure 4.3 depicts a simple state. Note that in the LiFE language a graph place is referred to as a “STATE”; the state of a context is defined by its *m*-tuple that includes all of the values contained in its name/value pair list.

In this example the state is named “TNTFY(1)”. The name of a state is comprised of a state label and an offset from that state. Each state is required to be explicitly named in this fashion. In the example state shown, the state declared immediately before this one has a state label of “TNTFY “ declared in its STATE_NVP (name/value pair) section.

In this example, two actions are possible: the first is that an event named “SendAck” has been routed to a context that is currently at this state, the second is a timeout action. If nothing is sent to the context within 750 ms of the context arriving at the place, the context will traverse to the place labeled “TOP(0)” in the graph.

If the timeout arc is taken, since the arc action is NULL, the Context will simply traverse to the next place without any other processing.

If the event “SendAck” is sent to the context, then the arc action SEND_MESSAGE will be used to create and send a message. There are several aspects that come into play when a message is created and sent using the SEND_MESSAGE action action: 1) the outbound service will be the same as the context's type; 2) if there is more than one template file declared in arc's name/value pair list, then the system will choose the template file (RULE) that is tagged with the type of the current context; and 3) optional parameters to the specified action (such as SEND_MESSAGE) may be listed.

In the LiFE model, an arbitrary number of contexts may be at any particular graph state. The contexts may or may not be of the same “type”. The situation could arise that an equivalent set of events are routed to a collection of contexts located at the same graph state. This is entirely possible, and in fact occurs in the version of the graph used for softswitch emulation. The SEND_MESSAGE arc action is naturally thought of as a method that corresponds to the context's service type. It is possible to override this association on a temporary basis, which allows a context of one type to send messages from a different service. Since the service is not only the binding, but also the encoding, it means that a SIP context can directly send an HTTP, MGCP or H.248 message, depending upon the flow graph definition.

Finally, the example depicts how parameters are supplied to arc actions. Each parameter is named with the name of the arc action and the parameter name. Order is not important for the parameters. In the example shown, the parameter tells the system that this message is a reply, and the system will save enough information so that it can automatically replay the reply if necessary, without requiring adding additional states to the graph definition to handle the loss of the reply. Additional information about these facilities will be covered during the discussion of the SEND_MESSAGE arc action.

To build upon the example above, a multiplicity of context types may occupy the same graph state. In order to correctly respond to the currently posted event, a particular context may need different data elements in the response message. There are two ways to accomplish the tailoring of a response. One is to use conditional expressions within the template file to tailor the message, the other is to supply a different template file for some or all of the different

context types. The system will choose the template file type that corresponds to the context's type. Consider the excerpt in Figure 4.4 from the SEND_MESSAGE arc action argument list:

```
 "SV:default", "SET:rsip::this:LastCmd",
"SEND_MESSAGE_OPTION", "SaveReplyInfo",
"RULE:default", "template.ack",
"RULE:h248", "ServiceChange.ack.h248",
```

Figure 4.4: Example of SEND_MESSAGE Arguments

If the context type is *h248*, then the system will use the last line, due to the "RULE:h248" value. The template file "ServiceChange.ack.h248" is used to form the outgoing message. If the context had been any other type, such as *mgcp*, or *sip*, the "RULE:default" line would have been selected and the outbound message would be formatted according to the structure contained in "template.ack". The example illustrates a specific case of Arc action overloading [39], in that a context's *type* is used to select the appropriate version of the arc action, and the type is also used to select the set of arc action parameters. In these examples, the name of the template file is a constant, however it is also possible to embed variables in the names of the template files. The variables are taken from the current context's name/value pair list. While this allows another degree of dynamic tailoring of a response, using this facility can result in the creation of a template file that does not exist. If the flow engine encounters a condition in which a specific resource does not exist, a named exception will be raised to the current context. In the case above, if the *h248* context encountered a SEND_MESSAGE arc action, and attempted to access the file "ServiceChange.ack.h248" and that file was not found by the flow engine, and "ILLEGAL_RULE" exception would be sent to the current context at the current state.

4.3 Messages and Exceptions

As noted above, the flow engine graph is composed of a set of nodes. At each place, zero or more events are defined. Messages, whether from external or internal sources are transformed into events. Messages are processed by a context in the order of arrival. If a message is routed to a context at a place that does not contain the corresponding event, then that message is simply dropped. In the flow engine, exceptions are implemented as messages, but with two distinct differences: 1) Exceptions are placed at the front of the context's any-message queue; and 2) if context is at a place that does not have a defined event that corresponds to the exception, the context will be killed. As implied, named exceptions may be fielded by the context, but only if there is an event defined for the particular exception. One could easily imagine that the behavior of exceptions that require an event definition for every place would either result in unexpected behavior or require the graph author to add in event definitions for a large range of graph states. A construct called the "Supplementary State" was created to provide a convenient mechanism to field exceptions in a consistent manner, without requiring the repetitive inclusion of an exception event in each state.

A Supplementary State is declared in the same manner as any other state in the flow graph. The difference arises when the Supplementary State is associated with a regular state. Normally, if an event is received by the flow engine and routed to a context, the current state of the context is examined to determine if one of the ARC definitions contain a MATCH_STR which matches the incoming event. If no match is found, then the event, and the message associated with the event is discarded. If the state has a Supplementary State defined, then once the flow engine has determined that the context's current state does not match the incoming event, the engine uses the ARCs defined by the Supplementary State as a source for possible matches for the incoming event. If a match is found, the flow engine processes the event as if the match was found at the original state. In this way the Supplementary State follows the context as it traverses the flow graph. Supplementary States can be assigned as part of a the name/value pair processing, so that the system behavior can be

changed as result of where in the graph and what type of event is received.

Practical systems must deal with the fact that duplicate messages may be encountered. In some cases, it is important that every message is visible to the flow, but most of the time dealing with duplicate messages can be correctly handled by the lower layer of the flow engine without requiring the definition of additional event handlers (state arcs). Since the flow engine itself is not aware of any specific type of message, a mechanism needed to be added that allows messages to be checked for duplicates. The flow engine allows any message sent by the SEND_MESSAGE verb to be associated with a received message key and tagged as a reply. When a new message is received by the flow engine, the base filter constructs a hash string to identify the message. Once the message has been routed to the owning context, the hash string is used to check to see if this message is a duplicate, that is if the flow engine has already sent a reply message to an earlier copy of the message. If so the reply message is fetched from the context's name/value pair list and automatically sent to the host/port of the originator. The flow engine does not create a new event to correspond to the duplicate message. Therefore, the context will never see the event, and the graph definition does not have the requirement to explicitly code for duplicate messages.

4.4 Arc Actions

In the section above, the SEND_MESSAGE illustrated a number of important aspects that apply to all of the Arc Actions in the flow engine. A set of arc actions have been created to implement a computation model that has enough concise, expressive capability to realize working systems. The following six arc actions are the most important when defining a practical LiFE graph:

SEND_MESSAGE – As discussed in the previous section, this arc action is used to transform members of the context's name/value pair list into an output message. The message may be directed either externally or internally;

PROCESS_MESSAGE – If this arc action is triggered, the message corresponding to the triggering event is transformed by a

context-sensitive filter into name/value pairs. The newly created pairs, update the current context's name/value pair list. The context's type, the current state and the contents of the context's name/value pair list are used to choose the filter;

EVAL_NVL - The arc action “EVAL_NVL” can use any name/value pair from the context or the global list to specify a context to its next graph state. The addition of this arc action changes the basic model of the flow engine, providing a mechanism for the flow graph programmer to reduce the number of states needed to describe a computation. Although the EVAL_NVL was originally conceived as a variation of the C switch statement, it more closely resembles a set of if / elseif statements [60]. Figure 4.5 depicts an example;

```
"EVAL:default", "@Vremark@:restart:EQ:2",
     "EVAL:default", "@Vremark@:forced:EQ:1",
     "EVAL:default", "@VHangupCxt@:NIL:NE:3",
```

Figure 4.5: Example EVAL Statements

GRAPH_CALL – Introduces the notion of a control stack into the flow engine model. In practice, there are many common sets of message exchanges that re-occur within a given flow graph. In a traditional state-machine structure, these state sequences would be repeated, adding to the size and complexity of the graph. In addition any errors in a section of graph that is repeated would have to be corrected in many places. Systems such as AT&T's Q-Plus [51], a graphical queue modeling system featured this concept, as a way of simplifying both the programming and display of complex queue models. When this directive is encountered, the flow engine pushes the current graph state on the context's local graph stack, and the engine directs this context to the graph state contained as the zeroth entry of the NEXT_STATE array. Execution continues at the new graph state. At some point the context will encounter transition arc that has the GRAPH_RETURN flag set. The flow engine will then use the values of NEXT_STATE as indices, rather than the specific next graph state for the current context. The index is used to select one of the values of caller's NEXT_STATE array. Figure 4.6 illustrates this concept;

```
at a the returning transition arc:
  NEXT_STATE={0,1}
with a calling transition arc:
  NEXT_STATE={FREEINX(3), FLASH(13), CWDIAL(0)
```

Figure 4.6: Example of Return Values Declaration

CREATE_CXT – This is the way new contexts are created in the system. When the engine is started, exactly on context is created at state zero. All other contexts in the system are a result of the execution of the arc action. Much like Unix's fork [70], this arc action has the ability to allow one context to create additional contexts that contain a copy of all or part of the parent's name/value pair list. The new contexts may be created so that the relationship is maintained with the parent or the context may be disassociated from the creating context. Creating a context this way does not create any handle lookup associations, that is left to the graph programmer. New contexts are created at a specific named state by using arguments to the CREATE_CXT arc action.

4.5 Arc Name/Value Pair Directives

Since the flow engine uses name/value pairs to capture much of the state, it is critical to be able to manipulate the name/value pairs. The name/value pairs are natively strings. Just as in languages such as "awk" [1], name/value pairs as variables do not need to be declared, but are brought into existence by reference.

As an example, the statement: "SET:@Vfoo@::this:NewFoo". If the context has the pair "foo", "bar", then this assignment will create a new name/value pair with the name "NewFoo" and set its value by first evaluating the current context's NVP that has the name "foo" and substitutes the value (which would be "bar"), as the new NVP's value. As a result there would be a new NVP "NewBar" , "foo" contained in the context's NVP list. Although our example shows a single constant string or a single variable, the evaluation of the

arguments performs a concatenation of any substrings to form the argument used for the operation. As an example, given: {“car”, “528i”}, {“bike”, “s1000rr”}, an argument defined as: @Vcar@IsFasterThanA@Vbike@ will result in the nonsensical result of: “528iIsFasterThanAs1000r”.

There are four categories of arc directives: 1) assignment, which includes “SET” and “DEL”, the delete function; 2) conditional assignments; 3) string manipulation, and 4) set/list operators.

Cleaning up the name/value pairs can also be done conveniently using the ”CLEAN_NVL” directive. Using this directive, a list of all of the names associated with name/value pairs are listed. Only the tuples that have the names on the list are retained, all of the others are destroyed. The list of names can also be a variable, so it is possible to set up a set of variables that should be retained and periodically clean any other tuples off of the context's list, without needing to enumerate each deleted tuple. Assignments can be made conditionally, so that a change to a name/value pair is dependent upon an independent pair in the context's list.

4.6 String Manipulation

As previously mentioned, all of the name/value pairs are stored as strings. Only very simple string operations have been mentioned so far, however the manipulation of protocols messages can and do entail more complex string operations. The use of REGEX allows a graph programmer to perform more sophisticated transformations on the values. The REGEX capabilities contained in the arc directives have been transplanted from the BSD *regex* (regular expression) library [2]. The facility in the flow engine exposes all of the *regex* operators however, the input is restricted to a value from the current context's name/value pair list. In this case the example below is helpful to illustrate some of the capabilities.

A number of the textual protocols use lists to describe capabilities, resources or current end point state (SIP, MGCP, H.248 all do this). One of the design goals of the flow engine is to create a programming model that closely matches the way in which protocols are described

and used. The goal of these set operations is to take a list that is contained as the value portion of a name/value tuple and allow the graph programmer to perform a range of transformations on one or more of the list elements, and perhaps to create an alternative list which is sent back to an endpoint as a reply.

A natural construct to accomplish some of these tasks is through an iterative control loop. This is certainly possible in the LiFE language. A flow-graph programmer could create loops using a succession of states with a loop termination defined by a variable contained in the name/value pair list. As an alternative, the LiFE language defines a set of operations that are designed to take the place of common string operations the would normally be performed using iteration.

Accordingly, the flow engine has a set of operators that permit the manipulation of lists. In the flow engine, lists are modeled as sets. Facilities exist to create a set from a value that contains a list, select one or more elements of a set, count the number elements in a specific set, transform selected data elements in a set, and create a list from a set. These operators are listed below:

UNPACK – transforming a list into a set of name/value pairs

PACK - marshaling the set of name/value pairs back into a single value.

SELSET – selecting values from a set of related name/value pairs

CHGSET – updating the value of a set of related name/value pairs

4.7 LiFE and Turing Machines

A Turing Machine [65] is a useful benchmark to understand the expressibility of a system. In this section, the example from Turing [71] will be used to show a way that LiFE can implement a Turing Machine.

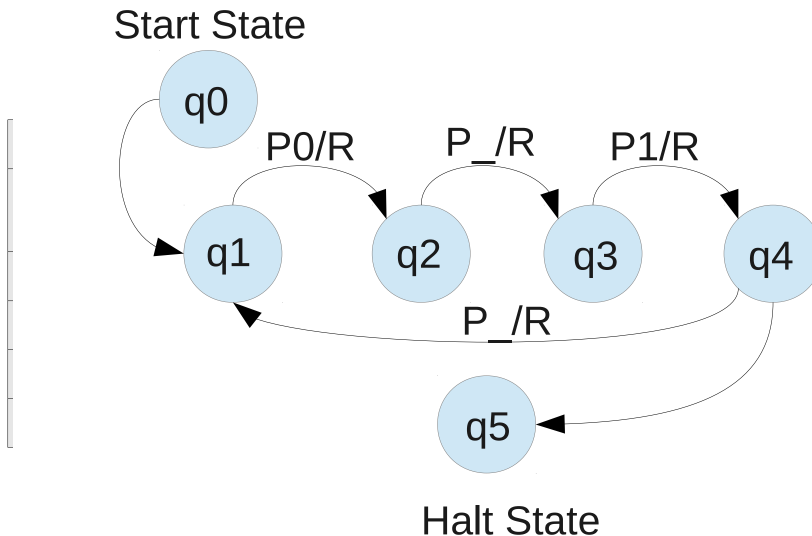


Figure 4.7: State machine with Start and Stop states

Turing's example [71], will be used to illustrate a way that problems can be expressed using the flow engine. Alan Turing described a machine that can be constructed to compute the sequence “0 _ 1 _ 0 _ 1 _ 0 _ 1 ...”. That is a sequence that consists of alternating zeros and ones with a blank in between them. Turing's machine consisted of an infinitely long tape that was comprised of discrete locations. A head could read the current value of the location directly under the head, erase the contents of the location, or write either a 1 or 0 in the location. The machine could also advance the tape either one spot to the right or to the left. Turing used a set of starting states, conditions, tape operations, and final states to define the computation. This is shown in Table 4.1; an alternative representation is given in Figure 4.8.

From this representation, the service logic for a flow engine program can be created in a straightforward manner. A necessary part of Turing's machine is an infinitely long tape, on which to move, read and write. Each flow-engine context has a name/value pair list.

Both the names, and values of the list have no explicit limit, so a name/value pair with the name of “tape” can be used to serve as Turing's tape. As in numerous languages, such as awk, the name/value pair is created upon its first reference, therefore, the first Print (P) operation to the “tape” object, in this case, setting the value of “tape” to a blank, will bring the variable “tape” into existence.

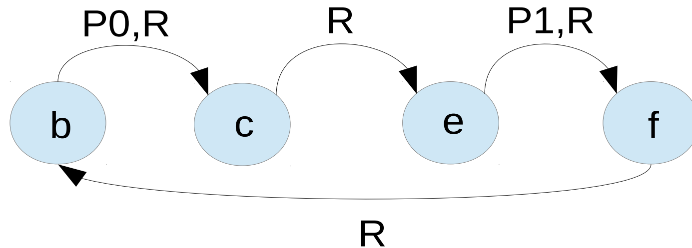


Figure 4.8: A more compact representation of Table 4.1

The reader will notice that Turing's table has neither a formal *start* state, nor a *halt* state, so this machine will run forever. The flow engine does have an explicit start state, but it can be constructed to have no halt state. So in our example a starting state *a* will be added to the table. That state creates a blank tape and then proceeds to the state *b*, one could also initialize a blank tape of a specified size, prepopulated to all blanks. As an alternative, one could modify the table to write a blank rather than the original skip of a pre-existing blank. In the following diagram, the states are relabeled in the form: $q(0)$, $q(1)$ ....

One of the first things to notice is that there is now a halt state (state *d*). More significantly, there are two arrows from state *f*, one that prints a blank and advances the tape (grows the tape), and another that connects state *f* to the halt state. This latter transition is not part of Turing's machine, but allows the programmer to choose

```
DEF_STATE q(0)
            STATE_NVP=
                        "STATE_LABEL", "q"
                        NULL
            STATE_ARCS=
                        MATCH_STR=NULL, ARC_ACTION=NULL,
                                    NVP_LIST=
                                                "SV:default",  "SET: ::this:tape",
                                                "SV:default",  "SET:0::this:LoopCnt"
                                    NULL
                        NEXT_STATE={q(1)}
                        END_ARCS
END_DEF_STATE

DEF_STATE q(1)
            STATE_ARCS=
                        MATCH_STR=NULL, ARC_ACTION=NULL,
                                    NVP_LIST=
                                                "SV:default", "SET:@Vtape@0::this:tape"
                                    NULL
                        NEXT_STATE={q(2)}
                        END_ARCS
END_DEF_STATE

DEF_STATE q(2)
            STATE_ARCS=
                        MATCH_STR=NULL, ARC_ACTION=NULL,
                                    NVP_LIST=
                                                "SV:default", "SET:@Vtape@ ::this:tape"
                                    NULL
                        NEXT_STATE={q(3)}
                        END_ARCS
END_DEF_STATE

DEF_STATE q(3)
            STATE_ARCS=
                        MATCH_STR=NULL, ARC_ACTION=NULL,
                                    NVP_LIST=
                                                "SV:default", "SET:@Vtape@1::this:tape"
                                    NULL
                        NEXT_STATE = {q(4)}
                        END_ARCS
END_DEF_STATE

DEF_STATE q(4)
            STATE_ARCS=
                        MATCH_STR=NULL, ARC_ACTION=EVAL_NVL,
                                    NVP_LIST=
                                                "SV:default", "SET:@Vtape@ ::this:tape",
                                                "SV:default", "INC:1::this:LoopCnt",
                                                "EVAL:default", "@VLoopCnt@:@EMaxCount@:GE:1"
                                    NULL
                        NEXT_STATE={q(1),q(5)}
                        END_ARCS
END_DEF_STATE

DEF_STATE q(5)
            STATE_ARCS=
                        MATCH_STR="XXX", ARC_ACTION=NULL,
                        NEXT_STATE={q(5)}
            END_ARCS
END_DEF_STATE
```

Figure 4.9: Flow Language Equivalent of Figure 4.7

the number of iterations that this machine will perform before ending its execution. At the start state, a counting variable is referenced, both creating it and initializing its value to zero. Each pass through

state *f*, increments that counter. When the counter equals a predetermined value, the final state will be set to *d* rather than *b*.

It has been previously mentioned that one can model Turing's tape using the value portion of a name/value pair contained on the context's name/value pair list. The tape could also be modeled using an external file, instructing the flow engine to append the proper sequence to the file.

Each of the states defined in the flow graph language shown in Figure 4.9 corresponds to one of the states shown in Figure 4.7. All of the states except *q*(4) drop right through to the next state. State *q*(4) contains the logic used both to increment the counter (LoopCnt), and to use the value of LoopCnt to determine the next state. The constructs shown here act like the C expression LoopCnt++, in that, since the "SV" directives are always executed before the ARC_ACTION, the variable will be incremented before it is tested. The test, in this case, is "GE" or greater than or equal. This will evaluate to TRUE if the first expression evaluates to a value lexically greater than or equal to the second.

State *q*(3) in Figure 4.9 uses the statement: *"SET:@Vtape@1 ::this:tape"* to add a "1" to the tape. The flow engine uses strings as the default way that values are represented for both names and values. In the example shown, accessing the value of "tape" is done using the 'V' operator. Matching '@' characters are used as anchor points, delimiting an operator. Thus the portion @Vtape@1 is evaluated to: the current value of tape, concatenated with a "1". The SET directive creates or updates the name/value pair corresponding to "tape" with <*old value of tape*>1. In this way the flow graph programmer can emulate an element such as a tape. For all practical purposes in this example, the length of the value is unlimited.

In the example above, which is quite artificial, it has been shown that the computational model of LiFE allows one to code a program directly from a state machine representation. But what about real world problems? Does the model used by LiFE have any advantages in capturing real-word complex problems? In the experience of the author, the answer is affirmative. Business processes are routinely expressed as "work-flow", as testified by languages such as BPEL and

WS-BPEL (discussed above in related work). It is interesting to note that a system used to realize business processes has different performance requirements from a system used for interactive communication (such as VoIP). Business processes may have artifacts that reside within the system for long periods of time (order of days to weeks); these could be transactions that require a number of parts that are presented to the system over a long span of time all which are needed to allow the transaction to complete. A transaction may have an exceptional condition that requires a number of steps that are performed by external users before it is deemed complete. In these cases the system must preserve the data so that all transactions survive the system being taken down due to a hardware fault or routine maintenance. Communication systems rarely have active elements that survive beyond 24 hours. Rather the sessions are set up and torn down usually within 3 minutes; in fact, the standard used to measure the capability of telecommunications equipment involves the number of calls the system can handle, with residential calls estimated with a duration (hold time) of 180 secs, and business calls with a hold time of 90 secs. These metrics provide insight into very different expected lifespans of call sessions in contrast to session transactions.

The computational model provided by the flow engine allows one wide latitude in terms of realizing communication sessions. As an example, two different models were implemented to support VoIP testing. It happens that the stimulus/response protocols (MGCP & H.248) were implemented as a set of contexts in which each context represented a physical line. Communication sessions were modeled as sets of name/value pairs, related by a session key within the name/value pair list.

In the case of SIP, a functional protocol, a context was used to represent each physical line, but an additional context was created for each communication session (active or inactive). As one might suspect, the former model used less compute and memory resources, but even using the less efficient model, the flow engine can sustain more than 2,200 full calls per second, running as a back-to-back User Agent. To give some perspective to this number, the usual sizing for a telephony server is set at an over-subscription ratio of 9:1. If this

machine were to be used for residential service (180 second hold time), one instance could support: 2.2 K x 180 x 9 or 3.5 million subscribers. So about 5 units would support the population of the Netherlands. This benchmark was in single mode (non-protected) running on an Intel i7 920 3.1GHz desktop PC.

Using a modern server configured with dual Intel E5-2690 8 core 2.9 Ghz CPUs with 64 Gbytes of main memory (hyperthreading off), the heavily multi-threaded architecture (v 4.4.9), can sustain 6500 full calls per sec (13.000 half-calls per sec). In this configuration the system reports the use of about 7.5 CPUs to sustain the 6.5k cps rate.

4.8 Influenced by Data Flow

The flow engine contains mechanisms such as *rendezvous*, and the main thread schedules contexts based upon ready events. Experience has shown that context scheduling has a significant impact upon system performance (e.g., first in – first out *vs.* any ready context) An essential aspect of the “Data Flow” model of computation is that rather than focusing upon control, a data flow model focuses upon the flow of data [22],[46]. In fact at the heart of LiFE is a dispatcher that reads events (the data), routes events to contexts that are ready to run. An important aspect of a data-flow approach is that the computation is purely based upon ready events. If one strictly followed this approach, the flow engine would simply fire any transition that is ready. Experience has shown that once the system experiences a significant amount of offered load, sessions will experience a large variance in response time from the engine. Often there will be enough delay to trigger re-transmission of messages from the external systems, adding additional load into the flow engine. The additional load creates an even larger variances in engine response times, exacerbating the situation. While the original design of the flow engine was influenced by data-flow concepts, events are kept in strict order internally and limits are placed upon the number of transitions processed successively for any particular context. These changes result in a stable, responsive system even when the threads are running at over 95% CPU utilization.

While finite-state automata are a convenient way to model many

protocols, adhering to a strict FSA model has its drawbacks, including dealing with retransmissions, duplicate messages, unexpected messages, modeling exceptions and the large number of repetitive state sequences. In the course of creating and using LiFE, mechanisms have been created that greatly simplify creating a state machine representation for certain classes of protocol problems.

4.9 Lazy Linking

As stated before, a central idea in the flow engine is to create a system that can be extended without recompiling the engine code itself. Previous discussion has covered the idea of a filter to transform incoming text messages into name/value pair lists. For a number of protocols, the approach outlined above, one that relied on line oriented textual messages, was an effective, efficient method of parsing messages into name/value pairs attached to specific tokens. Protocol messages, such as SIP & MGCP can be handled in a straightforward way, however, some protocols, such as H.248/MEGACO, pose a bigger challenge. This VoIP protocol is much more complex to parse.

When the requirement to support H.248 arose, one choice would have been to build a significantly more complex parser into the flow engine. But rather than write yet another parser for H.248, the goal was to leverage the existing H.248 library. In addition, a design goal was to avoid hard-coding whenever possible, in this case the goal was to find away around the need to hard-code the mapping of any of the library data structures into the flow engine code. A design that generalized a flexible mapping between library data structures and name/value pairs would allow the system to accommodate any number of protocol libraries into the flow engine without touching the code. Rather than having the assignment logic coded into the library as would be done in a plug-in, I wanted to use the same approach that was used by existing textual protocols, and have a context-sensitive assignment description file used to create name/value pairs from specific messages (at specific states). All but the context-sensitive assignment ended up implemented in the flow engine. A side benefit of realizing the H.248 library this way was

providing parsing support for either binary or text encoded forms with the same processing. Either form presents to the flow engine as the same data structure, with a bit set to note whether this message was sent in the binary or text form.

An approach was suggested by the way in which symbolic debuggers work [45], [73]. Modern debuggers allow access to arbitrary data structures within a program image. Using a symbolic debugger one can simply declare that a portion of memory should be interpreted as a known structure, and then examine the memory contents. The debugger will format the contents of memory for display, consistent with symbol information read in at the start of the debugging session. If there is a 32-bit integer named *foo* at an offset of 64 octets from the start of the structure, then, after assigning the base address as a pointer to the data structure, one can dereference *foo* and the 4 octets at base address + 64 and it will be interpreted as a 32-bit integer and displayed as a string. Taking this simple notion, a facility was added to the flow engine which allowed the H.248 library to be linked (in this case at link time, but below one can see that it could be done later) and using a symbol table generated using standard utilities (nm, or pahole [21]), provides access to any of the H.248 data structures selectively using a description file.

In analogy to debuggers mentioned above, the flow engine makes use of the symbolic resources at run time to make the connection between the existing object file (library) and the filter file. Performing this as late linking at run time, instead of a compile-time mechanism, is consistent with the “late-binding” architectural theme of the flow engine.

```
../../include/h248_api.h:
typedef struct _TransactionRequest /* id 125 */
TransactionRequest;
struct _TransactionReply { /* size 20 id 126 */
  struct %anon127 { /* size 4 */
    unsigned int immAckRequiredPresent; /* bitsize 1, bitpos 0 */
  } m; /* bitsize 32, bitpos 0 */
  TransactionId transactionId; /* bitsize 32, bitpos 32 */
  int t; /* bitsize 32, bitpos 64 */
  union %anon128 { /* size 4 */
    ErrorDescriptor *transactionError; /* bitsize 32, bitpos 0 */
    ActionReply *actionReplies; /* bitsize 32, bitpos 0 */
  } u; /* bitsize 32, bitpos 96 */
  ASN1OpenType *extElem1; /* bitsize 32, bitpos 128 */
};
```

Figure 4.10: Example of a symbol file

As an example, Figure 4.10 includes a portion of the symbol file created by object-file processing utilities such as pahole [21] from the H.248 library.

As shown in this example, the symbol file includes several parameters that are used to interpret the underlying memory, including the bit position and the size of the object. When this information is added to the structure member declaration, the flow engine can format each structure member into a string that can be stored in a token's name/value pair list. Since the symbol table is constructed mechanically, it is complete, however there may be members of the structure that are not useful to store in a token's name/value pair. So something else is needed to identify which data structure members should be captured in the token's name/value pair list. In addition, there may be name clash, depending upon how the library author named various data structures, so there should be a way of uniquely constructing the name used as part of the saved data tuple.

Figure 4.11 shows a portion of a filter file used to select and store members of the TransactionReply structure created by the H.248 library as output of message parsing.

```
#
#   TransactionReply.h248.filter
#
#
#
%s TransactionReply
transactionId=TransactionReplyId
%x t %union u
%v 1 ErrorDescriptor transactionError
errorCode=TransErrorCode %flags:EventName
errorText=TransErrorText
%e
%v 2 ActionReply actionReplies
contextId=MsgContextId
%c ErrorDescriptor errorDescriptor
errorCode=ActionErrorCode %flags:EventName
errorText=ActionErrorText
%e
```

Figure 4.11: Example of a filter file

The flow engine is designed to read in each of the filter files as elements are referenced, so that the TransactionReply.h248.filter will be read into the flow engine as necessary if, for example, an H.248 message is read in that contains a Transaction Reply. As part of the service definition for H.248, a file is included called "Base.h248.filter". This is the root for all of the messages that will be decoded as part of the service. All of the messages that are processed by the H.248 library are described as part of this data structure. This does expose a current implementation limitation on the current engine; that is, all messages are presented to the flow engine as a variation of a common top-level data structure. The current implementation does not provide for a library that may present some number of disjoint messages. There are several current implementation limitations that could be overcome in a straight-forward manner as the implementation is generalized. For instance, right now there three H.248 routines that are referenced directly in the code, one to initialize the stack, one to configure debugging, and one to call the parsing routine with the input message. Since most, if not all,

message-processing libraries would also have the same categories of functions (initialization, configuration and parsing), it is straightforward to read these functions into the engine at run time, as is done as part of the plug-in facility to be described later.

When an external H.248 message is received by the flow engine, by a service (server socket) associated with the H.248 protocol, the engine will call the H.248 library to parse the message. The input message buffer and a portion of anonymous memory (malloc'ed memory) is provided as arguments. The library routine creates a data structure in the supplied buffer that represents the parsed message. The output H.248 data structure may have lists, arrays and arbitrary levels of nesting. The complexity of the output data structure is dependent upon the input message. As a result, the H.248 filter name/value pair construction facilities have ways to insure that the names constructed from a complex data structure do not clash, so that, for example, in a list, successive elements do not over write each other.

The facility is not specific to H.248, but could be used for any set of library routines. The approach can be contrasted with that of a plug-in. A plug-in requires that code is created that is specific for use in the flow engine. This approach allows the use of a common library as is, with very little additional code created for each library. All that is needed is a mechanism to initialize the library, send a buffer to the library and identify a return pointer that contains the resultant data structure. More importantly, if the data structures or functions are changed in the library, the flow engine does not have to change at all. At most one would have to re-run the structure mining software (nm, pahole) [21] to pull the data structure names, offsets and sizes out of the object file (since they may have changed), and potentially change the filter file to take advantage of any new data structure members. But if there was an addition to an H.248 library data structure that does not need access, one would only have to run the structure mining utility on the updated library in order to use it. In fact this happened a number of times in practice.

Many protocols are sent over the wire in a binary or encoded form. The messages used to communicate between the eNodeB (eNB) and the Mobility Management Entity (MME), both part of the wireless

Enhanced Packet Core, use ASN.1 coding [41]. In order to for LiFE to play the part of an MME, it would have to receive and transmit binary ASN.1 messages over the wire.

There are multiple ways to add the capability to send ASN.1 messages to LiFE. The first could be writing a module that directly supports all of the MME messages into the code base of the engine. The module would be a new service that could be instantiated and bound to various ports. Any change to the message set, would require that engine code would have to be changed. This runs counter to the design principles of the engine, that is the goal is to build a platform that is not tailored to any one protocol, but can be extended to support new protocols without a need to change the flow engine each time. A second approach could be to create a *scriptable* ASN.1 encoder/decoder, and allow a user to instantiate a service that encodes messages from name/value pairs, and decodes incoming ASN.1 messages into a set of name/value pairs that is routed to a particular context. This latter approach is consistent with the existing textual-processing capabilities of the flow engine. This approach may still be implemented, but, instead, a third approach has been taken, the implementation of a plug-in capability.

4.10 The Plug-In Architecture

Many modern software applications are structured as extensible engines [74], browsers are great examples of applications that are routinely extended by adding additional code, without requiring a recompile or complete upgrade. The flow engine has also been designed to be extended without the need for a recompile or a traditional re-linking. It has been established that new messages can be parsed and produced by the flow engine without the need to change or recompile the engine's source code. The biggest limitation to the most flexible approaches outlined above is that they apply only to text messages.

```
  "WFA_BUILTIN_SERVICE_CLASS",
      "mme#plugin#lte_mme:UDP:SR:7522"
```

Figure 4.12: Example of a plugin declaration

The flow engine uses a plug-in mechanism to support a means of processing binary-message in a flexible manner. Each new plug in will convert the message representation from its external form, to a set of textual name/value pair sets. Along with the decoding / parsing method, the plug in will also contain a set of routines to create an external message from a specific set of name/value pairs. A transformation library can also exist for encoding a context's name/value pair set into a message suitable for transmission to the engine's peer. A set of access routines are provided to the plug-in author, that provides the ability to read and write a context's name/value pair list. Plug ins are declared as in the example of Figure 4.12.

In this example a service named “mme” has been declared. It is a UDP service, capable of send and receive, bound to UDP port 7522. The parameter “#plugin” indicates that this service needs a plug in to be activiated, and “#lte_mme” indicates the name of the dynamically loadable library that must contain the plugin transformation entry point routines. Clearly the libraries will contain a rich set of functions, needed to carry out the necessary conversions between the particular message structure and the name/value pair presentation of the equivalent data sets. At start up time, the system will search for two files: one is a table of contents file, in this case “lte_mme.toc”. The second file is the plib file, in this example “lte_mme.plib”. The toc file lists the four entry points that are needed by the engine to use the plugin. These are the ATTACH, INPUTCB, OUTPUTCB and DETACH EntryPoints. The table of contents (toc) file lists the C language routine for each of the four methods. The toc file also lists the name of the plug-in library object file, known as the PluginLib. The system will scan the object file named in the toc file for the named entry points. If those four entry points are found in the file, then the service can be activated.

If found, the system will then activate the service, by creating a thread that will execute the plugin. In a callback fashion [20], the

INPUTCB routine will be called with octets sent to socket created as part of the service definition. In our example above, any datagrams sent to UDP port 7522 will result in a call of the mme service INPUTCB routine. The received octets will be contained in the inmsg argument, and the length in the msglen argument. This is consistent with systems such as Sun's RPC XDR routines [68]. The third argument in the INPUTCB routine is a pointer to a name/value pair list. This list is created for each incoming message, and the role of the INPUTCB routine is to parse appropriate portions of the incoming message and create name/value pairs for the important portions of the message. While the INPUTCB routine is responsible for transforming the incoming message into name/value pairs, the incoming name/value pair list comes to the routine with the sender's host address and port already loaded.

When a plugin is created for a service, three access routines are supplied to the plugin creator: addToNvl, delFromNvl and getValueNvl. The results of message parsing placed on the list using these supplied routines. It should be pointed out that the system expects that the name/value pairs are encoded as strings. When the INPUTCB routine returns, the name/value pairs on this list are processed to determine to which token the NVP list will be added. As part of the input message processing, the INPUTCB routine should have set a name/value pairs that are used to: 1) route the result of the input message to the proper token [DemuxID]; 2) uniquely identify the message [MessageXID]; and 3) set the event name [MessageId] that will be used to trigger flow engine actions.

The plugin software supports the synthesis of messages from tokens in much the same way as it is accomplished for textual messages. When the SEND_MESSAGE method is encountered, the token's current name/value pair set is made accessible to the OUTPUTCB method. The output method is provided with a pointer to a buffer, and a pointer to a size. In addition, the output method is provided with an “output_name_string”. This value is copied from the current token state, and may be used to signify context (beyond that provided by the name/value pairs) for the output message. The called method is expected to make sure that the HOST_ADDR and HOST_PORT name/value pair tuples have been correctly set for the

output message. Although the values on the name/value list are usually strings, the output message created by the OUTPUTCB method can be any set of octets. The current implementation limits this buffer to the maximum length of a UDP message.

In this manner new services can be added to LiFE without adding additional code to the main engine. At this time, plug-ins are written in the "C" language, however, it should be possible to extend the library creation to at least C++ and Objective-C in a straight-forward manner.

In this chapter, many of the aspects of the LiFE language have been covered. An example of using LiFE to implement a Turing Machine was illustrated. In addition, it was shown that the expressive power of the flow graph language opens the possibility of expressing a wide range of solutions to computational problems. Finally, a number of language features were covered to illustrate mechanisms designed to realize complex service logic without the need to use an auxiliary language such as C or Java.

5 The computational model of LiFE*

(This is from the previously published paper: *"Portable Call Agent: A Model for Rapid Development and Emulation of Network Services" BLTJ 12(4))*

The Portable Call Agent (PCA) runs on a variety of hardware platforms ranging from embedded systems such as IBM's PowerPC* architecture, to PCs with Intel architecture, SPARC* workstations, and Precision Architecture Reduced Instruction Set Computing (PA-RISC*) architectures. It has been ported to a variety of software platforms, including Linux*; Solaris*; AIX, IBM's version of UNIX*; and HP-UX, Hewlett Packard's version of UNIX. Despite the general and platform-agnostic nature of its design, it has proven to be highly efficient in its performance. On an Intel Pentium* III 800 MHz PC (as an illustration of a baseline low-performance platform) running Linux, it achieved a performance of more than 100 call legs per central processing unit (CPU) second. It is also quite scalable, supporting hundreds of millions of calls each of which may use any one of the three protocols MGCP, H.248 or SIP. Finally, its memory footprint has proven to be stable—over a six month test period, the MGCP implementation used a 1.5 MB base with 20 kilobytes per active line.

5.1 A Model for Specifying Network Services

The input message parsing and output message generation files for the PCA are written in a form that closely resembles Unix shell scripts. The heart of the service processing is specified in the call flow definition file, using a language defined in PCA. In this section, we present a model that distills the key concepts and programming primitives of this language and the semantics of the call flow definition files written using the language.

5.1.1 Informal Overview

In the call flow definition file, a network service is described as a graph consisting of states and state transitions. Multiple concurrent threads execute over this graph, with each thread's execution consisting of traversing the set of defined states while following the state transitions. A transition describes the conditions under which the transition may be followed causing a thread's execution to move from one state to the next, as well as additional actions that take place as a result of following the transition. Typical conditions involve the receipt/processing of an event—events can be messages, timeouts, or exceptions. Transition actions can consist of sending messages to other threads as well as the spawning of new threads. Additionally, each thread has its own data state. As previously mentioned, it is natural to view the data content in network messages abstractly as a list of name-value pairs. We regard the data state associated with each thread uniformly as also consisting of a list of name-value pairs. Transition conditions can therefore additionally specify constraints on the thread's data state, and transition actions can update the data state as well. One of the states specified in the graph is specially designated as a start state. When the model (call-flow definition file) is loaded and instantiated, one thread is created to start in this specially designated state. All other threads then arise as a result of subsequent transition actions taken either by the initial thread or the other threads it spawns.

5.1.2 Formal Model

We present our definition of the formal model in stages, progressively adding computational features. Formal semantics can be defined in terms of suitable configurations and transitions between them for the full set of features presented below. Due to space considerations and in the interest of readability, we restrict ourselves to an intuitive description of semantics in this paper.

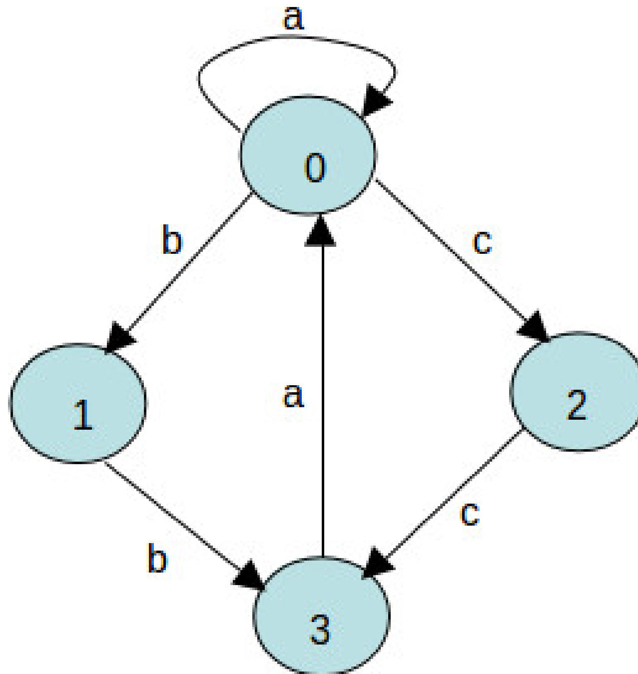


Figure 5.1: Example of a Simple State Machine

Labeled transition graph.

The basic underlying structure to the specification of the network service is a definition of control states and transitions between them. We represent this as a labeled transition graph defined to be a tuple (Q, Σ, δ, q), where Q is a set of states, Σ is a set representing labels on transitions, $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation, and $q \in Q$. Intuitively, the elements of Q represent the control states and the special state q represents the initial state. The elements of the set Σ represent possible labels on the transitions. The transition relation δ is

a set of elements of the form (q_1, σ, q_2) where q_1, q_2 ∈ Q, σ ∈ Σ; such an element represents a transition labeled σ from state q_1 to the state q_2. It thus defines a graph whose nodes are given by the set Q, and with labeled directed edges given by the transition relation δ. An execution in the labeled graph is a labeled path in the graph starting in the initial state q.

To provide the reader some working familiarity with our notation, we consider the example given in Figure 5.1. The labeled graph would be represented in our notation as follows:

The set Q = {0,1,2,3}, corresponding to the four states in the graph.

The transition labels are given from the set Σ = {a,b,c}.

The transition from state 0 to state 1 with label b would be represented by the tuple (0, b, 1).

The complete transition relation δ = {(0,a,0), (0,b,1), (0,c,2), (1,b,3), (2,c,3), (3,a,0)}, containing six tuples corresponding to each of the six transitions in Figure 5.1.

As we have defined so far, the transition labels in Σ are completely uninterpreted. Our model will be defined on top of a labeled transition graph by refining the syntactic structure of the transition labels with associated semantics for execution. The general syntax of a label σ is given by:

σ ::= *guard* → *action*

For a transition label σ = *g* → *a* (where *g* is a "guard," and *a* an "action"), the meaning of a transition (q, σ, q′) is as follows:

If a thread is currently in state q and the guard condition *g* is satisfied,

then the action described in *a* is executed and the thread moves to state q′.

In the following sections, we define the syntax and semantics of the guard and action components of transition labels.

5.1.3 Asynchronous messages and concurrent threads.

Threads can receive and send messages. Each message consists of a message name and data that is sent with the message. Additionally, a data state is associated with each thread at any point in its execution.

The guard component of transition labels is given by the following context-free grammar:

guard ::= **<** *message-match , val-cond* **>**
message-match ::= *x* **?** *m(y)* | ε

The *guard* in each transition consists of two components: a message match condition (defined by “message-match”) and condition on the data values (defined by “val-cond”). The syntax of a message match is of the form *x ? m(y)*, where *m* is a message name and *x,y* are variables. The condition is satisfied if the message received has message name *m;* the variable *x* gets bound to the sender of the message and *y* gets bound to the data value sent with the message. Optionally, the message match condition of the guard can be specified as ε to indicate an empty requirement on the message (i.e., the transition is enabled without a specific message being received). More precisely, each thread’s execution consists of moving through a sequence of state configurations. A state configuration has three components: (1) the current control state of the thread, (2) the queue of messages that the thread has received (but not yet processed), and (3) the current data state of the thread. For a transition with a guard of the form *x ? m(y)* to be executed, the message at the front of the thread’s message queue must have name *m*; in this case, the message is removed from the thread’s queue and variables *x* and *y* become bound to the sender and data value of the message. These variables are then available for use in the rest of the transition label as we shall further see. If the message-match component of a guard is ε, then the transition can be taken independent of the contents of the message queue and no message is removed from the queue. The condition on the data values, defined by val-cond, will be defined more precisely later but intuitively, they prescribe some predicate on the data values received in the message, together with the current data state of the thread. For the transition to take place, this predicate must be satisfied. For example, using an informal notation, the guard < *x ?* ack*(y)* , *y = 1* > requires that an ack message has been received and that the data value sent with it is 1.

The action in a transition has three components: a list of messages sent, a list of thread actions (such as creation of new threads), and updates to the data-state of the thread. It is therefore syntactically

defined as follows:

action ::= **<** *message-send-list , thread-action-list , lvp-exp* **>**

The data state maintained in a thread is a list of name-value pairs. The last component of the action lvp-exp will evaluate to such a name-value pair list; the thread's data state is then updated to this value. The syntax of the labeled value-pair expressions lvp-exp will be defined more precisely in the next section. The message send list and the thread action list simply consist of a list of corresponding actions.

message-send-list ::= **[** *message-send*$_1$ *, ... , message-send*$_n$ **]**
thread-action-list ::= **[** *thread-action*$_1$ *, ... , thread-action*$_n$ **]**

In all such list expressions, the value $n \geq 0$; when it is equal to 0, it indicates an empty list so that there could be 0 or more message send actions performed as well as 0 or more thread-actions performed along any transition. We now consider the format of message sends.

message ::= *m(lvp-exp)*
message-send ::= *thr-exp* ***!*** *message*

Referring to the above grammar, a message consists of a message name and associated data. The data sent with the message is specified using a labeled value-pair expression lvp-exp that dynamically evaluates to a list of name-value pairs. A message send consists of specifying the thread to which the message is sent; each thread has a unique identifier that is dynamically created when it is started. If T is a thread identifier, then T *! message* denotes a sending of message to thread T; in this case, the message will be added to the message queue maintained with thread T. More generally, the thread T is specified using a thread-expression "thr-exp" that evaluates to a thread identifier.

Thread actions can be the initialization (forking) of new threads which can optionally be specified to be a child thread, and a rendezvous action to join with all child threads; they are given by the following grammar:

thread-action ::= init *(st-exp , message-list, lvp-exp)*
| init-child *(st-exp , message-list, lvp-exp)*
| rndz

The init action creates a new thread. As stated before, a thread's state configuration consists of the control state it resides in, its message queue, and its data state which is a labeled value-pair list. The parameters of init correspond to the initial values of these three

components for the new thread. The state expression st-exp evaluates to a control state that will be the initial state where the new thread will begin executing, message-list specifies the list of messages to be put in the message queue; and the evaluation of lvp-exp gives the initial data state of the new thread. The init-child action is exactly like the init action except that the new thread is associated as a child of the current thread. Finally, the action rndz specifies that the current thread will be suspended in the target state of the transition until all its child threads also reach that state.

Data values and expressions.

We have seen a number of data values being used or as parameters to actions. These are control states (used in initializing threads), thread identifiers (used in message sends), and name-value pair lists (used in message sends and in updating the current thread's data state). Rather than requiring that these be statically specified as constants in the corresponding actions, we allow them to be specified as expressions that are dynamically evaluated to produce the necessary value. In this section, we detail the precise syntax of these expressions.

The main datatype that is used to drive the dataflow in the execution is a name-value pair list. Let L be a set of name labels. A name-value pair list consists of a set of field labels with associated values; we use $\{ l_1 = v_1 , \ldots, l_n = v_n \}$ to syntactically denote the name-value pair list with field labels $l_1 , \ldots, l_n$ (that are in L) with associated values $v_1 , \ldots, v_n$ respectively. The number of entries n can be 0 in which case it represents the empty list { }. The values v_i (for $1 \le i \le n$) can be of four kinds:

1. A control state $q \in Q$,
2. A thread identifier,
3. A field label $l \in L$, and
4. A name-value pair list.

Thus, e.g., $\{ l_1 = l_2 , l_2 = \{l = \{\}\} \}$ is a name-value pair list in which the field l_1 has the value l_2 (which is a field label), and the field l_2 has a value which is another name-value pair list. Allowing field labels to be values enables, for example, a field in a message received to name another field which should be looked up for a certain value.

We define some basic operations on the various kinds of values. For a thread T, T . val denotes the name-value pair list that is the current data state of thread T, T . par evaluates to the identifier for the thread that is the parent of T, and T . chld evaluates to the list of child threads of T. For a name-value pair list $r = \{ l_1 = v_1 , \ldots , l_n = v_n \}$, the field extraction operation r . l returns the value associated with the label l in r; r \ l denotes the result of deleting label l from r; r • l = v denotes the field update/add operation and returns the name value pair list consisting of updating the label l in r to be associated with the value v, if l is already present in r, and otherwise adding the label l with value v to r. The update/add operation is extended to two name-value pair lists; if r_1 and r_2 are two such lists, then r_1 • r_2 denotes the name-value pair list which includes the label-value associations in r_1 for all labels that are not present in r_2 together with all the label-value associations of r_2.

The syntax of data expressions is given by the following grammar:

exp ::= *st-exp* | *thr-exp* | *l-exp* | *lvp-exp*

An expression can be one of four possibilities corresponding to the four types of values. A st-exp is one that evaluates to a control state, thr-exp evaluates to a thread identifier, l-exp evaluates to a field label, and lvp-exp evaluates to a name-value pair list. The syntax for these four types of expressions is given by the following grammar:

st-exp ::= q | *lvp-exp* . *l-exp*
l-exp ::= l | *lvp-exp* . *l-exp*
thr-exp ::= *x* | this | *thr-exp* . par | *thr-exp* . chld | *lvp-exp* . *l-exp*
lvp-exp ::= *y* | { $l\text{-}exp_1 = exp_1 , \ldots , l\text{-}exp_n = exp_n$ } | *lvp-exp* \ *l-exp* | *lvp-exp* . *l-exp* | *lvp-exp* • *l-exp* = *exp* | $lvp\text{-}exp_1$ • $lvp\text{-}exp_2$ | *thr-exp* . val

Most of the above grammar can be understood on the basis of the basic values of each kind (e.g., control states q for st-exp) and the previously defined basic operations on the values (such as field extraction, field update, parent thread etc.). The additional syntactic constructs not previously described are the following. Both thread expressions thr-exp and name-value pair lists lvp-exp can be variables (the *x, y*) – these correspond to the variables referred to in the message match in the guard *x* ? *m(y)* – they evaluate to the values bound from the message match. Additionally, the syntactic construct "this" (which is a thread expression) evaluates to the identifier of the

current thread. Note that with the grammar as we have defined, all four kinds of values are first-class values and can be evaluated, e.g., through arbitrary levels of nesting of field extractions of name-value pair lists.

Finally, we define the syntax of val-cond, the conditional predicate on data values specified in the guard.

val-cond ::= true | $exp_1 = exp_2$ | $val\text{-}cond_1 \lor val\text{-}cond_2$ | $\neg$ *val-cond*

The conditional predicate "true" always evaluates to true (it is useful when one wants to specify no data-value condition in the guard); $exp_1 = exp_2$ is satisfied if and only if both exp_1 and exp_2 evaluate to the same value; other conditional predicates are built using disjunction and negation. Note that the syntax for expressions (exp_1 and exp_2) in the equality test are given by the grammar for exp—the conditional predicate can therefore refer to the data on the received message (through the variable bound to it), the data state of the current thread (through this and this . val) as well as the data states of any parent or child threads or those associated with any of the fields on any such accessible data value. We also remark that other boolean predicate combinations (such as conjunction, inequality) can be expressed using negation and disjunction and are therefore not included as primitive constructs in the syntax of val-cond.

As an example of this syntax and the dynamic evaluation possibilities in the model, we can write the transition

< *x* ? resp(*y*) , *x* = this . val . friend > → < [*x* ! ack (this . val . pckno)] , [ ] , this . val • friend = *y* . next >

which is enabled if the message named resp is received from the thread which is currently associated with the name "friend" in the thread's data state. If this is the case, the message sender is sent a message ack with the data value currently in the packet number field "pckno" of the thread. No thread actions are performed, and the current thread's data state is updated to change the field "friend" to be the value specified in the "next" field of the data received with the message resp. In this way, the original sender of the message can also specify to the recipient the thread from which it should next expect to receive the message resp.

Timeouts and exceptions.

In the previous two sections, we have shown how asynchronous messaging and concurrency are expressed in our model. We now describe how synchronous programming features are incorporated into the model—the synchronous features we include are timeouts and pre-emption through exceptions.

Timeouts are incorporated into the model by following the ideas from the formalism of timed automata. Timed automata, introduced in [4], provide a simple yet powerful mechanism for formally specifying the behavior of systems that are governed by timing constraints; we refer the reader to [5] for examples of interesting real-time behavior requirements that can be thus described. We augment our model with a finite set of real-valued clock variables z, z_1, z_2, … belonging to a set Z. Each thread's data state, in addition to its name-value pair list, now includes the current values for its clock variables. Any of the clock variables can be reset to zero simultaneously with any transition. At any instant, the reading of a clock variable equals the time elapsed since the last time it was reset. Guards on transitions can now include a clock constraint that must be satisfied by the current values of the clock variables for the transition to be taken. Actions on transitions can specify a set of clock variables that are reset to zero. The clock constraints used in guards, "clock-cond," are defined by the following grammar:

$$\textit{clock-cond} ::= z \leq c \mid z \geq c \mid z < c \mid z > c \mid \textit{clock-cond}_1 \wedge \textit{clock-cond}_2$$

where z is a clock variable, and c is a constant that is a non-negative real number. Thus, the clock constraint $z \leq 3.2$ in the guard of a transition would require the transition to be taken before 3.2 seconds have elapsed after the clock variable z was reset to 0 while the clock constraint $z > 3.2$ would express the requirement that a timeout of 3.2 seconds happen before the transition is taken. Clock-constraints appear as another component in guard; the action clause includes a set of clock variables that are reset to 0 with the transition.

Exceptions are included in a syntactically similar fashion to message sends. To this end, we add another syntactic form to message sends using *!!* to denote the throwing of an exception.

message-send ::= *thr-exp* ***!*** *message* | *thr-exp* ***!!*** *message*

where the syntax of the non-terminal message is the same as before. Exception handling is syntactically identical to message receipts, and therefore requires no extension to the syntax. While having syntactic similarities to message sends and receipts, their semantics is quite different. First, exceptions are not queued, but are immediately notified to the recipient thread. Second, when an exception is received by a thread that is in a state q, if there are no transitions out of q with a match for the exception that is received, then the thread is killed and the exception propagated to its parent. On the other hand, if the message in the front of the queue is not matched with any of the transitions out of a state then the message is simply dropped. Thus, exceptions act synchronously and provide a mechanism for pre-emption.

5.2 Operations Expressible in the Model

In defining the formal model, we have included only the minimal set of constructs necessary for achieving the required expressiveness. In this section, we briefly outline how other operations, although not explicitly included in the model, can nevertheless be expressed. Many of these operations appear in the call definition language of PCA.

5.2.1 Data Types

The only data values in our model are lists of name-value pairs. We do not have, for example, natural numbers or strings or functions and operations on them. However, we can show that the name-value pairs list of the form included in the model are sufficient to encode other data types and operations on them. To this end, we show how natural numbers and all computable functions on them can be expressed in the model. Since other data types such as strings can be encoded as natural numbers, it follows that most data types of interest can be translated into our model.

We encode natural numbers as follows. The number 0 is encoded as the empty list {}, and the number n is encoded as the list {l = {l = {…{l = {}}…}}} with n-levels of nesting. The successor and predecessor

functions can then be encoded as follows. If the variable x is bound to the encoding of a natural number n, then the expression $\{l = x\}$ evaluates to the encoding of n+1, and the expression x . l evaluates to the encoding of n-1. The test for equality with zero can be defined as the predicate $x = \{\}$. Using transitions with ε message-match guards and loops on states, we can encode inductive and recursive loops. It then follows by standard theorems of recursion theory [54] that all partial-recursive (i.e., computable) functions on natural numbers can be represented.

5.2.2 Thread Types

The call definition files in PCA can ascribe types to threads, and guards on transitions can specify the types of threads to which the transition applies. This is useful because different protocols can share a similar processing flow; and the call graph, instead of replicating states for each possible protocol, can share states across different protocols. A thread can then be given a type corresponding to the protocol it is meant to be processing, and differences in the transitions from states can then be captured by specifying guard constraints on the type of the thread. Such thread types can be expressed in our model by including a special field label thrd-type and having its associated value be set to the type of the thread. When a thread is initialized, the field thrd-type can be set to the appropriate type and guards on transitions can specify predicates on the thrd-type value to specify transitions based on the thread type.

5.2.3 Dynamic Extensibility

Because the control state parameter in thread initialization is not statically defined but rather specified through an expression that is evaluated dynamically, our model naturally includes the ability to extend the labeled transition graph with new nodes and transitions and have new threads instantiated on the extended graph. For example, suppose we define a new graph that includes a new state q that any currently executing threads are unaware of. The new state can be sent as the data value of a message which can then be used by an existing thread to instantiate a new thread starting in the new state

and which would traverse the newly defined graph extension. As a concrete example, when a new thread needs to be instantiated starting at a newly defined state, one could send a message named “extend” within whose data parameter, the field named “start-state” has the value of the newly defined state. Then, the transition < x ? extend (y) , true > → init (y . start-state , [ ] , {}) would create a new thread starting in that state (with an empty message queue and empty data state).

5.2.4 Graph Calls and Returns

PCA includes a mechanism analogous to function calls and returns. The state transitions in the call definition file can specify a special action to “call” starting in some specified state of the call graph. A corresponding “return” action then returns control to the state where the call was originated. This is an important feature of PCA because it is one mechanism for achieving dynamic extensibility of already-executing call flows. Such calls and returns can be encoded in our model as follows. We include a special field name call stack in each thread’s data state. Stacks and associated push and pop operations can be encoded using nesting of name-value pairs, similar to the encoding of natural numbers previously described. Then a graph call is simulated by forking a new thread with the value on the call stack field updated to push the current thread identifier, accessed through the “this” expression. A graph return is then simulated by popping the stack in the value associated with the call-stack field to send a return message to the thread on top of the stack.

5.3 Conclusion

We have presented a precise and complete definition of a formal model that we believe is extremely well-suited for describing and programming network services. The model provides the theoretical underpinnings for the way call flows are specified in the Portable Call Agent tool. The PCA tool has proven itself over many years to provide a reliable, flexible, high-performing, and portable solution for

the implementation and emulation of many services, including multiple protocols and calling features in VoIP call control.

From a model-theoretical standpoint, the following are our technical contributions. To our knowledge, this is the first formulation of a graph-theoretical or automata-theoretical model in which high-level programming constructs are ascribed to transitions. These richer transitions enable one to express:

1. The concurrent execution of multiple coordinating threads traversing the same automaton,
2. More sophisticated data state maintenance,
3. Dynamic evaluation of the contents of message,
4. Dynamic evaluations of the targets of message sends leading to dynamically established communication patterns among coordinating threads, and
5. Dynamic extensibility.

At the same time, it retains the automata-theoretic appeal of an explicitly identified control flow in terms of states and state transitions. The expressiveness of the model distinguishes it from other automata-based formalisms with the most prominent examples being extended finite state machines (EFSM) and Statecharts [31]. Extended finite state machines include variables with variable values being used as guard conditions on transitions and transformations on variable values associated with transitions. EFSMs have a single thread of execution and transitions do not include concurrency primitives for forking/joining of threads. Statecharts allow hierarchical specifications of nested automata—a feature absent from our model. However, the concurrency in Statecharts is limited to what can be statically specified in the transition structure; in particular, it cannot express dynamically evaluated targets of message sends or forking of new threads based on the contents of message receipts. On the other hand, formalisms such as Promela used in the model-checking tool SPIN [35], are C-like programs without an explicit identification of control states and control flow.

Our model smoothly unifies a number of computational paradigms that to this point have remained separate. First, it can express both the asynchronous characteristics of interactive systems as well as the synchronous characteristics of reactive systems. Second,

while it allows rich data operations, the dataflow is explicitly delineated along message sends and receipts, thus assimilating the spirit of dataflow languages. The combination of asynchronous and synchronous features, as well as an explicit control and data flow seems to correspond well with the way network services are typically specified.

6 The LiFE execution engine architecture

6.1 Overall System Architecture

LiFE is composed of a set of cooperating resources. Depending upon how the flow graph is defined a particular solution may be composed of one or more instances of the LiFE processes along with one or more additional support processes called for historical reasons, Work Flow Agents. In addition, control and administration of LiFE instances is accomplished using either the LiFE tracker for low level information, or the higher level element management processes. Figure 6.1 depicts a simple LiFE configuration.

The LiFE executable is designed to read in a graph definition at start up time. The flow graph is described in a text file. A flow graph may conditionally include other named flow graph definition files. The actions described in the flow graph may result in programs sent to a Work Flow Agent. As shown above, the flow graph description may reference additional text based files that contain definitions of Filters which are used to transform messages into name/value pairs and Templates which describe the transformation of name/value pair sets into messages. Objects referred to "contexts" traverse the nodes of the graph.

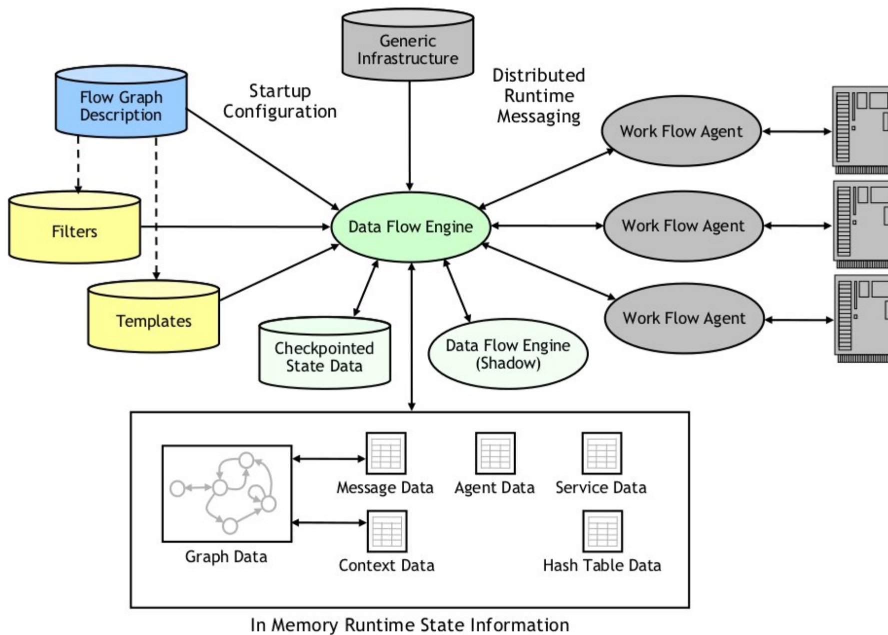


Figure 6.1: Software structure of LiFE

Each LiFE instance can cooperate with another flow engine instance, this is represented by the "Shadow" designation above. In this case, a duplicate context is created on the "shadow" engine instance, and the state of that context is updated when the context enters a place in the graph definition that contains the "DO_STATE_CHECKPOINT" directive. Contexts created and updated in this manner on the shadow engine instance, do not traverse the graph, but are moved from place to place within the graph as directed by the main engine instance. These contexts are considered "inactive" by the shadow engine instance. If the activation criteria is met for a set of inactive contexts, the shadow engine will then promote the contexts to an active status and the contexts will begin to traverse the context in the same manner as a "native" context. These activated contexts will continue to be labeled as "foreign" even when activated. If an instance of LiFE re-establishes communication with the shadow engine instance, and advertises itself using the same identifier as was used when the once inactive contexts were created, the shadow engine may offer to migrate the contexts back to the original engine instance. As

alternative, the shadow engine may "adopt" the contexts and relabel them as "native".

6.2 LiFE and failover

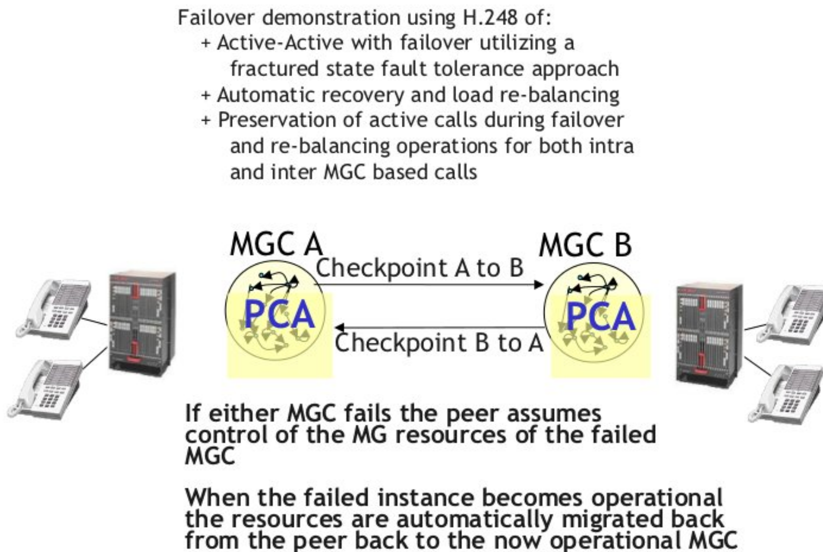


Figure 6.2: Support for Failover

As depicted in the diagram of Figure 6.2, the flow engine supports saving state between instances of the flow engine. The example above outlines an "Active-Active" failover configuration. In "Active-Active", two instances are involved. Each instance is supporting its own distinct set of active elements, in our case they are voice over IP endpoint. The "failover" part is simply that if one of the instances fails, the responsibility for managing the endpoints shifts from the primary engine to the backup (sometimes called secondary) controller. So the configuration that is shown in Figure 6.2 permits both instances to do useful work, but at the same time backing up the other instance. It should be noted that, using the flow engine's state migration facilities, it is straightforward to create an Active-Standby configuration. Using this approach, the state associated with some or

all of the endpoints are transferred to a different instance of the flow engine. This can be done either manually, using a defined externally triggered procedure, or automatically using the checkpoint method. In either case, once the endpoints have been transferred, the old primary flow engine can be shutdown. As the name suggests this is a configuration in which a second flow engine will only provide management of an endpoint if the primary endpoint manager (flow engine) suffers a failure.

At the root of this capability is the ability of the flow engine to transmit - a context's name/value pair space, the global registrations associated with the context and where in the graph the context is currently located, to another instance of a flow engine. But is this enough to actually make the failover work? In practice, the answer is "it depends". For some systems, the information that contained in the name/value pair describes all that is needed to correctly operate on the session. The session referred to in this case is not specific to telecommunications (such as a SIP session), but rather it is the unique set of values that comprise the state of the currently modeled computation. As an example, a credit card reconciliation application may have sessions that exist in the system for long periods of time as the data needed to complete the reconciliation process trickles into the system. The information captured as part of the session state is dependent upon how the computation is defined by the flow graph programmer.

6.3 LiFE and fractured state

For some systems in which the state of the endpoint can change quickly relative to the time it takes to migrate state information, such as the H.248 system that was depicted above, upon examination, it might be found that the migrated state (the data that has been migrated to the backup controller) is now out of date. It could be that the end user took an action just when the failure occurred and it was missed.

In order to create a system that appears seamless to the end user, care must be taken to insure that there is a mechanism to create or discover the current state. In the case described above one can take

advantage of the end point itself to provide the information necessary to update the migrated state and make it current. A successful approach has been to take advantage of the difference between fast and slow changing state. Most session protocols explicitly name the session using a name that is unique among the sessions (or at least unique for the expected lifetime of some number of session lifetimes). This information will therefore change slowly (if at all) during the lifetime of the communication session. Looking at the last 3 popular VoIP protocols: MGCP, SIP and H.248, all of them feature this idea of a slowly changing identifier (eg. Connection Id, Session Id) that can be used to launch additional queries to refine the endpoint's idea of its current state. The responses can be used to update the context's name/value pairs to reflect the current state of endpoint. In this case, the endpoints are being mined to get the data needed to drive the migrated state into sync with the endpoint. The true state of the system is split between the manager (controller, flow engine) and the endpoint.

At startup, the LiFE instance is provided with the name of the flow graph definition. The engine parses the definition and converts the graph into an internal data structure. The data structure defines the possible nodes that can be traversed by a context. Part of the graph definition is a set of arcs that are guarded by named events. These events are routed to individual contexts as a result of messages or timer events. A context's state is described by its current place in the graph, its type, and the contents of its name/value pair list. As messages are received, they are routed to an individual context and using the current context's place are transformed into name/value pairs that are assigned to the context's name/value pair list. Each place definition of the graph may indicate a unique transformation of the message, so that the transformation is based upon the current place of the context. In addition, LiFE can synthesis the name of the filter based upon the current context's name/value pair list, so that there is a high degree of flexibility available in managing the transformation of messages to name/value pairs. In a similar way, the transformation of a context's name/value pair elements into a message is quite flexible. Each place may have a unique template file, which describes the output message with references to the context's

name/value pair list. In many cases a place will have a specific template file associated with an event, that is, if a specific event is delivered to a context at a place, that event is used to choose the action to be taken. If that action is to generate an outgoing message, there will be a parameter at that action definition which contains the template file name. In the same manner as the filter, this template file name may itself refer to the context's name/value pairs, so that the template file selected may be different depending upon the results of a previous message or the path the context has taken through the graph.

Depending upon the contents of the graph file, sophisticated behaviors can be described. As an example, Figure 6.3 shows an application of LiFE as a softswitch.

In this configuration, the VoIP gateway ends messages to LiFE executing a graph that describes media gateway controller (MGC). In the case depicted a MEGAGO “Notify” message has been sent to the LiFE instance, which I will refer to for this purposes of this example as the MGC. The “Notify” message is passed through the filter: NotifyRequest.h248.filter. This file defines which data elements of the message are converted into name/value pairs of the owning Context. Close inspection of the example indicates that the value of

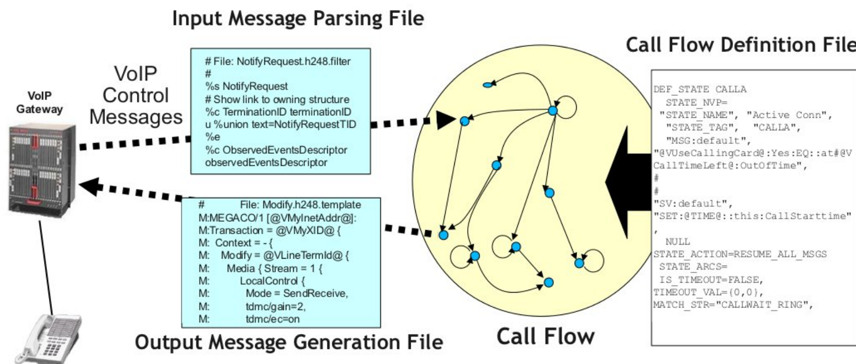


Figure 6.3: LiFE as a soft switch

of the union TerminationID member, NotifyRequestTID will be used

as the value placed on the Context's name/value pair, paired with name “terminationID”. In the example above the Output Message Template File “Modify.h248.template”, is used to transform the Context's name/value pair file into a message sent to the VoIP gateway. The template file contains the structure of the message, with references to the names of the name/value pairs contained on the Context's name/value pair list. The example shows a common use for substitution is the string @VLineTermId@. In this case the value of the name/value pair that has “LineTermId” is substituted into the message at the octet offset where the @VLineTermId@ is located in the template file. Much more sophisticated substitutions can be performed, including the capability for output of conditional blocks and conditional lines. The name/value pairs of the Context are used, at the time of evaluation, as the criteria for inclusion or exclusion of blocks of text in the message. The example diagram also depicts a representative of the call graph that controls the movement of Contexts through the graph.

6.4 Just-in-Time Element Management

Many of the features that are part of LiFE were a direct result of the difficulties encountered creating a complex system given a strict deadline. One of the issues that arose repeatedly was the development time lag associated with the element manager system (EMS). An EMS is used by an operator to turn up a system after power on, gracefully bring the system down, perform upgrades, change its configuration, migrate users from a live system, and potentially take parts of the system down for maintenance. The EMS, being the control panel for the system, should have ways to manipulate all of the features of the system. As a consequence, whenever features are added, or changed, invariably the EMS would undergo a revision. In addition, if the EMS is not finished, the system can not be deployed. The EMS could be and often was the last part of the system to be finalized and tested.

During acceptance testing of the VoIP Line Access Gateway, LiFE was used as the controlling soft-switch, either as a Back to Back User Agent (B2BUA), or as a Media Gateway Controller. In these

situations, LiFE also had an EMS to make certain operations, such as selectively bringing down line cards, or monitoring the health of the system easier for the customer team members. The approach taken to create the EMS for LiFE was different than the usual practice. The elements managed by the EMS were defined in the flow graph itself. This includes resource monitoring (events such as low-water marks, excessive delays, resource exhaustion), and operational procedures such as taking cards out of service, selectively migrating user endpoint between active systems, creating test conditions, and selectively turning on services.

The effect was to have a "blank slate" EMS which when it attached to the specific flow engine instance, would create widgets that corresponded to those defined in the flow graph. Some of the widgets corresponded to element alarms, while others would appear as labeled buttons that could be used to initiate management operations.

```
 "THRESHEVENT:LostEndPoints:LVL:2",
        "LostResGW:SENSE:pos:THR:Low=5,Medium=10,High=19:H
YSTER:2,Top:INIT:0"
```

Figure 6.4: Example of a THRESHEVENT

A "THRESHEVENT" flow graph construct was used to define an element that would serve as an alarm. This is of course nothing more than the value of a name/value pair. Any context in the system has access to the variables that correspond to the THRESHEVENT name/value pair list. These variables can be read and written. The declaration of the THRESHEVENT defines the name used, if an event is sent when the variable exceeds, drops below, or equals a target count. The declaration also includes the initialization value, and three threshold values, a Low, Medium and High. Since some counts may jitter around one of the threshold levels, creating a string of alarm events, a hysteresis value may also be defined. Clearly not all alarm events have the same gravity, so the declaration includes a severity index. This, like all of the other elements of the threshold event declaration have no intrinsic meaning. It is up to the graph designer to assign a meaning to each of the elements. An example of

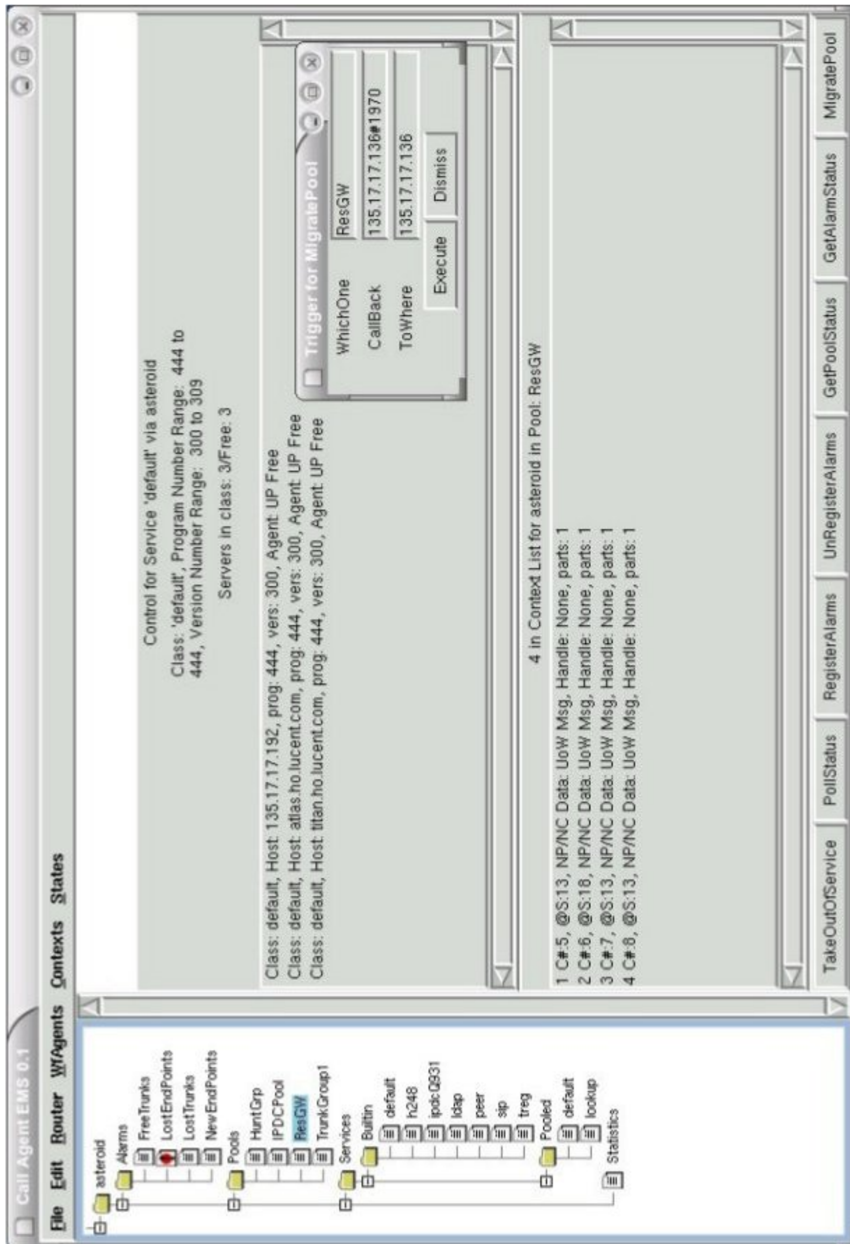


Figure 6.5: Example of the LiFE Element Manager

a threshold event is shown below in Figure 6.4.

The form of the declaration is the following: THRESHEVENT key

word, followed with the name of the threshold event, in this case "LostEndPoints". This threshold event is declared as a level 2. The value portion of the declaration starts with the internal name that can be accessed by the contexts. Since the "SENSE" is positive, the alarm is activated when the count becomes more positive than the various threshold levels: low at 5, medium at 10 and high at 19. This definition includes a hysteresis declaration of 2, with an alignment of "Top", so that the count must exceed the threshold by 2 in order to trigger the alarm, or drop below the threshold in order to turn off the alarm. In the case of "Low" as in the example, an alarm would be triggered if the LostResGW value was 5 + 2 or 7. The alarm would not be turned off until the value of LostResGW went below 5. If the hysteresis alignment were to be centered, then the "on" threshold would have been 1 more than the defined level to turn on the alarm and 1 less than the defined level to turn it off.

"TRIGGER" procedures are used to create entry points used by the EMS to perform operational actions on the flow engine. The declaration line contains the externally visible procedure name, a list of data elements required by the flow engine to fire the trigger procedure, the entry point into the flow graph for a context (the graph place), and the color (type) of the context. As indicated, the declaration of the trigger procedure contains a set of variables that must be supplied by the EMS in order to allow the procedure to be executed. The types and ranges of these necessary variables are contained in the graph as part of the trigger definition. Each variable is described in a "TRIGGERVAR" definition. Each line of this definition contains a reference to the trigger procedure, the name of the particular variable, and the type of variable. Variable numbers, ranges of numbers, fixed field length strings, free form strings and enumerated lists are supported.

When the EMS attaches to the flow engine, it will get a list of all of the manageable elements, both threshold variables, and trigger procedures. The current EMS will then request additional information about each of the trigger procedures when the user attempts to activate that procedure. The additional information about the trigger variables could be requested by the EMS at any time. During this additional information request, the triggervar

information is sent to the EMS, so that it can prepare any data entry widgets needed to trigger a particular operational procedure on the flow engine. The current design of LiFE puts the requirement on the EMS task to make the specific TRIGGERVAR followup queries needed to characterize all of the variables needed for a trigger procedure. When the user activates a specific trigger procedure, the EMS will prompt the user to input each of the necessary input variables. To the extent that each variable has been assigned a type, the EMS performs range or type checking. With all variables input by the user, the EMS then issues the trigger procedure command along with each of the required variables. Figure 6.6 depicts an example of a trigger procedure with the required trigger variable definitions.

```
#
#   Trigger procedure that forces a MG to undergo MigratePool to
another MG
#

"TRIGGER:MigratePool:LVL:1:VAR:TargetMG,MGPort,NewMGC,NewMG
CPort","StartState=EMSForceHandoff(0),
        ContextColor=h248,Oper=migrate",
      "TRIGGERVAR:MigratePool:TargetMG", "I:IpAddress",
      "TRIGGERVAR:MigratePool:MGPort", "I:Number",
      "TRIGGERVAR:MigratePool:NewMGC", "I:IpAddress",
      "TRIGGERVAR:MigratePool:NewMGCPort", "R:2944-3600"
```

Figure 6.6: An Example of a Trigger Procedure

This example shows a trigger procedure that can be used to command a specific media gateway to re-home to an alternate media gateway controller. This example came from a flow graph used to create a H.248 media gateway controller (MGC). This procedure could be used to selectively move elements from one MGC to another. The declaration begins with the reserved word “TRIGGER”, followed by the name of the trigger procedure, in case “MigratePool”. The LVL field indicates that the level of this procedure is “1”, that is the EMS will only be permitted to see this procedure if it has level 1 privileges.

The required variable list follows, in this example: TargetMG, MGPort, NewMGC and NewMGCPort are required. The value portion of declaration contains the starting state which is

“EMSForceHandoff(0)”. If the trigger procedure is successfully triggered, a context will be placed at this state in the graph. The next portion of the declaration contains the ContextColor, which is set to h248. The last part of the declaration is a name/value pair that should be included in the created context's name/value pair list. Here a pair should be created with the name: “Oper” that has the value: “migrate”.

It is expected that procedure EMSForceHandoff will check for the value of Oper as part of its processing in order to disambiguate which trigger procedure was used to generate the handoff event. The next set of line contains the trigger variable declarations. Looking at the declaration of NewMGCPort, the keyword “TRIGGERVAR” is followed by the tuple: HandOff:NewMGCPort. Using a tuple allows us to position the TRIGGERVAR declarations ahead or behind of the particular trigger procedure. The value portion of the declaration indicates that the acceptable value range for the NewMGCPort is between 2944 and 3600. The current design relies upon the EMS to enforce the range, this should be augmented to include additional checking by the flow engine before accepting a value as valid. The other three trigger variable declarations follow the same format as the NewMGCPort, except that these elements are declared as Internet addresses (I).

As a sanity check the flow engine will only accept a trigger procedure if all of the defined variables are sent by the EMS. With this check passed, the flow engine creates a context of the defined color at the entry point place within the graph, and nodes the variables on the context's name/value pair list. The context is now activated and can traverse the flow graph in the same manner as context's created as part of any other protocol exchange. As a result of this design, a user from the EMS can initiate any number of behaviors on the flow engine, including making calls, sending e-mails or interacting with any of the other systems currently described by the flow graph. Deattaching from one flow engine, and attaching to another may change both the alarms and functions available to the end user. This approach ensures that an EMS connected to a flow engine always results in the latest and correct functions available to the end user. When the definition of the procedure above is read by

the LiFE element manager, a button will appear along the bottom of the graphical application. Figure 6.5 shows an example of the LiFE element manager. The reader will be able to see the "MigratePool" button on the lower left-hand portion of the application.

7 Use Cases

So far, the thesis has presented a system that can be used to describe a general computation. In this chapter, several applications of the flow engine are explored. In some of these cases, not only was the flow engine used in a production environment, but it became a critical component in demonstrators used by the company to close a sale.

The flow engine was used in the following applications:

- FTS-2000 – an OCR driven order entry system used to support the creation of calling cards for the (US) Federal Government;
- Lucent ProCard – internal application used to match faxed or scanned images of receipts with bills as part of a financial reconciliation process;
- AT&T Bill reader / automated comparison of user's (scanned) bills used to make counter offers;
- AnyMedia Line Access Gateway (Load testing and feature demonstration) of the Session Initiation Procotol (SIP) in place of an Application Server. The flow engine was used to demonstrate LAG capabilities to customers at the customer's location;
- AnyMedia Line Access Gateway MGC (H.248) Benchmark Tests . The flow engine was used to emulated several variations of Media Gateway Controllers (MGC) needed to perform load and feature tests, as well as control the LAG during

Benchmark/Acceptance tests at the customers facilities; and

- AnyMedia Line Access Gateway Call Agent (MGCP). The flow engine was used in place of a Call Agent, to perform functional and load testing of the LAG's MGCP capabilities. It was also used to demonstrate the VoIP system to customers.

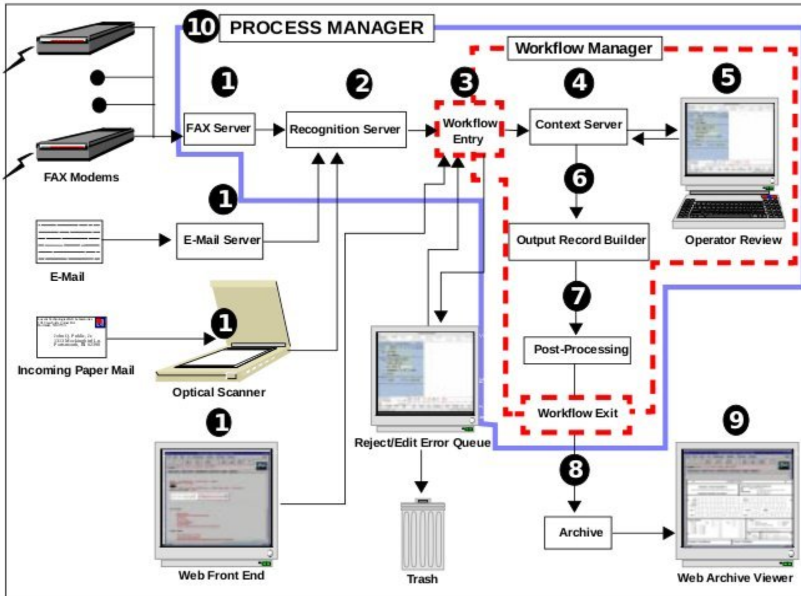
Below, two of these applications are described in detail: (1) The image-processing application that was part of the AT&T bill reader application, the Lucent ProCard application and FTS2000, and (2) the telecom emulation service that was used in the AnyMedia Line Access Gateway.

7.1 Image Processing Support

As part of the first three applications in the overview above, the flow engine supported processing steps that were needed to manage the conversion of image data into information ready to be accepted in a downstream application. The general form of the first three applications follows the structure that is shown in Figure 7.1. These applications used neural-network based optical character recognition (OCR), which converted handwritten numbers and letters into sets of character & probability tuples. The remainder of the system used the context of the character, along with dictionaries to choose and validate entries.

All decisions made by the system carried a confidence parameter along with the selected interpretation. If the conversion confidence fell below a configured lower bound, the system queued the conversion data along with the original image data to a user-review station. The operator had the option to defer the review of some of the forms, and process others instead. Once an operator reviewed and possibly corrected a form, that item was then released back into the system. In many cases, a specific collection of forms had to be processed together. The system was required to hold forms that might have been converted with high confidence, waiting for other forms in the group that needed manual review. Once all the forms of

How FIRST Works



1) **INPUT**: FIRST is capable of receiving completed forms from FAX, e-mail, paper mail, or the Internet. Faxed or scanned forms can consist of multi-page originals or photocopies.

2) **RECOGNITION**: After receiving a form, FIRST processes it through the Recognition server. The recognition server identifies the form and then performs OCR/ICR on it before sending it into the workflow. If the workflow does not recognize the form, it sends it to the Edit Error Queue where users can either recover it or trash it.

3) **BUSINESS PROCESSES / WORKFLOW**: After FIRST performs recognition on a form, the form enters the workflow. While in the workflow, administrators can monitor the status of current work items in the workflow with the WorkFlow Tracker. This enables an administrator to identify possible resource constraints in FIRST and address them quickly.

4) When in the workflow, the UOW goes to the Context server for contextual analysis where the form's data is verified against internal databases. If a context field fails the database lookup, the Context server highlights the flagged field and sends the form to the Operator Review Workstation.

5) Once at the Operator Review Workstation, users view and correct forms containing failed fields. When completed, FIRST sends the form back to the Context server for re-validation.

6) **OUTPUT**: FIRST sends corrected forms from the Context server to the Output Record Builder, where results can be placed into an ODBC compliant database or a predefined ASCII file. This data can then be sent to an upstream system for further processing.

7) FIRST takes the formatted file and transfers it to a downstream system for further processing.

8) FIRST archives the image and the corresponding results data file. The transaction then exits the workflow.

9) Once archived, users can retrieve archived data through FIRST's Web Archive Viewer by entering information into a query screen.

10) **MONITORING**: The web-based Process Manager allows administrators to remotely manage FIRST by enabling them to start, stop and monitor FIRST modules throughout the network. The Process Manager's automatic recovery process can restart FIRST modules should a mission critical application suffer a catastrophic failure.

Figure 7.1: Overall Architecture of FIRST

a group passed the confidence limit, all members were released to the downstream data-entry system. The orchestration of the forms was handled by the flow engine.

These data-entry applications feature a lower offered load, usually in the order of tens of events per minute. However, the managed elements (forms) could end up depending upon a human operator to

complete, leading to long-lived transactions. It was common for elements to reside in the system (engine) for days or weeks at a time. Given operational considerations, a requirement arises that the long-lived transactions may not be lost across reboots caused by maintenance or failures.

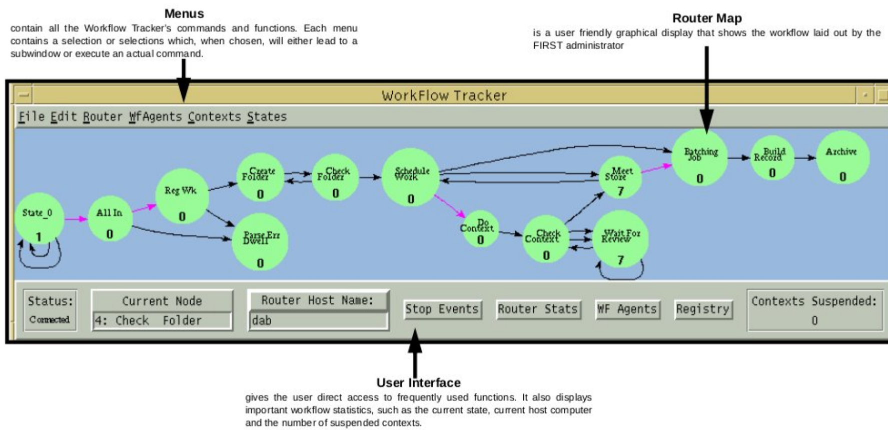


Figure 7.2: Example of Workflow Tracker Use

A typical work flow graph is depicted in Figure 7.2. Using the Tk based "Workflow Tracker", adminstrators of the system could interact with the LiFE engine and manage jobs that were currently in the system. The workflow tracker provides a more detailed level of control for the system adminstrator than the management system described in Chapter 6. But like that system, the graphical tracker interacts with the LiFE engine to create a representation to the user. The graph that is shown in Figure 7.2 is created by the Tk application based upon topology downloaded from the LiFE engine. This allows a user to get a graphical representation that changes based upon the particular managed flow engine. With this application, an operator could manually move jobs to move between states, disable or enable the use of specific external agents, or re-scan for external resources. In addition, the operator could turn on diagnostic levels, or selectively trace specific contexts between states in the graph. Often it was useful to examine a context's name/value pair list to understand a context's current condition. Another feature of the Workflow Tracker is its real-time display capabilities. When this mode is

enabled, the Tracker flashes the connecting arc when a context traverses an arc from one state to another. It then updates an integer count that represents the number of contexts resident at that state. This is a handy way for an administrator to see how the system is performing. If a state that represents the use of an external service shows a rapid increase in the number of contexts, the administrator can use the Workflow tracker to find out the status of all of the external workers supporting that state, and possibly start additional copies on available machines. He or she could then use the Workflow tracker to scan for the new services, which could then be pressed into service. The adminstrator could also tag resources so that the flow engine would no longer use them. In this way an operator could gracefully take support elements out of service without taking the main workflow application down.

For computations that have few defined states, the interface provided by the Workflow Tracker provided a quick way to judge the health of the system.

7.2 Emulation of Telecommunication Services

The second application arose from a specific request. There are a number of ways to provide a voice-communication service. The architecture of modern, large-scale communication systems has been evolving quickly in the past decades. Evolving voice communications to Voice-over-IP (VoIP) has emerged as a dominant activity for service providers. The ability to centralize operations, replace special-purpose hardware with less expensive compute nodes and reduce operational expenses by combining hardware platform types between Web and traditional voice services makes the shift an obvious choice for service providers. In order to take full advantage of the evolution, service providers have steadily moved traditional voice calling capabilities from the special-purpose appliances such as the 5E switch to server blades running a VoIP-control application. After switching out the telephony controller (5E), the service provider must face the question of how to provide the service to end users. A number of service providers deployed media gateways that re-used the “last mile” connections and end equipment. The media gateways,

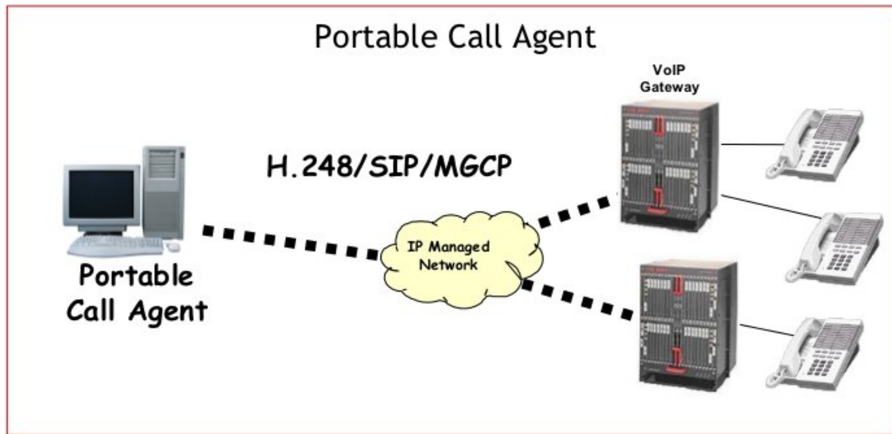


Figure 7.3: LiFE Configured to Test VoIP Line Access Gateways

also referred to as Line Access Gateways (LAGs) were installed in the local concentration point, sometimes called the "central office". The customer-facing interface of the LAG used the existing copper wire (also called loop), while the network-facing interface was connected via a 1Gbps Ethernet link. Thus the LAG would interpret the analog signals on the copper loop and convert them into messages directed towards the network. It is the LAG's role to route call-control messages to a specific network control element, while the information that represents the voice information is routed to a different network element. To accomplish this, the LAG exchanges *control* messages with a network element designed to manage the procedures necessary for the set up and release of calls. The network control element goes by different names depending upon which VoIP protocol is being used, but a general term for this element is a *softswitch*. One may hear terms like: Call Agent (MGCP), Media Gateway Controller (MGC – H.248/Megaco) or Application Server, Back-to-Back User Agent (B2BUA – SIP). These are all names for the same thing: a network element that manages VoIP calls. Central to this architecture is a separation between the set-up information and the voice (conversation) data. Figure 7.3 depicts LiFE configured as a "Call Agent". In order to make a call on a VoIP network, two distinct applications are needed: a media gateway to handle the data that represents the conversation; and a softswitch to set up the call. The

LAG functions as the media gateway, but needs a softswitch to make calls.

Service providers were primarily interested in moving their customers from a traditional telephony solution to VoIP, but preserving the same user experience. They saw the LAG as a way to evolve their networks. Of course they also had several distinctly different ideas on how to accomplish that task. Helping them along was a set of *standardized* VoIP protocols that had very different ways of setting up and managing calls. In broad terms, some protocols were designed around intelligent endpoints. These protocols were envisioned to put the burden of call management at the endpoints, with a very limited control role for a network element. These protocols are referred to as *functional* protocols. The best known functional VoIP protocol is the Session Initiation Protocol (SIP). The reader will notice that I used the word "envisioned" in the previous description of functional protocols. It turns out that during the roll out of VoIP services, it was found that it was difficult to roll out endpoints that were able to implement all of the features needed to equal the traditional telephony user experience. Testing various vendors endpoints for compatibility with the standard was, and still is, a daunting task. Early on, many times the endpoint was relied on only to manage simple call models, such as originating and termination of a call, using elements in the network to set up more complex features such as call parking, attended-call transfer and call holding.

The other major category of VoIP protocols are referred to as *stimulus/response*. As the name suggests, endpoints using this type of protocol report user actions to a network element and may be subsequently commanded to perform an action. Media Gateway Control Protocol (MGCP) and H.248 are prominent examples of stimulus/response protocols. A major attraction of s/r protocols is that behaviors are a result of a small number of centralized network elements. Another more practical advantage is that the complex services are much easier to implement, even if the services are implemented in a unique manner.

The reason for using LiFE

A VoIP solution does not always contain only one company's product suite. It was common for the proposal requests to include only the media gateway portion, rather than the combination of a matched set of media and control elements. The theory was that a standard set of protocols would permit a service provider to mix and match solutions, taking the best-of-breed media gateway and pairing that with the best network-control element. Customers considering the VoIP system as a collection of inter-operable standards-based components increased the difficulty of LAG verification and test for several reasons, among them:

1. Many times the business unit could not get access to a competitor's softswitch forming part of a customer's requirements;
2. Often the product schedule required that testing of LAG needed to be performed before a softswitch's feature was available. Interestingly, many times demonstrations to potential customers had to take place before the feature was built by the development team;
3. Differences in feature priorities would drive development schedules such that a required feature on the LAG was not supported by the company's control element;
4. The test organization needed to test the full range of responses from a controlling element. These messages would not be sent under normal circumstances;
5. In several cases, the LAG product organization had to deploy a fully functional system on the customer premises, with limited or no outside connectivity. Therefore, the unit under test was required to be fully self-contained.

Using LiFE allowed the LAG testing organization to have a system that was fast enough to keep up with a fully loaded configuration, that could support all of the protocols and variants, that was easily tailorable, and that was stable enough to run week-long tests without error. Figure 7.4 shows the testing configuration used to test the Line Access Gateway product. In this figure, multiple call control elements are shown. In one of the locations the testing organization was able to get access to two Application Server products (for certain SIP testing).

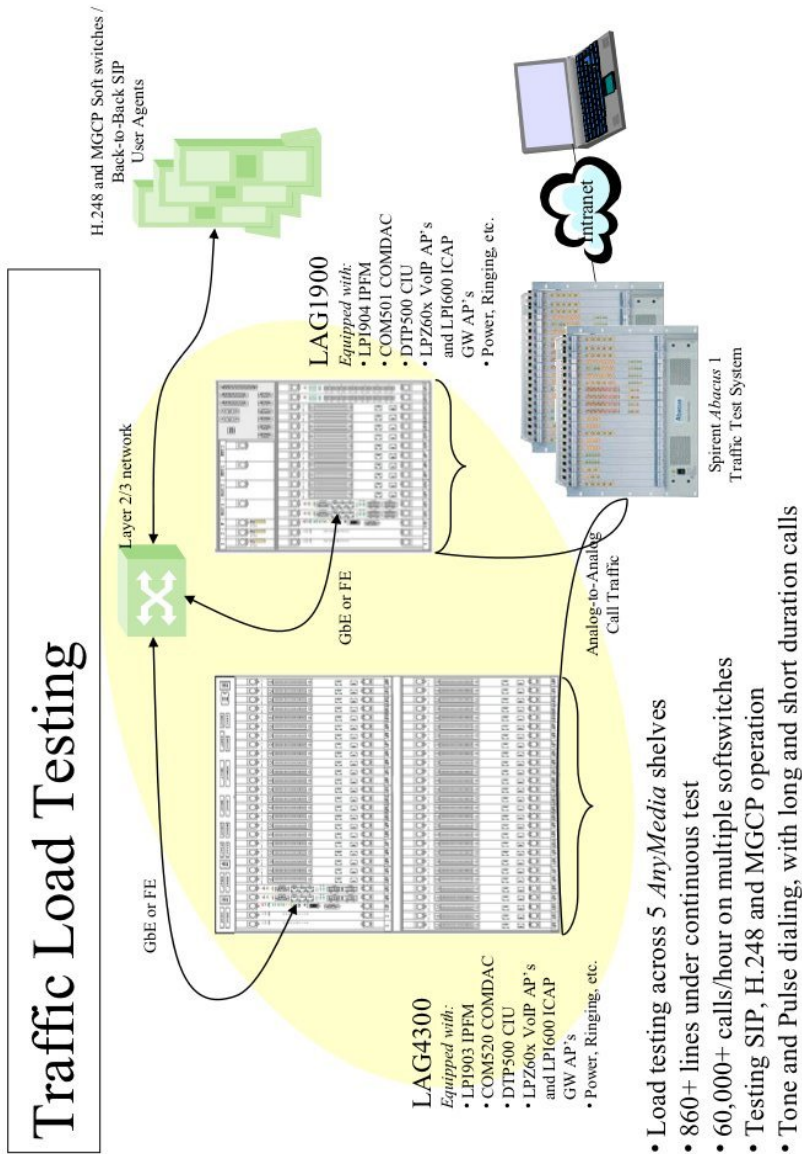


Figure 7.4: LiFE as a Call Control Element

These products, however were not able to support all of the SIP specifications required; for those customers the flow engine was used. Table 7.1 shows how the flow engine was used as part of the test setup. Testing took place in several locations; access to the SIP

IP Subsystem R3.2 CKC Testing							Locations SSW CA = Call Agent		
Country	Customer Variant	LPZ600	LPI600	LPZ602	VoIP Protocol	Customer Softswitch	Locations SSW CA = Call Agent	NBG	SH
J-1	v1	WH			SIP	N-1	CA	CA	CA
J-1	v2			WH	SIP	N-1	CA	CA	CA
J-1	v3		WH, SH	NBG	MGCP	O-1	CA	CA	CA
K-1	v1		NBG		H.248	S-1	CA	CA	CA
M-1	v1				H.248	S-2	CA	CA	CA
T-1	v1		WH		SIP	N-1	N-1	CA	CA
K-1	v1				SIP	N-2	N-2	CA	CA
P-1	v1				SIP	N-2	N-2	CA	CA
U-1	v1				SIP	N-2	N-2	CA	CA
M-1	v2			WH	SIP	N-3	N-3	CA	CA
U-2	v1				SIP	N-3	N-3	CA	CA
M-2	v2				SIP	N-3	N-3	CA	CA
N-1	v1		WH	WH	SIP	N-3	N-3	CA	CA
N-2	v2				SIP	N-2	N-2	CA	CA
R-1	v1		WH		SIP	N-2	N-2	CA	CA
T-1	v2		WH		H.248	N-4	N-4	CA	CA
M-2	v2		WH		H.248	N-5	CA	CA	CA

Table 7.1: Testing Configurations supported by LiFE

Application Server was limited to North America, so the Call Agent, as LiFE was called, was used to perform all testing.

Each call leg was monitored by an external piece of test equipment. This analog test fixture was presented to the LAG as an analog phone set. The test fixture correlated the actions of each call leg and determined whether the call went through correctly. Testing routinely went on continuously for well over 70 hours at rates of 58 thousand calls per hour, with 100% completion rate common; 58K calls is an appropriate busy hour call rate for the LAG, however, it is only 16 calls/s, a rate that is nowhere near the limit of the engine as the reader will see in Chapter 9.

Variations of a standard protocol provided by potential customers were presented as message charts. This form of specification is particularly easy to code in the flow engine. Exceptions to the normal flow of the protocol are readily introduced as conditional execution of extra states. Introducing the new control path does not impact the existing set of states, which avoids the case of unintentionally introducing new coding errors into a known good portion of the graph. Using a name/value pair that is set when a certain type of behavior is required, allows the graph programmer to construct a distinct output template to handle new variations in output messages. It may be convenient to create shared states to represent a common condition, such as “in an active call”, or “called party is busy”. Shared states such as these may be occupied by contexts

Figure 7.5: LiFE's Diagnostic Interface

represented in dissimilar protocols, without causing errors, since the flow engine will use the context's type to format and route outgoing messages. Since the filters convert incoming messages into specified

Using the current call flow definition
- Class 5 features
    - Call Hold/Waiting
    - Call Forward – Unconditional/Busy/No Answer
    - Call Transfer – With & Without consultation
    - 3 way calling
    - Distinctive Ring
    - Call Id Type I & II supported
    - Direct Inward Dialing
    - Line number aliasing
    - Hunt Groups
    - Call Parking
- Trunk Sharing – between instances of the PCA
    - TGCP (MGCP) FXO analog trunk
    - IPDC/Q.931 ISDN-PRI & SCTP/M2UA SS7
- PCA to PCA call setup (tandem)
    - Endpoint to Endpoint
    - Endpoint to Trunk/Trunk to Endpoint
- Other
    - Interface to non-SIP endpoints to IMS/FS5000
    - Creation of multiple line appearances

Table 7.2: LiFE Used as a Call Agent - Feature Set

name/value pairs, a flow graph programmer has ready access to all of the parameters of any incoming message. If a customer has specified non-standard behavior based upon a value of the incoming message, it is straightforward to test and change either the output message, or behavior of the system based upon the customer's criteria. In the case of the change to the output message, the engine does not even need to be restarted. A programmer can simply edit the template file, changing the message as necessary and use the diagnostic interface shown in Figure 7.5 to instruct the flow engine to update to the new version. The test team primarily used this diagnostic interface to LiFE as a means to manage operation of the system, and diagnose problems with the components of the system

including the LAG .

In contrast to the Image Processing applications discussed in the previous section, the Network Emulation graph was much larger. The flow graph used for much of the testing had over 900 defined states. Unlike a traditional finite state machine model, the use of GRAPH_CALLs in the graph greated reduced the number of states that were needed to be defined.

As part of the testing for the Bell Labs VoIP residential gateway program, a number of common end-user features were implemented in the flow engine. These are listed in Table 7.2 Adding additional support for other stimulus/response protocols as well as adding additional features was quick and only required minimal regression testing of older features. The ability to quickly adapt the flow engine to new requirements, while not compromising its existing capabilities or impacting its speed, made it a very attractive choice for the LAG testing team.

8 Performance Architecture

8.1 Design Principle

One of the main goals of LiFE is a linear scaling of resource utilization, especially CPU burden in response to increased offered load. The most important design principles that were followed to achieve this behavior were these:

1. Use of hash functions for lookup;
2. Eliminate sorting of timer elements;
3. Keep system data structures contiguous; and
4. Minimize the number of *malloc*/*free* calls during context and name/value pair list creation.

The following sections provide the motivation and approach taken in the pursuit of outlined design principles.

8.1.1 Use of Hash Functions for Lookup

This is a straightforward design choice, given the trade off between memory utilization and coding complexity (invested time) on one hand and run time on the other. The choice of the hash function is more interesting. Name/value pairs are used for almost all aspects of LiFE, so the impact of the hash function is significant. The requirement for a hash function is that it is fast to perform, but gives a reasonably even distribution. There are any number of hash

functions that are available. The candidates range from simple character summing to complex functions that attempt to provide an even distribution function for a large number of items. The selection of an appropriate hash function must keep in mind the intended use; if one is willing to trade some memory size for CPU load, then a simple function that yields a less than ideal distribution may be the proper trade off. Having an even distribution function results in faster lookups, because the average length of lookup chains in a hash bin will be shorter. This will keep the search result closer to the desired O(1) performance.

The name/value pair lists created by the flow engine provides a useful way to measure hash functions for use in LiFE. Using an example given in Figure 8.1, provides a way to measure the evaluation time and obtain a measure of the distribution of hash values created by the candidate approach. The following hash functions are examined: 1) a simple horizontal checksum; 2) MurmurHash3 [40]; and 3) lookup3[21].

Table 8.1 contains a comparison of three string hash functions when applied to the name/value pair list partially shown in Figure 8.1. For this comparison, each candidate hash function was executed one million times on the name portion of the full list of 133 entries. In addition to the CPU time needed to execute the hash, I also included a rough measure of how well the hash function distributed the input key over a set o 128 hash buckets. In this comparison, I list the units as "links". A link represents a linked list traversal needed to

	Horizontal Checksum	MurmurHash3	Lookup3
CPU Time	5.6s	6.3s	13.2s
128 buckets	85links	83 links	90 links
4096 buckets	4 links	1 link	0 links

Table 8.1: Performance of Name/Value pair String Hash Functions

21 http://burtleburtle.net/bob/c/lookup3.c

access the data element. So in the case of the Horizontal Checksum, 85 link traversals would be needed to access all of 133 of the members


```
        "AnnouncementFeature", "yes"
        "HoldingToneFeature", "yes"
        "HoldingUsingMusicFeature", "yes"
        "ReAnswerTime", "3"
        "ReleaseToBusyFeature", "yes"
        "OffHookWarningAnnouncement", "no"
        "UseT38", "yes"
        "BRelToCT", "yes"
        "ARelToWT", "yes"
        "HDCycleTime", "300"
        "RelToneTimeout", "30"
        "ADTRelToneTimeout", "20"
"FeatureList",
"ThreeWayFeature,AnnouncementFeature,HoldingToneFeature,HoldingUsingMusicFeature,ReAnsw
er\
Time,ReleaseToBusyFeature,OffHookWarningAnnouncement,K1Flow,UseT38,LowBattRelease,BRelT
oCT,ARelToWT,HDCycleTime,\
RelToneTimeout,ADTRelToneTimeout,DSGather,DSGatherLength,Dph1,AuditRoot"
        "CallIsWaiting", "no"
        "RegisterBeacon", "2"
        "15.reply", "MEGACO/2 [135.112.125
Reply = 15 {
   Context = 0 { Notify = C0/1 }
}"
        "FindCmd", "Free"
        "FindPassCnt", "0"
        "NewLineIndex", "1"
        "templatename", "AddtoContext"
        "ContextCreated", "yes"
        "PrimaryValid", "yes"
        "AmmRequestMultistreamStreamID", "1"
        "LocalRemoteElementName.98.0", "v"
        "LocalRemoteElementValue.98.0", "0"
        "LocalRemoteElementName.98.1", "o"
        "LocalRemoteElementValue.98.1", "- 656038 656038 IN IP4 135.112.125.215"
        "LocalRemoteElementName.98.2", "s"
        "LocalRemoteElementValue.98.2", "-"
        "LocalRemoteElementName.98.3", "c"
        "LocalRemoteElementValue.98.3", "IN IP4 135.112.125.215"
        "LocalRemoteElementName.98.4", "t"
        "LocalRemoteElementValue.98.4", "0 0"
        "LocalRemoteElementName.98.5", "m"
        "LocalRemoteElementValue.98.5", "audio 8002 RTP/AVP 0 8"
        "LocalRemoteElementName.98.6", "a"
        "LocalRemoteElementValue.98.6", "rtpmap:0 PCMU/8000"
        "LocalRemoteElementName.98.7", "a"
        "LocalRemoteElementValue.98.7", "rtpmap:8 PCMA/8000"
        "LRList", "98.0,98.1,98.2,98.3,98.4,98.5,98.6,98.7,98.8"
        "LocalRemoteElementName.98.8", "a"
        "LocalRemoteElementValue.98.8", "ptime:20"
        "KEY:98", "8"
        "RecoverCount", "0"
        "MTAStreamID", "1"
        "ActiveContextId", "101"
        "MTAConnectionId", "Eph4001.1"
        "Eph4001.1.Context", "101"
```


Figure 8.1: Example of Name/Value Pairs from an Active H.248 call

of the name/value pair list. In both "links" and CPU time, smaller is better. Expanding the number of buckets to 4096, demonstrates that Lookup3 did the best job of distributing the keys, however MurmurHash3 was very close.

The name/value pair lists used in the flow engine can be sized for different purposes. For the lists that are likely to have 10s of elements, such as those that are created as a result of the first phase of input message scanning, only a few hash buckets are created. For lists that may have 10s of thousands of entries, the name/value pair list is created with thousands of hash buckets. The name/value pair structures themselves have a plug in for the hash function. In small name/value pair lists, the Horizontal Checksum hash is used, with the MurmurHash3 used for lists that may be much larger.

8.1.2 Handling Timers

LiFE is suitable for describing classes of protocols in which timing elements are of the order of 100 ms. This includes all of the major VoIP call control protocols including Session Initiation Protocol (SIP), MEGACO / H.248, and Media Gateway Control Protocol (MGCP).

Timers are one of most important functions of event handling. All data-exchange systems must have a timer somewhere in the system to ensure that an exchange completes. If one considers a VoIP system that supports 50 thousand registered users, without any active sessions, the system will have to support a timer for each registered user. Of course, to be useful, the system will have to support the creation and management of some sort of communication session. Many times, and in the case of the use cases discussed in Chapter 7, the communications are handled as half-calls. That is, for each call initiation into the system a session will be created to represent the user endpoint. A call coming into the system will result in the creation of an additional session for the called party. Together, these two sessions, sometimes called call-legs, represent the call. Each call-leg has its own timer, to keep track of call progression, and possibly the call duration. Along each step of session creation and destruction, timers are used to allow the protocol to appropriately deal with lost control packets or disconnected endpoints. For the

most part, the timer is set, the expected message arrives and the timer is then canceled. This behavior continues as the session progresses through its lifetime. As a result, most timers never fire at all, but are simply canceled. This means that the system must create a timer and make sure that the timer is ready to fire at the correct time. That same timer must be located so that it can be canceled.

Clever designers are able to move the responsibility away from the server and onto the client, since the resultant system avoids the need to have a node that is responsible for managing a large number of timers. But, alas, the common VoIP protocols are all saddled with timers for both clients and servers. In addition, most applications that manage transactions relying on direct user interaction need the capability to define timers.

It is tempting to use a simple approach to manage timers, such as a sorted link list of timer elements. In this approach, the next timer to expire will be kept at the head of the list so that after a list update a system timer could be set to expire at the time of the next event. An advantage of this approach is that timers of arbitrary precision can be supported. The penalty is that the list must be searched in order to insert a new element. If the list is kept in a linear (linked list) form, the system will incur an O(N) penalty, using a binary tree will reduce the time to O(log N), however with a large number of timers, this still results in a large processing burden. Canceling a timer can be done in a straightforward way, if one is willing to keep a pointer to the timer element, and chain the timers together in a doubly linked list. If these two conditions are met, canceling a timer element can simply be carried out by healing a doubly linked list. One is still left with the challenge of how to organize and manage large number timer elements without creating a large CPU burden.

Use of quantized timers on a timer wheel

Call control protocols, such as the ones described above, can be described using timer elements that have approximately 100 ms of resolution, SIP, MGCP and H.248 fall into this category [RFC 3261]. Narrowing the application of LiFE allows an alternative realization of timers that perform substantially better than the baseline approach outlined above.

The application of the BSD timer wheel approach [45] is used in the current version of LiFE. This strategy involves quantizing the requested timer value to the nearest 62.5 ms, and then creating a simple hash of n least significant bits of the seconds and the most significant m bits of microseconds to form an index into the timer wheel. The resolution of 62.5 ms was chosen due to two factors: 1) as indicated above, most of the control plane protocols that the flow engine executes do not have a requirement for timers greater than 75 ms, and 2) one can simply use the most significant 4 bits of the microsecond member of the standard unix timeval struct. A timer element is then added to the head of a list of timer elements contained in the slot. Inserting the timer element, therefore, does not involve any searching. As a consequence of the structure of the engine itself, timer requests are always requested as offsets from the present time, therefore a timer will always refer to a future event.

A check is performed to see if the element maps into the current revolution of the wheel; if it is more than one wheel revolution in the future, the element is put on a second wheel structure – the future wheel. In practice there are very few elements that are mapped beyond the current timer wheel. The current design uses a 64k (65535) element timer wheel which, when a 62.5 msec slot is used, represents an interval of about 4096 seconds or about 68 minutes.

In general, most call-control protocols do not have timers that exceed 68 minutes, however, the particular features that are realized as part of a telecommunications element (such as a softswitch or application server) can certainly have a requirement for timers that span over an hour or even multiple hours. An example implemented using LiFE that comes to mind is that of a calling card. In this case the system has to terminate a call if users exceed the provisioned time on their calling card. This is easily done by setting a timer that represents the total time left on the call card when the call is established. When the timer expires, the system checks to see if the account / calling card has been replenished since the call was started and, if not, the call is terminated.

When the timer task is started, an initial computation is made to determine the current slot in the timer wheel. That slot is checked, and if there are any elements present, each element on the list is then

processed in turn as expired and timer notifications are created for each timer element. Then, the corresponding slot in the future wheel is examined. Each element that corresponds to the current revolution, that is within 4096 seconds of current time, is moved to the current timer wheel, leaving the elements that expire in more than 4096 seconds. The task will repeat these actions after the passage of 62.5 msecs. Removing a timer element before expiration (i.e., cancellation), is performed using a pointer to the element itself and exploits that the list is doubly linked. Therefore no searching is required in this case

The flow engine is designed with a separate timer thread. Moving this function to a separate thread contributes to the scalability of the engine. LiFE is designed to support a large number of simultaneous contexts, each one having its own timer. Figure 9.7 located in Chapter 9 clearly shows the inordinate CPU load that can result from even a moderate number of timers, if the software design is not done in a thoughtful manner.

8.1.3 Prioritizing Message Processing

An important design goal for any system is stability. It is desirable for a system to deal with temporary overloads in a way that it continues to do the maximal amount of useful work. During a call setup phase, the system will use resources, both memory and CPU (such as timers), during the call teardown phase, resources will be released. In addition, there are certain points within the protocol exchange, that may provide an opportune place to quench the peer, inhibit additional messages. A ready example is the 100 Trying response to a SIP INVITE message. Prioritizing the INVITE message, and responding with a Trying will result in the endpoint going into a wait state, rather than continually retrying the INVITE. Prioritizing messages that result in freeing resources over others, such as those that result in the allocation of additional resources, can help to provide additional margin for stable operation. A necessary requirement is a mechanism that provides a context sensitive method to identify which messages should be prioritized. The US Patent #2009021981, “A Method for Prioritizing Message Flows within a Formal State Machine Execution Environment” outlines such a

mechanism.

The method described applies to a software application that is constructed using an abstract state description language in an environment which reads in and executes the results of the flow graph definition language.

Software applications, specifically, Voice over IP call control, have been realized using an approach in which a high level language is used to describe the data processing and message-exchange behavior of a VoIP Media Gateway Controller (MGC). In this system, an "execution engine" accepts a high level "graph language" used by the engine to create internal data structures which realize the behaviors described in the input language. The combination of the execution engine and the file which contains the graph language describing the VoIP MGC behavior results in a system that exhibits the desired MGC processing characteristics. The language supports an enhanced transition graph model and, as a result, the computation can be best understood as a graph consisting of a set of nodes connected by arcs. Using this model, there may be many contexts that exist at a given node. The arcs represent transitions contexts take between nodes. Contexts traverse arcs in response to events which can be derived from either timers or messages.

In many systems, especially those that implement VoIP call control, the system must perform predictably in the face of high spikes of traffic. Typically, a call flow will be started with an event in the form of a message, generated on behalf of a user, when a user goes off-hook or completes dialing a destination's number. The initial event will be followed by a series of message exchanges used by the specific protocol to move the call into a quiescent state. In a non-priority system under heavy/overload conditions, all messages, and therefore transitions between states, will be given equal priority. This may result in a delay for the completion of a call-setup message sequence (representing a call accepted by the system and already in progress), as requests for new service (new calls) will compete with the calls in the process of being setup or torn down for resources. The net effect is that additional resources will be expended for a given offered load, as the system will not give precedence to the call in a teardown phase which would be returning resources to the system.

A Summary of the Prioritizing Message Flows Method

The method to mitigate this situation presents itself in the case of a formal state machine execution environment by using an explicit control flow mechanism (the definitions of states/places and transition arcs between them), and the two-stage processing of the messages – the first phase is a preliminary pass which may not scan every message line, and the second is a pass processing the messages using the context of the computation to choose a specific message-parsing behavior. Using a combination of the preliminary parsing and the target token's position in the call-flow graph (the token's state occupancy), it is possible to assign a priority to the subsequent processing of the message within the formal state machine execution environment. Using this priority, call flows that are in the midst of being set up, or more importantly, being torn down, can be processed sooner, thus minimizing the peak resources needed by the system to function responsively and those providing extra margin for handling overload.

Managing Overload within the Flow Engine

"A Method for Managing Overload Control within a Formal State Machine Execution Environment" US Patents 8141065 & 8146069"

Building upon the information that is available to the execution engine as a side effect of prioritization of the messages, the overload-control method uses the queue length of the lower priority messages to set a dynamic policy for treatment of message processing. Since in this system, messages are partially processed upon reception, this mechanism can be used to selectively discard messages that are either unassociated (not part of an existing call flow/computation) or unsolicited (not an expected response). As the load on the system reaches a predefined "yellow zone" queue length of the lower priority messages currently waiting processing, the system would start to discard messages currently unassociated with a current flow. If the queue length of the lower-priority messages continues to grow, reaching a predefined "red zone", the system

begins to discard messages that are associated but are unsolicited. As the queue length shrinks below each of the thresholds described above, the system ceases discarding the related class of messages. In this way the system will maximize its ability to gracefully handle extreme spikes in message traffic, thus minimizing the impact on memory and CPU burden on the host system.

Figure 8.2: Name/Value Pairs from an H.248 VoIP Session

The execution engine is realized as a set of cooperating threads. The current architecture is comprised of a timer thread, a thread responsible for advancing contexts from place to place, a multiplicity of threads used to transform messages into name/value pairs and a multiplicity of threads to transform name/value pairs into messages as per Figure 8.2.

Each instance of a transformation — whether it is an input transformation or an output transformation — has a dedicated thread. This is true of both generic (string based) or plugin transformations. If a graph is defined with multiple generic (string) filters, each instance will be realized in its own thread. Each thread is linked to the main thread using a separate unidirectional queue. The structure of the program is such that there is only one multiple-writer queue, the input queue to the main task; the other queues are

single-writer/single-reader queues, which can be implemented without locks or read-modify-write instructions (which tend to be very slow on multiprocessors) [34][3]

The key data structure in LiFE is the *context*. As described earlier, the context owns the data structures that hold the results of input message processing and is used to create output messages. Over the life of a context, it is processed by a number of threads: input, output, timer and the main thread. Rather than use system locks to prevent concurrent access to portions of the context's data structures, the context is suspended from execution. Suspending a context removes it from access by the main thread and places it on a list that is owned by the particular input/output task. Once the input/output task has finished its operation on the context, the context is resumed and therefore accessible by the main task.

The current architecture could support multiple threads for advancing contexts from graph place to graph place, based upon messages and timers. In this case the context address space would need to be partitioned to support scalability. Performance testing has indicated that the current timer task could support a total of 4 context-advancing threads, given the the relative CPU burden of these two task types. Thus the overall machine would only be able to scale up to the use of ~10 cores if all of the traffic was restricted to one protocol type, such as Session Initiation Protocol (SIP). This suggests that a number of instances of LiFE would need to be executed to take full advantage of the coming generation of massively multi-core processors [22].

Late everything

Message transformation in LiFE is accomplished using constructs called *filters* and *templates*. Filters are used by a service to parse an incoming message into a set of name/value pairs and a named event. As part of the message processing, the message may be associated with an existing context. The resulting event + name/value pair set is routed to the existing context for processing. Templates are used to transform a context's name/value pair list into a message. For the

22 http://news.cnet.com/Intel-readies-massive-multicore-processors/2100-1008_3-6190856.html (Accessed June 2013)

generic class of services, which can describe textual protocols, the filters and templates are described using files. The files contain an ASCII description of the desired transformation. All template files are associated with a particular arc at a particular place in the computation graph. When an event triggers the execution of a specific SEND_MESSAGE arc action, the engine then evaluates which template file should be used. If this is the first time that the particular template file has been referenced at this place, then the text file is parsed at that point and the data structure that results is deposited in the current state's name/value pair list so that subsequent operations can reuse the results of the template parse. If the template file does not exist when it is referenced, an ILLEGAL FILE exception is raised for the context. If this is not handled, it will cause the context to terminate. A potential speed up for the engine could result if the template file name is invariant, so that no evaluation is needed when the SEND_MESSAGE arc action is encountered and the template file is parsed at startup and cached at the graph place. This would eliminate useless evaluations of the name of the file which may save additional time.

In a similar manner, the arc-specific filters are also cached. These are slightly different as some amount of parsing must always be performed upon reception of a message. This first baseline parsing may not need to be complete, but must provide enough information from the message to allow it to be routed to the owning context, or if no owning context is found, to be routed to the root context (Context 0). There are facilities within the infrastructure of LiFE to allow specific template and filter cached instances to be destroyed. This mechanism permits the behavior of a LiFE instance to be changed while it is in operation. The filter/template-update mechanism when paired with the graph update facility allows significant behavioral changes to be introduced to running LiFE instance without the requirement to bring the system down. The update-in-place capabilities as described are also complemented by the ability of two instances of LiFE to run in an active/active configuration. In this case, contexts are created on a secondary LiFE instance, and updated in a manner described by the graph. If the primary LiFE instance ceases operations, the secondary LiFE instance will activate the

contexts associated with the now defunct LiFE instance. If the orignal LiFE instance is restarted, possibly with a new graph loaded (an update), the secondary LiFE instance will offer the contexts back to the original owner, the now restarted LiFE instance. If the new instance refused, the contexts are kept by the secondary and promoted from secondary to primary contexts.

Even Later Binding

As described above internal instance of filters and templates are parsed and cached when first referenced during execution of the graph. The initial read in of the graph file, creating the internal representation of the graph file has also been previously covered. In addition, the engine supports the notion that a portion of the graph is not yet defined at start up time. These portions of the graph are referred to as deferred parts of the graph. When a context encounters a state that is declared as deferred, that context and any subsequent ones are put into a suspended state waiting for the resolution of the deferred portion of the graph. This is similar to a context waiting for a resource (such as an Agent) to become available. The engine (graph resolver service) then uses the description contained in the definition of the deferred state to resolve the incompletely defined state. If the graph resolver service is not able to resolve the portion of the graph referenced, all of the waiting contexts will be thrown an "UNKNOWN_STATE" exception. If the context's supplementary state does not handle this event, then the context will be killed. More likely, the graph resolver service will be able to use the description to create the incompletely defined state. In the course of resolving the undefined state, multiple new states may be created. All states in the graph must have distinct labels, so if the additional states would clash with existing states, the creation will be aborted and any waiting contexts will be thrown the "UNKNOWN_STATE" exception. The newly created set of states may contain incompletely defined states as long as the original incompletely defined state is completely defined. Using this facility it is possible to define a graph that initially consists of a single state and grows based upon the types of events received.

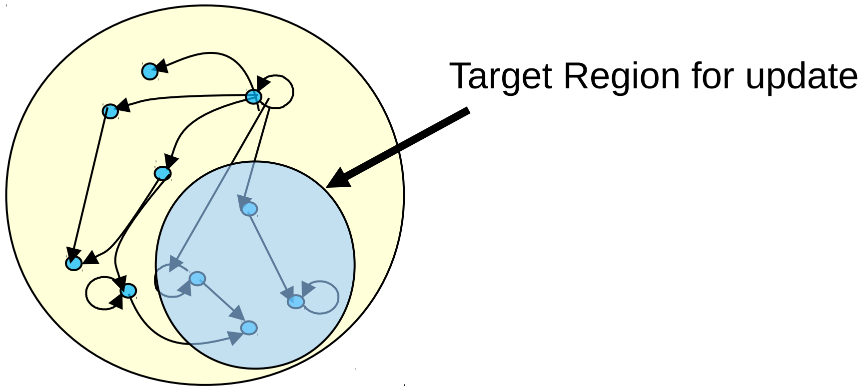


Figure 8.3: Update in Place

“Methods for Update in Place”

Although the computational model of LiFE is expressive, having a fixed set of explicit states ultimately does impose limits on the flow engine's use. LiFE's design principles provide a well described execution environment that allows a straight forward approach to provide updates to service logic, even while the system is in operation. Captured in US Patent # 8,146,069, awarded on March 27th 2012, the following section describes how the flow engine can be updated.

Detailed Description of the Invention

The present invention enables software to be updated, in place, without requiring the software to be stopped and restarted, thereby enabling portions of software that are not being updated and, optionally even portions of the software that are being updated, to continue to function while portions of software are being updated. The present invention enables explicit control over a graph such that a target region of a graph may be isolated from the remaining regions of the graph and the target region of the graph may be replaced with a new region of the graph. The present invention also enables an undefined portion of a graph to be defined dynamically. Although primarily depicted and described herein within the context of

updating a VoIP call control element software application, the present invention is applicable to any software constructed using a graph language.

Referring to the example in Figure 8.3 :

Any contexts that are occupying the target region are examined to insure that each token is at a “safe state” - that is a state that has a defined correspondence to a state in the replacement region. Once all contexts are in a “safe state”, this collection is suspended, the states corresponding to the target region are destroyed, and the new region is read into the graph.

There are variations in the way the new graph region can be introduced, including keeping the old region, renaming the entry points to seal off that region for new contexts, and introduce the new region for new contexts and any contexts that are already at “safe state” thus allowing more flexibility in the time frame of the update – allowing all contexts to continue to run, thus providing very small impact on computation even in the updated region.

As shown in Chapter 5, the state of a computation in the flow engine can be described as a combination of a token's place, and the contents of its name/value pair list. It must be pointed out that there are additional aspects to the state of a computation which include: other contexts (most easily seen in the example of the B2BUA), and state of the underlying flow engine internals, the operating system, attached network and peer applications. A checkpoint approach to perform updates on a system has been previously discussed. However, checkpointing the token will not address the state of the host (flow engine, operating system). The update in place approach, however allows the service logic to be changed / augmented, while preserving the state of the underlying system (both flow engine and operating system), so in this regard, updating the system using this method should lead to less inconsistencies.

The desired service is realized using a flow representation of the computation as depicted graphically below.

- A multiplicity of token circulate within the graph and execute functions during transitions between states.
- Transition functions include sending messages to other contexts and to external devices; and parsing incoming

messages from other contexts or external devices.

In this chapter, a number of key design principles have been covered. Each facet of LiFE's design contributes to the system's flexibility and scalability. In the next chapter, the elements of the design are brought together to show the impact on overall system performance.

9 Performance Evaluation

Many design choices exist when creating a message processing engine. One could strive for an implementation that is designed for embedded system use, in this case the result should have the smallest memory impact, and the most efficient use of limited processing power, even at the expense of application scalability. A design such as this should be created as a completely single-threaded application, relying upon a software structure that used non-blocking system calls and a call-back structure to provide service to a large number of independent external event streams. On the other hand, if the design criteria reflects a desire for maximum scalability on multiprocessor systems, a heavily threaded approach makes sense. In the following sections the performance/efficiency impacts of a range of software architecture trade-offs will be explored, from compact single-thread implementations (v3.6.0) to a heavily multi-threaded design (v4.4.9). In addition, a "mildly" threaded design will be benchmarked, one that offloads specific functions to helper threads, while keeping the bulk of the state management duties within a main thread (v4.4.6). In this case the event streams were messages associated with the control plane of various Voice-over-IP protocols.

Variations of the basic LiFE implementations are used as the context for the design discussions in this chapter. Each LiFE design variation is identified by version numbers, such as "v3.6.0" or "v4.4.9", mentioned above. The version numbers will be used to identify

which specific variants as they are discussed in the course of this comparison.

```
#
#   The loopback registratation request handler.
#
DEF_STATE LoopbackStart(0)
   STATE_NVP=
      "STATE_NAME", "LoopbackStart(0)",
      "STATE_TAG", "LoopbackStart",
      NULL
   STATE_ACTION=NULL,
   STATE_ARCS=
      IS_TIMEOUT=FALSE, TIMEOUT_VAL={0,0},
MATCH_STR=NULL, ARC_ACTION=PROCESS_MESSAGE,
         NVP_LIST=
            "FILTER:default", "filter.sip.register",
         NULL
         OUT_STR="NULL",
NEXT_STATE={LoopbackStart(1)}
      END_ARCS
END_DEF_STATE

DEF_STATE LoopbackStart(1)
   STATE_NVP=
      "STATE_NAME", "LoopbackStart(1)",
      NULL
   STATE_ACTION=NULL,
   STATE_ARCS=
      IS_TIMEOUT=FALSE, TIMEOUT_VAL={0,0},
MATCH_STR=NULL, ARC_ACTION=SEND_MESSAGE,
         NVP_LIST=
            "SV:default",
"@VOpaqueId@::lookup:@AC_ID@",
            "RULE:default",
"template.loopbackRegister.ack",
            "SV:default", "SET:0::this:MsgCnt",
         NULL
         OUT_STR="NULL",

DEF_STATE LoopbackStart(2)
   STATE_NVP=
      "STATE_NAME", "LoopbackStart(2)",
      NULL
   STATE_ACTION=NULL,
   STATE_ARCS=
      IS_TIMEOUT=FALSE, TIMEOUT_VAL={0,0},
MATCH_STR="HELLO", ARC_ACTION=SEND_MESSAGE,
         NVP_LIST=
            "SV:default", "INC:1::this:MsgCnt",
            "RULE:default", "template.ack",
         NULL
         OUT_STR="NULL",
NEXT_STATE={LoopbackStart(3)}
      IS_TIMEOUT=TRUE, TIMEOUT_VAL={30,0},
MATCH_STR=NULL, ARC_ACTION=NULL,
      OUT_STR=NULL, NEXT_STATE={EXIT(0)}
      END_ARCS
END_DEF_STATE

DEF_STATE LoopbackStart(3)
   STATE_NVP=
      "STATE_NAME", "LoopbackStart(3)",
      NULL
   STATE_ACTION=NULL,
   STATE_ARCS=
      IS_TIMEOUT=FALSE, TIMEOUT_VAL={0,0},
MATCH_STR="HELLO", ARC_ACTION=SEND_MESSAGE,
         NVP_LIST=
            "RULE:default", "template.ack",
            "SV:default", "INC:1::this:MsgCnt",
         NULL
         OUT_STR="NULL",
NEXT_STATE={LoopbackStart(2)}
      IS_TIMEOUT=TRUE, TIMEOUT_VAL={30,0},
MATCH_STR=NULL, ARC_ACTION=NULL,
```

Table 9.1: Loopback registration handler

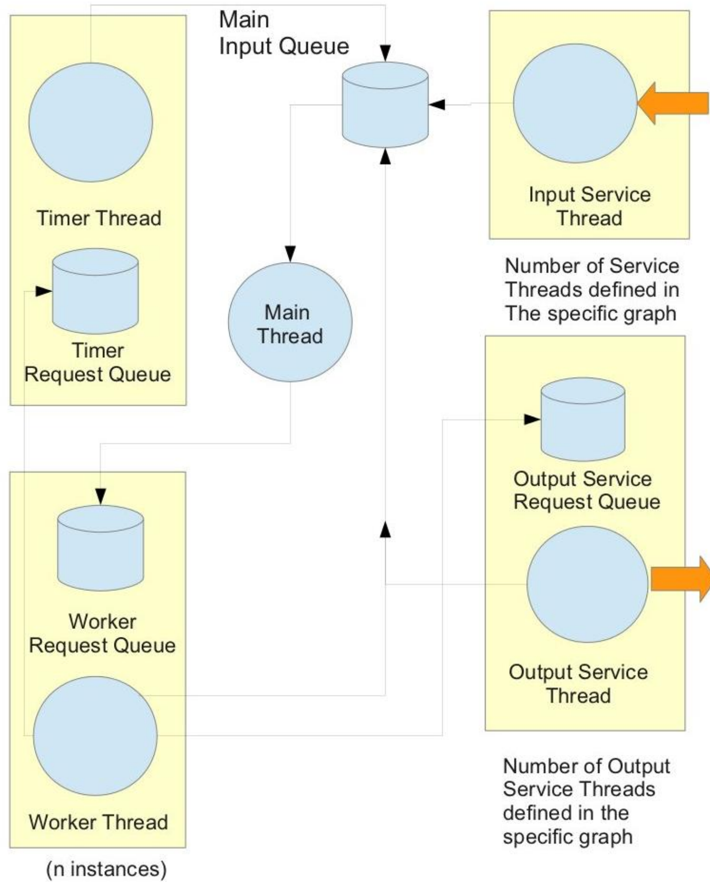


Figure 9.1: V4.4.9 Internal thread and queue structure

To obtain a measure of the performance capability of the existing software architecture of the flow engine, a simplified exchange was constructed using the graph in Table 9.1.

The graph responds to an initial client registration request with an identifier that the client uses to route incoming messages to a specific context. The client messages ("HELLO") are then counted by the flow engine. Each incoming message increments the MsgCnt variable in the token's name/value pair list. When the flow engine receives the "HELLO" message, it issues a "WORLD" response to the client. The

client “HELLO” message includes a second line of the form: X: <opaque identifier>. The client includes the value sent to it by the flow engine as the opaque identifier. Each instance of the client executes the send/receive loop for a million iterations, then reports the elapsed time.

9.1 Comparing Performance of Flow Engine Architectures

Figure 9.1 shows the current internal software structure of v4.4.9 of the flow engine. It shows a number of the major software structures in the lightly shaded regions. The number of worker threads is adjustable as a configuration parameter, while the number of input or output threads are controlled by the graph definition. It is anticipated that the number of input and output threads could be increased further, but at present an input thread is created for each built-in service that is defined in the flow graph. For each service that is defined using the “text” filter type, an input thread, a secondary-parsing thread, and an output thread is created to support processing of the specific service defined in the graph. Use of output and secondary parsing threads are tunable, so that the system can be configured to use the worker thread to perform parsing and output functions instead of using the helper threads. The input thread is associated with messages that arrive on the address tuple defined in the graph (protocol type/port). The parsing thread is called to support the PROCESS_MESSAGE verb, while the output thread is associated with the SEND_MESSAGE verb.

8 simultaneous clients (8M iterations total)		
Version	Thread Function	CPU Load
v4.4.6	Main	100%
	Input Processing (4)	42% (4 @ 10.5% av)
	Timer	20%
	Elapsed Time: 105 s	Rate: 76k iterations/s
v4.4.9	Main	31%
	Worker (x6)	192% (6 @ 32% av)
	Input Processing (4)	68% (4 @ 17% av)
	Timer	4%
	Elapsed Time: 73 sec	Rate: 110k iterations/s

Table 9.2: Component CPU Burden Breakdown

For the following tests, the first variant of the test client was used, which featured a receive operation instead of the 1 microsecond delay. In addition, every 10,000 iterations, the client outputs the current iteration count. The flow graph was modified as well, substituting the SEND_MESSAGE verb for “NULL” in the arc action of the LoopbackStart(2) and LoopbackStart(3) states. The template file “template.ack” is used as the basis for the reply to loopback client.

For the tests documented in this chapter, the “default” service was used, which does not create the secondary parsing or output threads. Thus the main (v4.4.6) or the worker threads perform any secondary parsing or output processing duties. With the example that has been chosen, no secondary parsing is used, so the input message processing is handled by the input thread, and updates of the owning context's name/value pairs and creation of the output message is handled by either the main task (v4.4.6) or the worker task (v4.4.9).

The flow engine architecture enables us to easily vary a number aspects of its structure to measure each parameter's impact on system performance. Baseline performance can be determined using the non-threaded model; however one must be careful, since a number of performance improvements (particularly around timers) were added only after the first threading model.

Intel i7 -960 4 cores					
Number of Simultaneous Clients (iterations)	CPU Burden			Elapsed Time	Observed Rate (it/sec)
	Main task	Workers n=6	Input Processors		
$2 \cdot 10^6$	34 %	126 %	39 %	36.6 s	55k
$3 \cdot 10^6$	36 %	156 %	53 %	40.9 s	73k
$4 \cdot 10^6$	35 %	169 %	60 %	46.6 s	86k
$6 \cdot 10^6$	32 %	192 %	66 %	59.1 s	102k
$8 \cdot 10^6$	31%	192 %	68 %	73 s	110k
$10 \cdot 10^6$	31%	198 %	68 %	91 s	110k
$12 \cdot 10^6$	31%	192 %	68 %	108 s	111k

Table 9.3: V4.4.9 Thread Function CPU Burden versus Offered Load

A “mildly” threaded version (v4.4.6) provides a basis to understand the CPU utilization efficiency *vs*. scalability trade-off. In this case one can directly measure the impact of multiple input process threads *vs.* a single input process. Breaking out the main task (which is shown to be the bottleneck in v4.4.6) into a message router plus a set of worker tasks, looking at Table 9.2, one can observe the impact on the resultant efficiency and throughput. Table 9.2 also shows the performance that arises from combinations of the number of workers and input processors.

Intel i7 -960 4 cores				
Number of Simultaneous Clients (iterations)	CPU Burden		Elapsed Time	Observed Rate (it/sec)
	Main task	Input Processors		
$1 \cdot 10^6$	73 %	27 %	27.1 s	37k
$2 \cdot 10^6$	99 %	38 %	27.5 s	73k
$3 \cdot 10^6$	100 %	42 %	40.9 s	73k
$4 \cdot 10^6$	100 %	41 %	54.7 s	73k
$6 \cdot 10^6$	100 %	41 %	80.2 s	75k
$8 \cdot 10^6$	100 %	42 %	105.5 s	76k

Table 9.4: V4.4.6 Thread Function CPU Burden versus Offered Load

With the ability to add additional input processing threads, the system can handle both simple input message structures, as found in fixed message structures such as Diameter & MGCP, and also handle more complex (variable) message such as SIP. It is common to have load balancers feeding a large cluster of servers [38]. In the case of telecommunication service providers, the problem becomes more challenging. A load balancer used with a cluster of SIP servers must process enough of the SIP message to route all of the traffic that is part of a single session consistently to the same SIP server instance. The simple test that has been outlined above defines a session between the client and the flow engine. Each incoming message is routed to the context that owns the session. This simple benchmark provides some insight about the performance of the flow engine used as a application-specific load balancer. ,

Using the same i7 Intel processor, with hyperthreading turned off, eight instances of the simple client were run, each for 1 million iterations. The result is shown in Figure 9.2.

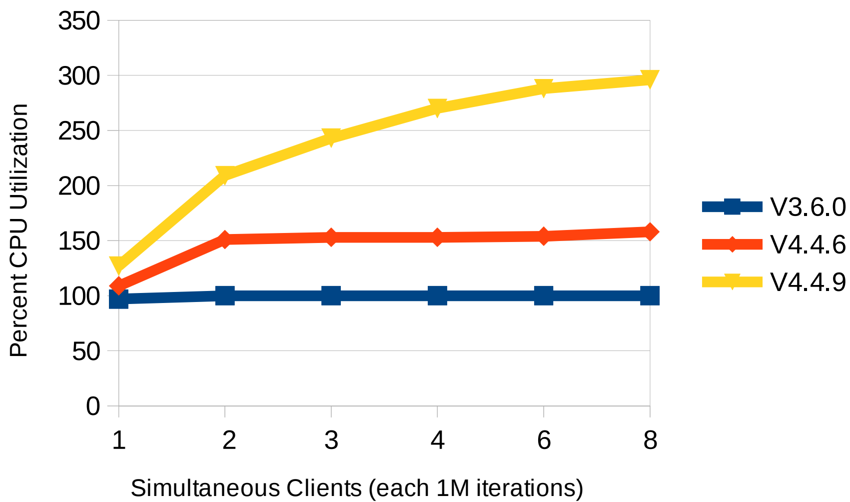


Figure 9.2: Flow Engine Architecture Impact on CPU Utilization

In addition, Figure 9.6 shows the opposite configuration, that is turn hyperthreading on and re-run the tests for v4.4.9. This is a confirmation that performance advantages from hyperthreading are configuration dependent.

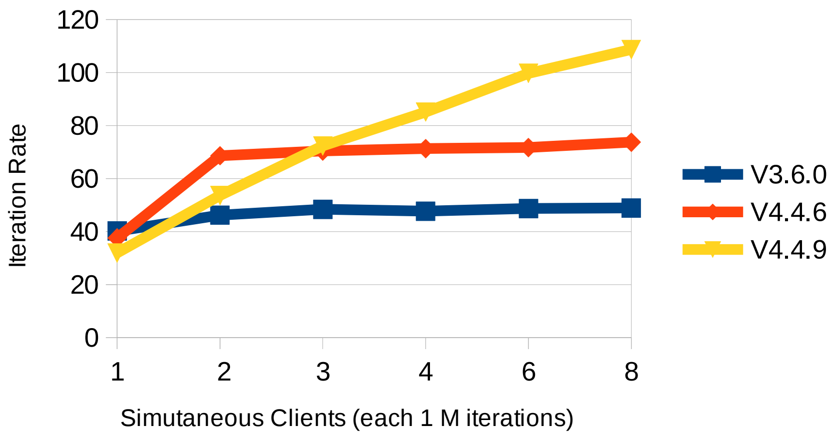


Figure 9.3: Flow Engine Architecture Impact upon Performance

Figure 9.3 clearly shows the benefits of progressively threading the flow engine. The single-threaded implementation (v3.6.0) quickly gets close to its maximum capabilities with a single client running, and reaches its maximum with two clients running. At 40k iterations/s, the single-threaded implementation is receiving, parsing, and routing incoming messages at the rate of 40k messages/s. In addition, the implementation is updating the proper context's name/value pair list by incrementing a counter, and then synthesizing a reply message and sending it back to the proper client.

Subsequent charts will show the CPU impact of each of the implementations, but one can see that both the single-threaded and mildly multi-threaded variants hit their limits within the range of the tests. Both of these versions are well behaved when saturated, as additional load does not degrade the overall throughput of the system.

The observed iteration rate continues to rise in the case of the heavily multi-threaded version. Over the range of the testing, the rise continues at a near constant slope. It should be noted that during the eight-client test, the system load utility indicated that the processor, a four-core, single CPU was in the idle state less than 7% of the time. Since this is an average measurement the system itself was totally

utilized.

Figure 9.4 captures the throughput of each architecture. The simple loopback client first registers with the flow engine. This causes a new context to be created for each client. The client includes the session identifier in each message sent to the flow engine. On the flow engine side the session identifier is used to route each message to the owning context. Once registration has been accomplished, the test client uses gettimeofday to save the starting time. The client repeats a loop that consists of sending a message to the flow engine, and waits for a reply. The client performs this sequence a million times, then checks the time to compute the elapsed time.

The faster the flow engine can respond, the shorter the elapsed time. In the performance charts the number of simultaneous clients running is indicated on the x-axis. All of the clients are started at the

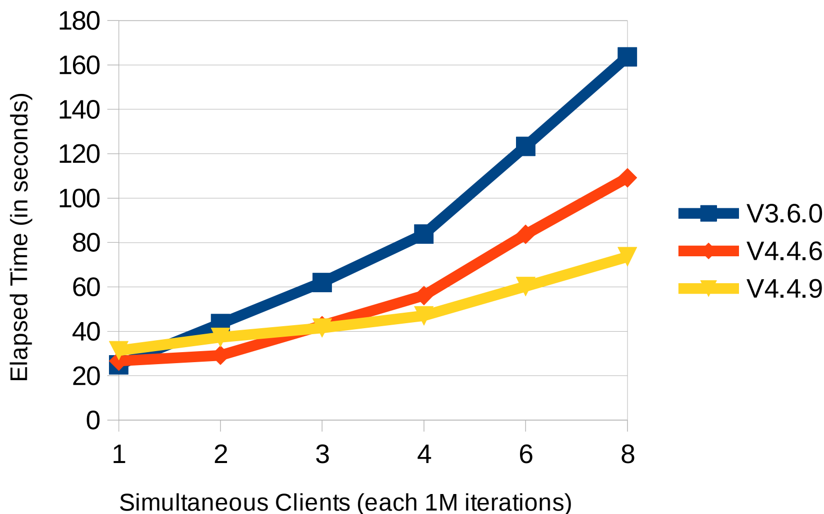


Figure 9.4: Flow Engine Performance

same time. In Figure 9.4, the single-threaded version of the client completes the one million iterations more quickly than the other two flow engine designs, but is overtaken as the offered load is increased.

This chart simply shows that both the single threaded and the

mildly threaded versions of the flow engine hit performance bottlenecks by the time two loopback clients are running (40k & 70k iterations/s respectively). Looking at the task/CPU breakdown in Table 9.4, in the case of the mildly threaded version (v4.4.6), the main task is the limiting factor. Table 9.3 shows the breakdown of CPU utilization of v4.4.9. In this case, there is no obvious limit that has been reached in this case. The critical main task has never gone above 34% CPU utilization. In the case of the eight client case, the system has run out of capacity, but looking at the pattern of the Main task versus the workers, suggests that the system could support significantly more throughput.

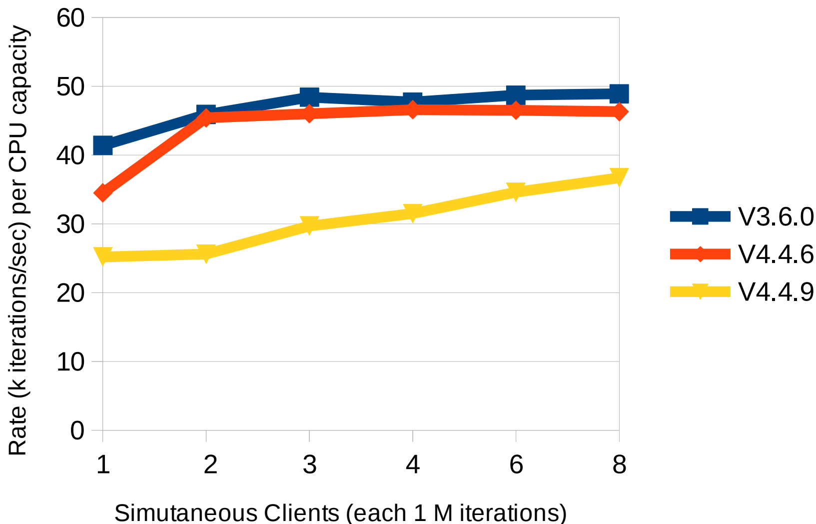


Figure 9.5: Flow Engine Efficiency

As Figure 9.5 shows, the efficiency, that is the percentage of CPU used per iteration varies between the three software architectures. Throughout the range of tests, the single threaded implementation remains the most efficient, with v4.4.6, the architecture that has timer and input processing helper tasks is very close as the load increases. Looking back at Figure 9.3, which depicts the overall throughput, both the single-threaded (v3.6.0) and the mildly threaded version

have reached the maximum-rate plateau when two clients are used to create the test load. The multiple-worker thread architecture (v4.4.9) has a significantly lower efficiency than the other approaches.

The gap in efficiency between the multi-worker software architecture and the single threaded versions could also suggest that the message router / worker thread architecture should be changed. In particular the v4.4.6 software structure does the majority of event processing directly, bypassing the work to queue requests and synchronize between a main task and the workers. A more efficient design could have a set of tasks that read from a main event queue, obtain the context referenced by the incoming event, and then act upon the event, performing all of the arc action name/value pair processing as well as arc actions as appropriate. At the end of the processing loop, the task would deposit the context as appropriate, perhaps placing it at the end of the run queue. Just as in v4.4.9, each main task must ensure that the context referenced in the incoming event is not currently being processed by another “main” task. In comparison to the current v4.4.9 design, the new design would eliminate filling out a message and setting a semaphore. However, since multiple consumers would now be reading the main input queue, instead of the current single main task, a mutex would have to be used for the main queue where none is needed now. However, there the proposed approach would still eliminate the worker status return message on the main queue and the associated semaphore, potentially increasing the efficiency.

As noted above, the testing of v4.4.9 suggested that the throughput may be limited by system resources, rather than the architecture of the flow engine. To gain additional insight, the computer system was restarted with Intel Hyperthreading[23] turned on. Upon subsequent reboot, the Linux kernel indicated the presence of eight cores, rather than four. Using v4.4.9, experiments were run using various instances of the simple client. Figure 9.6 shows the performance comparison for the system with and without hyperthreading, using flow engine v4.4.9 at two, four, eight, ten and twelve simultaneous instances of the loopback client.

23 http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html (Accessed June 2013)

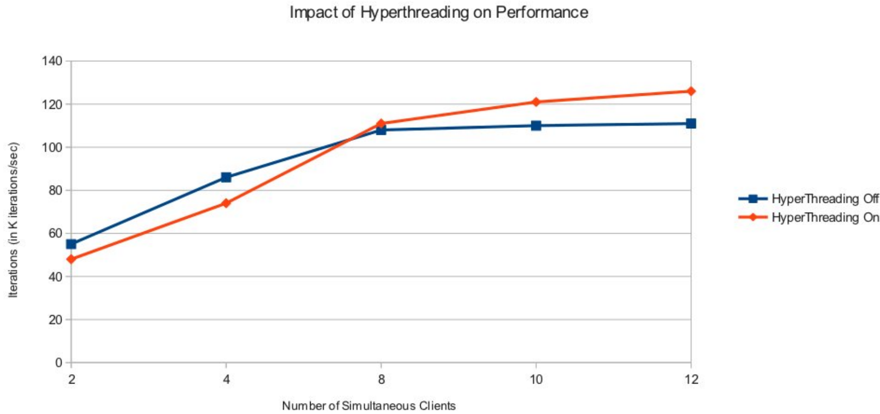


Figure 9.6 Impact of Hyperthreading on Performance

The experiments conducted above were designed to measure the potential of the flow-engine framework. To assess the impact of a more complicated input message, the simple benchmark was modified to send a SIP INVITE message to the flow engine. In this experiment, the flow engine must scan the incoming message to find the field designated as the call id, which is used to route the incoming message to the correct context. In this case the INVITE message has been captured from a CISCO 7900 SIP phone and adapted for use in this test. It should be noted that the INVITE message is not parsed or validated for this test, rather it is scanned to find a field to be used to route the message. Parsing and validation of the message can be accomplished by the worker tasks once the message has been routed to the proper context.

The experiment used the heavily multi-threaded version of the flow engine (v4.4.9), configured to use seven worker threads, four input processing in addition to a thread each for message routing and timer management. Multiple runs were used to determine the elapsed time needed for the system to complete eight simultaneous instances of a client running one million iterations each. The result was that using the simple message resulted in 108k iterations/s, processed, while the experiment that used the INVITE message returned a result of 104k iterations/s. To be clear, although a SIP

INVITE message was sued as input, the message was not parsed as a traditional SIP stack, would do, rather the message was scanned to pick out configured portions of the SIP message that are defined by the protocol to be unique. In this case the Call-Id was used, however, a tuple comprised of the “From” tag, the “Via” branch and the Call-Id could also have been used. Since all of the SIP message was scanned as part of the existing experiment, the impact of using additional portions of the header to create the basis for routing the SIP message should be minimal.

While the overall rate of message processing is slower for the case of the INVITE, the flow engine is still indicating well over 100k iterations per second.

9.1.1 Timer Performance

Looking at the message exchange of Figure 9.10, one can see that for each call there are two outstanding timer elements (per 1 half-call). In addition, there is an additional timer element that is used for each registered endpoint. For our benchmarked systems, one can ignore any additional timers added for an endpoint, however in a deployed system, there are a large number of endpoints that are not actively making calls at any time. If the offered load is 1600 calls per second, and the message timeout interval is 30 secs, then there are 1600 * 30 * 2 timers in use (96,000) even when there could be as few as 2 * 10 or 20 in use if the 10 calls are active. As part of the context cleanup processing, the flow engine now sends a single message to the timer task that cleans up any outstanding timers running on behalf of a single context. The result has been a reduction in the memory usage of the system, especially for circumstances that involve many short-lived communication sessions. Although it sounds like this is of use chiefly during benchmarking, there are real world situations that mimic this benchmarking situation, specifically, wireless applications such as e-mail clients routinely create very short connections to check for e-mail and, as a result, a large number of sessions are set up and torn down by the wireless base station [23].

An alternative would have been to send a message to the timer task for each outstanding timer object. In the case of a protocol that only permits one message to be outstanding at a time (MGCP), the

solution is quite simple, each session may only have a single message timer running at once. However since the flow engine is intended support a variety of control protocols, supporting standards such as H.248, that permit multiple messages to outstanding at once requires a general solution. Another approach would be to selectively retire each timer as permitted on a protocol by protocol basis. This would require that the flow graph programmer take care of this accounting for each scenario to be supported, and would result in additional internal messages sent between the main context control task and the timer task. The current flow engine facility to automatically retire all outstanding timers when the associated context is destroyed reduces the number of internal messages, does not require additional effort by the flow graph programmer, and provides very similar resource recovery characteristics to the selective timer destruction approach.

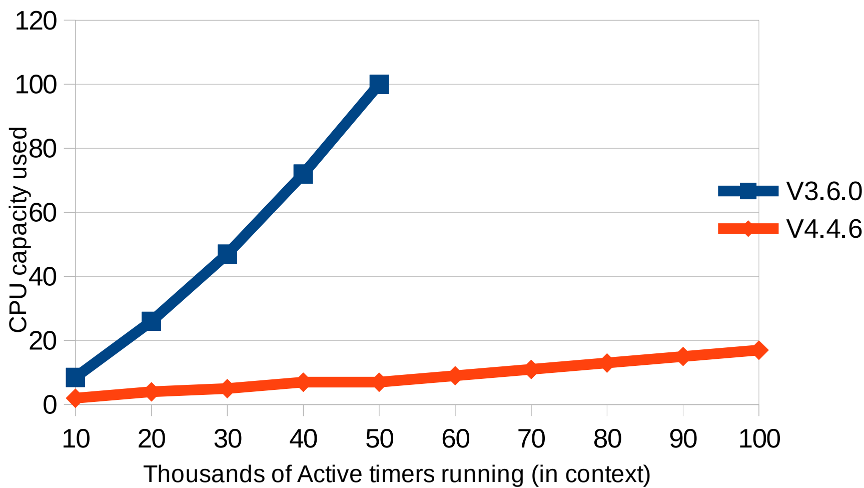


Figure 9.7: Flow Engine Timer Performance

Figure 9.7 shows the comparative timer performance of the flow engine, with and without the implementation of timer quantization. The chart was created by running a simple client that creates the number of contexts shown on the x-axis. After the initial message exchange, each context proceeds to a state that has a 1 second timer.

When that timer fires, a counter is incremented and checked against an iteration limit (50). Once the iteration limit is reached, the context is killed. If the iteration limit has not been reached, the context loops back to the same state, which restarts the timer. Thus during the execution every second a timer is being fired. Therefore in addition to timer processing, there is processing performed that accesses each context's name/value pair and controls the movement of the context within the graph. In the chart, for the calculation of v4.4.3 above, the CPU burden is the addition of the main task and the timer task. During the testing, the total CPU burden was split almost equally between the main and timer tasks, through out the range.

It should also be noted that in the case of v3.6.0, the single threaded case, once the system hit 100% utilization, new jobs were accepted very slowly, complicating measurements above 50,000 timers and contexts.

9.1.2 Minimizing the number of *malloc/frees* – Performance Impact

As described in Section 8.1.4, the name/value pair is one of core data structures used to implement the flow engine. The flow engine supports any number of ways to realize telecomunication system functions. In the case of implementing call agent or application server (such as a B2BUA) functions, the incoming message is converted into a name/value pair list. The lifetime of the list that represents the incoming message may be brief, it may be destroyed once its contents have been processed by the owning context. In other cases, a name/value pair set may be used as the storage mechanism for a session. In this case, the lifetime is much longer than an individual message, but will only exist for the duration of a call, perhaps 180 seconds, the average hold time of a business call. In this latter case, numerous addition, update and delete operations will be taking place. In general an additional context will be used to represent the user's presence or registration. The name/value pair list associated with this type of context will be much longer lived. The uses outlined suggest that one should use a design approach which is optimized to quickly create, destroy and add values to the list. Figure 9.8 Lists the code created to capture the types of operations discussed

above. The same code was linked against two different libraries and run to provide a quantitative measure of the impact of different memory management approaches on the performance of the name/value pair lists. In case *A*, *malloc* was used to obtain the memory used for holding the name/value pair components. In case *B*, the library requests a block of memory, uses that to manage all of the components of the name/value list.

```
int
utest_nvl()
   {
   int        i =0;
   CNvList* nv = NULL;

   for(i=0; i < 8000000; i++ )
      {
      nv = nvl_create();
      nvl_addNv(nv, "first", "one");
      nvl_addNv(nv, "second", "two");
      nvl_addNv(nv, "third", "three");
      nvl_addNv(nv, "fourth", "four");
      nvl_addNv(nv, "fifth", "five");
      nvl_addNv(nv, "Null", "");
      nvl_addNv(nv, "duplicate", "one1");
      nvl_addNv(nv, "duplicate", "one2");
      nvl_setNv(nv, "duplicate", "two");
      nvl_setNv(nv, "fifth", "five");

      nvl_delNv(nv, "first");
      nvl_delNv(nv, "second");
      nvl_delNv(nv, "third");
      nvl_delNv(nv, "fourth");

      nvl_destroy(nv);
      }
   return NULL;
   }
```

Figure 9.8: Name/Value Pair Benchmark Code

The tests were conducted on an Intel i7 960 3.2GHz processor. Table 9.5 shows the performance impact of designing the name/value pair structure so that for most contexts, only one *malloc*/*free* set needs to be used to manage memory associated with the name/value pair list. An additional benefit to this approach is an improvement of

locality of reference. Using a design that uses the *malloc* function will result in the components of the name/value pair list scattered through memory space. At the very least, each name and value string is obtained from the heap. Using *malloc* does not guarantee any spatial relationship between elements. Consolidating the memory space brings these, as well as internal management data structures, closer to each other, increasing the chance that successive accesses will be in some level of system cache. Recall that the evaluation of a token's name/value pair members take place when the context has been activated, so many members of the name/value pair list can be accessed in quick succession. A profile of name/value pair components in time would be an interesting future study. With the dramatic increase in speed of name/value pair handling, and given the central role that name/value pairs play in the design of the flow-engine, one would expect to see a measurable impact on system performance. To put the name/value pair processing gain into perspective, a series of tests were run that compared the user time needed to process 100,000 calls sent to the unit under test at a rate of 100 calls per second. This test was conducted on a dual core T9600 Intel process running at 2.8 Ghz, with 4 Gb of main memory. The instance of the flow engine that used the older memory management style for name/value pair lists, our Case A, used 403 CPU-seconds to process the 100,000 calls, while the Case B instance used 328 CPU-seconds. Even though these results are not as dramatic as those for the name/value pairs in isolation, it represents a significant gain, about 20% in efficiency in practical use.

Case *A*	Case *B*
31.1 s	7.9 s

Table 9.5: Impact of Memory Management Approaches on Name/Value Pair Performance

9.2 Evaluation of graph structures

A brief examination of the graph definition reveals that every State may have an Event definition which in turn owns a set of directives. These directives are evaluated when an incoming event matches an arc action of the token's current state. The directives are in the form of name/value pairs. In earlier versions of the flow engine, the directives were kept in their original string form. Clearly this meant that the engine would perform the same declaration parsing each time an event triggered an action. To eliminate this inefficiency, at graph read-in time, the declarations are parsed into internal data structures capturing the parsing of the declaration structure. In addition, invariant portions of declarations are evaluated and the results stored so that these actions (in many cases initialization or simple assignments) take place much more quickly.

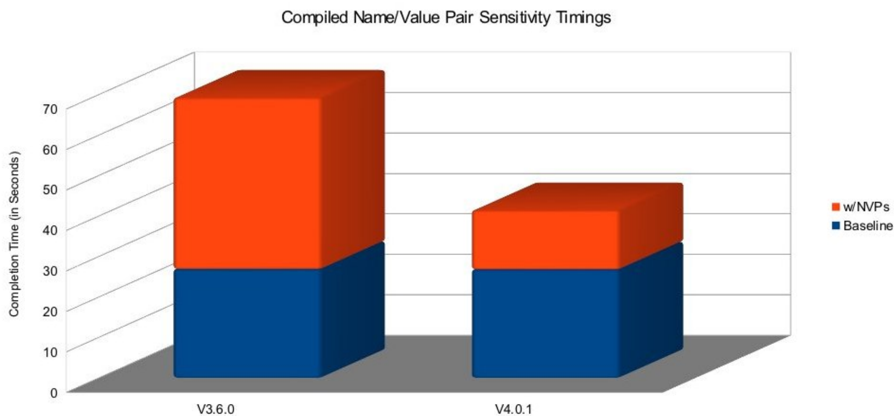


Figure 9.9: Compiled vs Interpreted Name/Value pair processing

To get a better idea of the impact of removing the repetitive processing, a set of tests were run that compare the performance of the current name/value pair processing approach to one which leaves the name/value pairs as strings and performs all scanning and evaluation when the name/value pair processing is triggered by a flow-engine event. In order to quantify the processing difference, four sets of measurements were taken: 1) a baseline measurement with a simple message and reply set on a version of the flow engine that uses the compiled name/value pair approach (v4.0.1); 2) the same

flow on a system that does not attempt to pre-process the name/value pairs (v3.6.0); 3) a version of the graph which adds a set of name/value pair directives taken from the INVITE processing of the B2BUA graph running on version (v3.6.0); and finally 4) the same graph from the previous version, now run on v4.0.1. The results are summarized in Figure 9.9. As one might expect, the pre-processing of the name/value pairs makes a substantial difference in performance of the flow engine, shrinking the n/v pair processing to about a third of its non-preprocessed processing burden.

9.3 Comparing to Other Systems

As mentioned in Chapter 3, Opensips is known as a high-performance, scalable system. The Opensips website[24] indicates that an Opensips stateless proxy can process 13,000 call setups/s. Since the comparison is for a system that is not performing authentication challenges, referring to Figure 9.4, a standard SIP exchange, one can see that there are either 6 or 7 messages that are either sent or received per call set up attempt (6 from SIPp, 7 from the flow engine side). Using the number 7, gives us 13 * 7 or 91,000 messages per second processed by Opensips. Referring to Figure 9.3, shows that the flow engine framework is capable of handling 125k iterations/sec, which is 250k messages (in+out)/s. Using the figure that includes INVITE message scanning still puts the flow engine at the 200k messages / second range. Opensips, if it is fully parsing the SIP messages, would be performing more processing on the messages than the flow engine, however, the flow engine is scanning a majority of the message and routing it to an owning context, which is then creating a message that is sent back to the originator.

Another benchmarking test that has been used to characterize the performance of the flow engine has been a SIP client set written using the SIP JAIN stack [32]. Recall that some modern complex event processing systems, such as Esper, are built upon Java, which makes it interesting to compare the flow engine to a Java-based SIP client.

24 http://www.opensips.org/ (Accessed June 2013)

Using the SIP JAIN stack to create end- user services is covered by O'Doherty, et al [53]. Built upon the Java Virtual Machine, building applications with JAIN is common, and represents a reasonable performance framework to use for benchmarking. To that end, a simple set of Java programs were created to provide the basis for performance comparison.

The test procedure consists of using a pair of programs, one initiator and one receiver to create an offered load on the flow engine. Each program has been written using the JAIN stack, a set of Java methods that is considered to be "high performance". The service logic executed the clients themselves to very straightforward, a call is initiated, held for 3 seconds and then released. Figure 9.10 shows the message-chart diagram which contains the exchange used for each SIP call.

Then in Figure 9.11, the boxes on each end of the diagram represent the two Java programs, while the section in the middle,

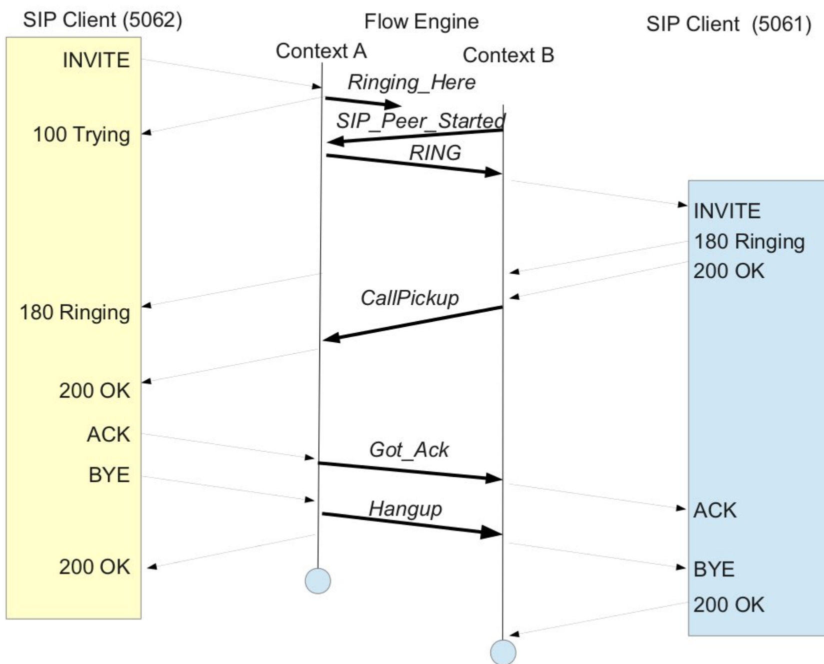


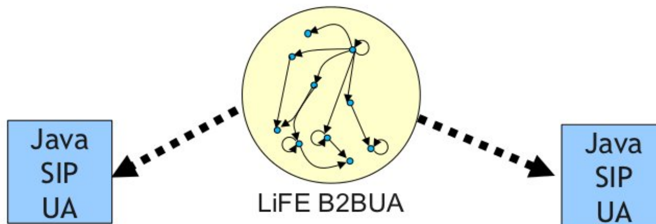
Figure 9.10: Simple SIP Call Setup

represents the flow engine. For these tests, the flow engine is using a

flow graph that describes a Back-to-Back User Agent (B2BUA) , so that two SIP sessions are being used per call. One session between the originating SIP client and flow engine, and another that is originated from the flow engine and terminates on the destination SIP client.

While Session Description Protocol (SDP) is exchanged between the calling parties, both during the INVITE and the 200 portions of the call, the Java client only performs processing related to the control plane, no media processing or evaluation is performed. Since the flow engine also does not process the bearer channel, the combination of the two SIP endpoints is processing the same external message load as the flow engine. It is instructive to measure the CPU and memory burden of the endpoint clients and compare the figures with those from the flow engine. As shown in Chapter 3, many systems

Tests using a simplified UA realized using the JAIN-SIP stack running on JVM 1.5.1 (x64) on a 4 core 3.1GHz Linux 2.6.13 machine



Call Rate calls/sec	Java Client ms/call	LiFE ms/call	Performance Ratio
500	1.98	1.12	56.5% CPU Use
700	3.41	1.27	37.4% CPU Use
750	3.89	1.32	34.0% CPU Use
Memory Use	2800 Mb	228 Mb	8.1% Working Set Memory

Figure 9.11: LiFE running with JAIN SIP Clients

are currently realized using Java, so one might get insight into the performance penalty incurred by the flow engine's design elements.

As a point of reference, the flow engine (v4.4.6), configured as a B2BUA, has been benchmarked at 4.4k call attempts per second

(half-calls), using the SIPp test client. This performance was observed on a i7 960 Intel 3.2 Ghz desktop processor with 12Gb of main memory.

9.4 Conclusions

The measurements contained in this chapter highlight the trade-offs of the approach used to create the flow engine. On one hand, the flow engine is flexible, it is not tied to a single protocol. There is always a price to pay for creating a more general solution, however, the measurements show that the flow engine compares quite favorably to VoIP solutions realized using the SIP JAIN stack. OpenSIPs is a very impressive focused system, capable of very high SIP message processing. Even here, the flow engine's ability to support 250 thousand messages per second (one in, one out 125 thousand times a second), places it in the same league with OpenSIPs, even though the flow engine can handle multiple protocols simultaneously. The use of an approach like the flow engine is not always the right choice, in an application that uses a single protocol or does not require flexible manipulation of messages, there are other solutions that offer a better fit, however, the flow engine provides a powerful, flexible way to process protocols that can sustain high message processing rates.

10 History of LiFE

The primary focus of LiFE is the description and execution of communication protocols, however the genesis of LiFE can be traced to a program started in the early 1990s. At that time, a team in Bell Labs was working on applying neural networks to hand written Optical Character Recognition as part of a system to speed the accurate creation of calling cards as part of a large federal contract. OCR performance on handwritten characters was then less than 80% accurate, so that several mitigation strategies were required to bring the overall system performance up to an acceptable rate (99.99%).

10.1 First evolution: workflow engine

At first the control of the system was handled by various job-control scripts. As the application grew in complexity, it became clear that a flexible, rapidly configurable work-flow system was necessary to keep the system from degenerating into an unmaintainable, tangled set of ever more “clever” bits of scripting. At the time, a small transition graph (TG) engine was already used as part of the OCR solution to handle incoming and outgoing faxes. The transition graph machine modeled the transitions contained in the ITU T.30 specification to command fax modems to receive and send faxes. The engine was modified so that, instead of defining the state transitions as data structures contained in C program code locked

```
MEGACO/2 [135.112.125.135]:2944
Transaction = 1312523 {
        Context = 103 {
                Modify = C0/1 {
                        Events = 2223 {
                                al/on {strict=state}, dd/ce
{DigitMap=Dialplan0}
                                },
                                Signals {cg/dt {Duration=6000}},
                                DigitMap= Dialplan0 {
                                T:16,S:2,L:8,
([2-9]xxxxxx|E[1-7]x|1xxxxxxxxxx|E9xxxxxxxxxx|E8xxxxxxxxxx|EE[
0-3][0-9]Fx) }
                }
        }
}
```

Figure 10.1: Conditioning an Endpoint to Make a Call

into a compiled executable, each state transition was defined in a language that was read in by the engine at run time.

An additional application of the early flow engine was to Internet Fax using the ITU T.38 standard. The standard wrapped the existing T.30 information frames into packets that were sent over the Internet, rather than dedicated lines. In order to compensate for the data path jitter, something that the analog based T.30 procedures were not designed around, the flow engine was used to create a stateful T.38 fax server. Rather than just translate the T.30 frames into the corresponding T.38 presentation layer protocol, the engine had the ability to spoof the local fax machine if it had detected that the required T.38 message exchange packet was late. To a limited extent, the flow engine could also keep the far end fax machine from prematurely dropping the fax session. The two T.38 flow engines were coordinating the two state engines contained within the pair of analog fax machines. This was a prime example that Telecom protocols, whether they are fax or VoIP, are fundamentally about state machines that influence one another's state. if you want to model protocol state machines you need the mechanisms for parallel

state machines to work together.

While the model for a deterministic finite state automaton (DFA) or TG fits well supporting a single entity such as a fax modem, the OCR work-flow presented additional requirements: there were many “jobs” that needed to be managed by the work-flow engine and the “jobs” could be in process for an arbitrary length of time, potentially staying active when preventative maintenance was scheduled for the machines. In these cases, the usual approach was to use a relational data base to store the workflow jobs, as there were well known ways to checkpoint and restart the data necessary to recover long term jobs. The problem was that, at the time, relational data bases were expensive in terms of licensing and performance on the commodity hardware that was used for the target system. As a consequence of these cost and performance pressures, a mechanism was added to the TG engine to allow its state to be checkpointed and subsequently be recovered.

By 1994, the TG engine had evolved into a construct that supported a multiplicity of contexts of computation to transition between a set of nodes that were read into the engine at run time. The contexts used an explicit set of name/value pairs to describe the state of the computation. This was inspired by Henry Baird's PIC file format – used in the Bayesian classifier and image processing the group was doing at the time. Also left behind was the notion of coded event routines that were particular to a specific workflow instance. A set of event actions coupled with name/value pair transformations took their place. At this point, the work-flow system was really a side-effect machine.

Contexts were created that represented jobs, which might create child contexts, or request external processes (Agents) to perform actions on behalf of the context. The result of the external actions manifested themselves as events that may move the context to a new place. External events could also add to the name/value pairs of the contexts, and the name/value pairs could be used to create outbound messages to the Agents. Child contexts could then rendezvous with their parent contexts, potentially updating the parent's name/value pair as a result. The structure of the Agents allowed the system to serialize or parallelize Agents depending upon existing conditions.

Additional Agents could be started or stopped as needed to handle the offered load without any changes to the running engine. As the engine discovered the new Agent resources, it started to use them. More importantly, the system was designed to deal with the failure of any Agent. If an operation queued to a specific Agent failed, the system marked the Agent charged with the failed operation as "down", so that no further operations would be sent to that Agent. The failed operation would be retried (restarted) on an Agent of the same Agent class as the original. If the failed Agent was the only resource, and with its failure, no further resources of that Agent class remained, then the context that initiated the operation would be suspended. Anytime a context was suspended due to insufficient Agent resources, the system scheduled a discovery process for the required Agent class resources.

10.2 Second evolution: media gateway controller

During 1999, work turned to creating a residential gateway (RG). This work was part of an effort to provide a triple-play (voice, data & video programming) fiber-to-the-home product line. Our team was responsible for the residential gateway, a customer-premises device that provided 4 lines of VoIP as well as 100 Mbps data. As part of the development process there was a need to perform unit and system testing. In the case of VoIP, the requirements called for documentation that the unit could support the overall availability and reliability requirements.

The VoIP call-control protocol specified was the Media Gateway Control Protocol (MGCP). Being a stimulus/response protocol, MGCP responded to commands from a controller, called a Call Agent, to perform each step needed to set up a call. This includes actions such as collecting digits, detecting the handset going off-hook, playing progress tones (dialtone, ringback), as well as the commanding of the endpoint to set up the proper data streams (correct voice encoding, rates, remote ports). Clearly any successful reliability and feature test of the residential gateway is predicated

upon a correctly functioning Call Agent.

While VoIP was not yet common, there were a few call agents that were available to the team. The first was a commercial version which, while expensive, was not able to run anything more than a basic MGCP call setup, and unreliably at that. The second call agent was from an internal source, but was extremely slow, so slow that a single user dialing a number could create events faster than the system could respond, resulting in missed digits. The last source was part of the target system (a product called iMerge), but in reality was a mix of a protocol translator and call agent, converting ISDN commands into a limited set of MGCP sequences. Using this latter device did not exercise the full range of MGCP capabilities in the RG, although fine for the immediate circumstance, a large limitation for the larger goal of creating a full featured MGCP endpoint.

At this point it was suggested that the event-driven workflow engine might be extended to serve as a Call Agent. The adaptation of the workflow engine to a new role as a call-control element included the ability of the engine to send and receive messages directly from an external element. The SEND_MESSAGE and PROCESS_MESSAGE methods perform the transformation between name/value pairs and formatted messages. As the name suggests, SEND_MESSAGE takes as input a name/value pair list, and a message description file (called a template file), producing a formatted message. Since there are a variety of protocols that may need to be supported, the SEND_MESSAGE method also uses the type of context when selecting the output transform, as well as the sending socket.

10.2.1 Adding Multi-protocol support

Soon after the RG effort started, the team was asked to add support for MEGACO/H.248 and SIP to the call-control stack. The customer wanted to be able to partition the 4 telephony ports in such a way that any of the supported three protocols could be run on each of the ports. In order to test the RG, the former workflow engine needed to support all three protocols, with some configurations requiring that the protocols should be active simultaneously.

Since the engine already supported the notion of a context type,

each of the distinct protocols could be associated with a different type. As part of the modifications to the engine (now just called the flow engine), was to add something called a "built-in" agent. As originally designed, the engine used dynamically found clusters of externally run programs called workflow agents or WfAgents as discussed above. Each call to a WfAgent involved a non-blocking remote procedure call. Even though the design of the system supported a large number of WfAgents to be in use simultaneously, the use of an WfAgent to send call control messages was too heavyweight, as in many cases the RPC message between the flow engine and the WfAgent could be many times the size of the desired control message. Rather, the concept of a service was extended to the notion of a "built-in" agent or service.

The built-in agent is, in effect, a server. A directive line defines what type of socket the engine should create, at what port the socket should be bound, the name of the service, and how incoming messages should be processed. This approach allows the flow engine to create an arbitrary set of servers, each bound to a different port, and named in such a way that could easily correspond to a "type" of context. Thus a SIP service could be defined with a single line in the flow graph definition file that would appear to the outside world to be a SIP server.

The first call-control protocol added to the flow engine was MGCP. This stimulus/response protocol is a textual line-oriented protocol. For this first protocol, a preliminary decoding routine was written as part of the engine code itself. The primary purpose of the primary decoding routine is to route the incoming message to the proper context and to assign a named event to the message. Complete parsing of the message is under control of the owning context, and is optionally done using parsing rules found at the current place of the context. This concept will be explained in more detail in a later section. As part of acceptance testing of the RG, a commercial analog line tester was used, a device aptly called a "Hammer". At one point the VoIP system of 24 lines was under continual test for 6 months with a success rate of over 99.9999%.

An outgrowth of the RG effort was the creation of a VoIP line card for the AnyMedia Line Access Gateway (LAG) product. During 2003

an increasing number of customers were migrating from traditional telephony to VoIP. The strategy for a number of operators was to provide identical telephony behavior for their customers, while moving from circuit switched equipment to VoIP LAGs at the central offices. Since the LAG''s VoIP feature card was derived from the RG, it was also capable of using all three VoIP protocols. At this point in the evolution of the telephony network, there was not a consensus on which VoIP protocol should be used. As a consequence there were requests from customers to trials and demonstrations of all three protocols.

In the majority of cases, the NGN softswitch element was not available due to reasons that included scheduling of existing systems, features not yet implemented, the specification of a competitor's softswitch, and the specification of a switch that did not yet exist. The requirements were provided by one of two means: a marked up standards documents, or by supplying the scenarios and messages traces that a conforming LAG must support. In other cases, a customer would specify a competitor's softswitch as the required controlling element. On more than one occasion, the customer specified a set of tests required to qualify the LAG, that no production softswitch could yet support.

The flow engine was used to pass the acceptance tests successfully. This included 24 busy-hour tests as well as functional tests. The landscape of telecommunications protocols included instances that correspond to both functional and stimulus/response models. At this time, there were three VoIP protocols that were the focus of our activity: MGCP, H.24/MEGACO and SIP. Since that time, functional models, including the Session Initiation Protocol (SIP) have come to dominate over stimulus / response protocols such as MEGACO/H.248 and Media Gateway Control Protocol (MGCP).

10.2.2 Standard yet Incompatible

Even though these protocols had been pronounced as “standards”, the reality was very different. Within all of these protocols there is the capability to use and abuse protocol elements to achieve a variety of user experiences.

Indeed, especially within the stimulus/response protocols, there

are a number of ways to create the same user experience while using a remarkably different call controller – media gateway message sequence. During the effort of supporting the LAG product, no two customers' sequences were identical, in some cases they used substantially different message exchanges to accomplish the same goal. Figure 10.1 and Figure 10.2 contrast alternatives valid ways that a Media Gateway Controller can command an endpoint to play dialtone and collect digits to set up a call. With each variation in protocol that was handed to the team, there was one constant: the short time frame required for delivery, while making sure that none of the previous call flows were invalidated. The combination of using a graph structure that is read in at startup time, with mechanisms that transform messages that are instantiated at first reference from files were well matched for the rapid prototyping needed to meet aggressive time lines. With the engine's capabilities, changes to both parsing of incoming messages and sending of outgoing messages could be changed without restarting the engine, greatly increasing the rate at which the system could be fine tuned for a new scenario. In many cases, systems that can be rapidly tuned for particular situations can easily become unstable, with changes to account for new requirements changing behavior for previously known good test cases. As an example, the LAG testing team tested a number of configurations simultaneously. At the height of AnyMedia LAG testing there were 2 variants of MGCP, 3 variants of H.248 and (at least) 4 variants of SIP under test at any time. Each was for a separate customer, with different customer key codes loaded into the LAG bank under test.

```
MEGACO/1 [216.33.33.61]: 27000
  Transaction = 1235 {
     Context = - {
        Modify = TermA {
           Signals {cg/dt},
           DigitMap= Dmap1 {(2XXX)}
           Events = 1112 {
              al/on, dd/ce {DigitMap=Dmap1}
           },
        }
     }
  }
```

Figure 10.2: A Valid Alternative - Conditioning an Endpoint

The flow engine used the same notion of customer key codes to allow multiple simultaneous dialects of the same protocol to co-exist. For instance, the various dialects could use the same well known port number (eg. 5060 for SIP, 2944 for H.248) but behave in the correct dialect based upon name/value pairs set in the context. The specific name/value pairs set by the customer key code lookup were then used to select a path through the graph that created appropriate behavior. With disjoint paths used to create specific behaviors, new features could be introduced without compromising the existing protocol definitions.

The creation of the flow engine, which has been referred to as the Lightweight Flow Engine (LiFE) interchangeably throughout this thesis arose from a set of requirements, first as a general coordination mechanism, and then as an element to test products. The use of a flow engine as a VoIP call control element for testing NGN LAGs resulted in the requirement to support multiple variations of multiple protocols, many times simultaneously, ideally sourced from the same physical computer.

The use of the Lightweight Flow Engine (LiFE) allowed all of the requirements to be met, with LiFE running on older commodity hardware executing a general purpose operation system (Linux).

10.2.3 Implementation Notes

The LiFE system is comprised of the flow engine, supporting executables and programs designed to permit management, diagnoses, and monitoring of the system. As this chapter has covered, the flow engine has been in use and evolving for a number of years. As one might expect there have been multiple contributors to the code during this time.

The “cloc” utility reports that the flow engine consists of 77.8 thousand lines of non-comment C language code. Richard Sun, formerly of Bell Labs, contributed 3.2 thousand lines located in the files supporting Q.931, IPDC and SS7 specific protocols. These modules have been used to control Media Gateways to provide connectivity using ISDN and SS7. I integrated Richard's modules into the flow engine, so that elements of the control packets could be accessed and manipulated by the name/value pair directives covered in Chapter 4. Kiem-Phong Vo (during his tenure at Bell Labs), wrote a memory allocator, *amalloc*, that I modified so that heap memory was saved in a file. Using this mechanism, a new instance of the flow engine could read the heap from a previous instance of the engine and rebuild internal state. Using Kiem Vo's code, I was able implement this function, and add monitors to memory usage that provides memory threshold alarming. Vo's allocator is 900 lines of code. Finally, Sape Mullender contributed 50 lines of code that is used for fast queue management in the engine. I would also like to acknowledge Sun Microsystems, who made their ONC/RPC code freely available. I rewrote and heavily modified the code to support asynchronous Remote Procedure Call (RPC) use, rather than using RPC calls synchronously. Early versions of the flow engine were single threaded, so no operation could block, as a result the existing RPC libraries were not suitable for use in the flow engine.

In addition to the flow engine itself, several key libraries, specific to the flow engine are used. In particular the name/value pair library is a key part of the engine's design. The first version was contributed by Troy Cauble of Bell Labs. In addition, he wrote the External Data Representation (XDR) routines that encoded and decoded data structures for sending over the a communications channel. While Cauble's XDR routines are still used, I needed to rewrite the original

name/value pair routines , to make the libraries thread safe and to reduce the number of malloc and frees performed during the lifetime of a name/value pair list. However, Cauble contributed the code for the Work Flow Agent executable and libraries used by the flow engine to communicate to the Agents. In all Troy contributed 1600 lines of code to the system. Another 100 lines of code implemented the hash functions, Murmur3 hash (Austin Appleby) and newhash (Robert Jenkins).

Including the libraries, LiFE consists of 82.7 thousand lines of code of which 5.9 thousand lines of code were written by other people than the author, leaving the author with responsibility for 93% of the code. Although the code contribution seems small based upon the numbers, the contributions were essential to the effort.

The monitoring applications were built upon Tcl/Tk. In addition, there is a key graphing function that is used to depict the current flow graph. This graphical extension to Tcl/Tk is called tkdot. It comes from an effort to integrate the dot libraries to Tk. Dot and tkdot was written by Stephen North and Eleftherios Koutsofios (both at Bell Labs at that time), with code contributed by Kiem-Phong Vo. My contribution was to write Tcl/Tk code to use these libraries, add code to interface to ONC/RPC and write the monitoring, management and diagnostic functions for the flow engine in Tcl/Tk.

11 Conclusions

A number of aspects of the Lightweight Flow Engine, LiFE, have been presented during the course of these chapters. Based upon a state machine paradigm, with language and system features that support the creation and execution of sophisticated behaviors.

During the progression of this thesis the discussion information has been presented to illustrate how the design and implementation of the flow engine addresses the requirements listed in the problem statement from Chapter 1. Specifically:

i. A formalized model of the flow engine has been presented, along with an example in Chapter 4 highlighting the expressibility of the computational model implemented by LiFE. This along with the discussion in Chapter 4 shows that requirement (i.) of the problem statement has been achieved;

ii. Chapter 8 introduced some of the unique system capabilities of the engine, including the patented "Update In Place" feature. Chapter 9 shows that the flow-engine framework combines the flexibility described in earlier chapters with a performance level that is consistent with, or higher than, systems that are purpose built for a specific protocol or function, thus satisfying our second Problem Statement goal;

iii. Language additions to the core state-machine model were shown that extend the flow engine in a manner to make it Turing Complete. The examples of flow graph constructs: states, arcs, actions and name/value pair transformations

demonstrate that the computational abstraction implemented by the flow engine corresponds closely to protocols expressed as either state diagrams (Chapter 4) or message chart diagrams (Chapter 9), and so supports the goal (iii.) listed of the problem statement;

iv. The engine is efficient enough to allow the system to solve real problems as an essential part of real products. The inclusion of checkpointing in the engine framework, as well as the ability to update the computation while running, provides multiple ways to achieve a highly available solution which preserves the state of the computation. The combination of these mechanisms, realized outside of the service logic satisfying goal (iv.) from the problem statement;

v. Examples were presented in Chapter 7 to illustrate that the flow engine satisfies problem statement goal (v.), that is that the flow engine is not only capable of solving real-world problems, but has been used to tackle disparate data-processing projects as well;

vi. Through the course of this thesis, I have presented a system designed to be easily modified and showed that the computation can be modified even as it is running. The system uses a concise, human readable form of the computation in the form of a flow graph. No special authoring tools are necessary, a simple text editor will do, achieving goal (vi.) from the problem statement goal. At the same time, the structure of the graph does not preclude the use of add-on tools for additional checking, such as reachability.

During the course of this thesis, LiFE, a flexible, expressive execution framework has been described. The system has been shown to be efficient enough to execute a wide range of functions on off-the-shelf commodity hardware, while supporting a computational model which corresponds well to the ways many communication-centric problems are described.

Future Work

There are several avenues that could be explored building upon the

existing flow engine.

One is to improve the syntax of the graph language. Use of the existing language results in programs that expose the state machine structure very clearly, and locate the logic with trigger events. This is helped by the fairly compact representation of both the logic and the state-machine aspects of the graph. There is an XML version of the graph flow language, one that was created to match the capabilities of the existing language. The XML graph flow language did not have the compact representation of the original, so much so, that it became cumbersome to edit the XML graph flow language directly. An authoring tool would be in order, as is the case for UML tools. In place of XML, JavaScript Object Notation [19] JSON may represent a syntax that could be both concise and standards based. Once a JSON syntax is defined, the audience for the flow engine could be expanded, opening the possibility for the creation of a suite of tools to handle verification, reachability and liveness of a state graph.

The flow engine uses name/value pairs to define the state of a computation. The current design uses hashed internal name/value pair lists. A possible extension to the engine would be to allow the engine to use a NoSQL data store, such as Cassandra [42] or Riak[25], as the storage for the system name/value pairs. These NoSQL solutions provide scalability, and their own reliability mechanisms. One could image a flow-engine context storing its name/value pairs in something like Riak when a context transitions to a quiescent state. In this manner the flow engine could evolve to a stateless worker in which contexts are created on demand at specific states in a flow graph to continue with a computation.

The measurements contained in Chapter 9 point out the relative efficiency that results from system architecture design trade offs. Currently there is a lot of activity in the area of Software Defined Networks [59]. Could the application of an SDN to a cluster of more efficient flow engine implementations provide a combination of scalability and efficiency that is more attractive than a single system that is designed to spread out over multiple processors?

The power and flexibility of the flow engine makes it an attractive

25 Basho Technologies, “Riak”, http://docs.basho.com/riak/latest (Accessed June 2013)

platform to explore these questions. As computer systems continue to increase in power, through the addition of multiple cores, and the cost of storage continues to plummet, the challenge for programmers is to make the best use of this increasingly complex and powerful hardware. While LiFE is not the only answer, it does provide one way to harness a range of computational resources to solve real world problems in a rapid and efficient manner.

12 Bibliography

[1] Aho, A. V., Kernighan, B. W., & Weinberger, P. J. (1987). *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc..

[2] Aho, A. V., & Corasick, M. J. (1975). Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, *18*(6), 333-340.

[3] Al Bahra, S. (2013). Nonblocking algorithms and scalable multicore programming. *Communications of the ACM*, *56*(7), 50-61.

[4] Alur R. & and Dill, D.L. , "A Theory of Timed Automata," Theoret. Comput. Sci., 126:2 (1994), 183–235.

[5] Alur R. & Madhusudan P., "Decision Problems for Timed Automata: A Survey," Proc. 4th Internat. School on Formal Methods for Design of Comput., Commun. and Software Systems: Real Time (SFM-RT '04) (Bertinoro, It., 2004), pp. 1–24.

[6] Andreasen, F., & Foster, B. (2003). *Media gateway control protocol (MGCP) version 1.0*. RFC 3435, January.

[7] Auburn, R. J., Barnett, J., Bodell, M., & Raman, T. V. (2005). State Chart XML (SCXML): State Machine Notation for Control Abstraction 1.0. *W3C Working Draft*, *5*.

[8] Auburn, R. J., Cafarella, M., Jackson, D., Peck, J., Sharma, P., Shanmughan, S., ... & Zhang, Y. (2005). Voice browser call

control: CCXML version 1.0. *W3C Working Draft*.

[9] Berry G., "Real Time Programming: Special Purpose or General Purpose Languages," Information Processing 89 (G. X. Ritter, ed.), Elsevier Science, North-Holland, Amsterdam, 1989, pp. 11–18.

[10] Berry G. & Gonthier G., "The ESTEREL Synchronous Programming Language: Design, Semantics, and Implementation," Sci. Comput. Programming, 19:2 (1992), 87–152.

[11] Berry G. & Sethi R., "From Regular Expressions to Deterministic Automata," Theoret. Comput. Sci., 48:1 (1986), 117–126.

[12] Birrell, A. D., & Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, *2*(1), 39-59.

[13] Bortzmeyer, S. (2006). Cosmogol: a language to describe finite state machines.

[14] Brand D. & Zafiropulo P., "On Communicating Finite-State Machines", Journal of the ACM, Vol 30, No.2 April 1983,323-342

[15] Brzozowski J. A. "Derivatives of Regular Expressions," J. ACM, 11:4 (1964), 481–494.

[16] Budkowski, Stanislaw, & Piotr Dembinski. "An introduction to Estelle: a specification language for distributed systems." *Computer Networks and ISDN systems* 14.1 (1987): 3-23

[17] Cohen, D. I., & Chibnik, M. (1991). *Introduction to computer theory*. Wiley. (pp. 86-92)

[18] Costello, A. M., & Varghese, G. (1998). Redesigning the BSD timer facilities. *Software: Practice and Experience*, *28*(8), 883-896.

[19] Crockford, D. (2006). JavaScript Object Notation (RFC 4627).

[20] Dabek, F., Zeldovich, N., Kaashoek, F., Mazieres, D., & Morris, R. (2002, July). Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop* (pp. 186-189). ACM.

[21] de Melo, A. C. (2010, September). The new linux'perf'tools. In *Slides from Linux Kongress*.

[22] Dennis, J. B. (1980). Data flow supercomputers. *Computer*, *13*(11), 48-56.

[23] Forman, G. H., & Zahorjan, J. (1994). The challenges of mobile computing. *Computer*, *27*(4), 38-47.
[24] Fowler, M. (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Languange*. Addison-Wesley Professional.
[25] Garcia-Martin, M. (2005). Input 3rd-generation partnership project (3GPP) release 5 requirements on the session initiation protocol (SIP).
[26] Gelernter, D., & Carriero, N. (1992). Coordination languages and their significance. *Communications of the ACM*, *35*(2), 96.
[27] Goncalves, F. E. (2010). *Building Telephony Systems with OpenSIPS 1.6*. PACKT publishing.
[28] Gosling, J., Joy, B., Steele, G., & Bracha, G. (1996). The Java Language Specification. *Sun Microsystems, Inc*, *2550*, 94042-1100.
[29] Groves, C., Pantaleo, M., Anderson, T., & Taylor, T. (2003). The Megaco/H. 248 Gateway Control Protocol, version 2. *IETF, Apr*.
[30] Halbwachs N, Caspi P., Raymond P. & Pilaud D., "The Synchronous Data Flow Programming Language LUSTRE," Proc. IEEE, 79:9 (1991), 1305–1320.
[31] Harel D., "Statecharts: A Visual Formalism for Complex Systems," Sci. Comput. Programming, 8:3 (1987), 231–274.
[32] Harris, C. (2001). JAIN SIP Release 1.0 Specification. *Sun Microsystems*.
[33] Hennessy M.., Algebraic Theory of Processes, MIT Press, Cambridge, MA, 1988
[34] Herlihy, M. (1991). Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, *13*(1), 124-149.
[35] Holzmann G. J., The SPIN Model Checker: Primer and Reference Manual, Addison-Wesley, Boston, MA, 2003.
[36] Hopcroft J. E. & Ullman J. D. , Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Reading, MA, 1979.
[37] Jantsch, A. (2004). *Modeling embedded systems and SoCs: concurrency and time in models of computation*. Morgan Kaufmann.
[38] Jiang, H., Iyengar, A., Nahum, E., Segmuller, W., Tantawi, A., & Wright, C. P. (2009, April). Load balancing for SIP server clusters. In *INFOCOM 2009, IEEE* (pp. 2286-2294). IEEE.
[39] Jones, M. P. (1993, July). A system of constructor classes:

overloading and implicit higher-order polymorphism. In *Proceedings of the conference on Functional programming languages and computer architecture* (pp. 52-61). ACM.

[40] Junczys-Dowmunt, M. (2012). A Phrase Table without Phrases: Rank Encoding for Better Phrase Table Compression. In *Proceedings of the 16th Annual Conference of the European Association for Machine Translation* (pp. 245-252).

[41] Kaliski Jr, B. S., & Redwood City, C. A. (1993). A Layman's Guide to a Subset of ASN. 1, BER, and DER.

[42] Lakshman, A., & Malik, P. (2010). Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, *44*(2), 35-40.

[43] Le Guernic P., Gautier T., Le Borgne M., & Le Maire C., "Programming Real-Time Applications with SIGNAL," Proc. IEEE, 79:9 (1991), 1321–1336.

[44] Leymann, F. (2001). Web services flow language (wsfl 1.0).

[45] Linton, M. A. (1990, June). The Evolution of Dbx. In *USENIX Summer* (pp. 211-220).

[46] Martin, D., & Estrin, G. (1967). Models of computations and systems—evaluation of vertex probabilities in graph models of computations. *Journal of the ACM (JACM)*, *14*(2), 281-299.

[47] McCann P. J. & Chandra S., "Packet Types: Abstract Specification of Network Protocol Messages," SIGCOMM Comput. Commun. Rev., 30:4 (2000), 321–333.

[48] Milner R., Communication and Concurrency, Prentice-Hall, Englewood Cliffs, NJ, 1989.

[49] Milner R., Parrow J., & Walker D., "A Calculus of Mobile Processes, Part I," Inform. and Comput., 100:1 (1992), 1–40.

[50] Milner R., Parrow J. & Walker D., "A Calculus of Mobile Processes, Part II," Inform. and Comput., 100:1 (1992), 41–77.

[51] Morris, R. J. T. (1990). Q+: AT&T's Queueing+ Analysis Software Version 1.0 User Guide and Reference Manual. *Holmdel, NJ: Bell Laboratories*.

[52] Myung, H. G. (2008). Technical overview of 3GPP LTE. *Polytechnic University of New York*.

[53] O'Doherty, P., & Ranganathan, M. (2003). JAIN SIP Tutorial: Serving the developer community. *Sun Microsystems*.

[54] Odifreddi P., Classical Recursion Theory, Studies in Logic and the Foundations of Mathematics, Vol. 125, North-Holland, Amsterdam, 1992.
[55] Ong, L., & Yoakum, J. (2002). An introduction to the stream control transmission protocol (SCTP).
[56] Org, S. (2008). SIPconnect compliance Program.
[57] Petri, C.A., Kommunikation mit Automaten. Schriften des Rheinisch-Westfälischen. Institutes für instrumentelle Mathematik an der Universität Bonn Nr (1962)
[58] Recommendation T.30, *Standardization of Group 3 Facsimile Apparatus of Document Transmission (2005)*
[59] Reitblatt, M., Foster, N., Rexford, J., & Walker, D. (2011, November). *Consistent updates for software-defined networks: Change you can believe in!. In Proceedings of the 10th ACM Workshop on Hot Topics in Networks (p. 7). ACM.*
[60] Ritchie, D. M., Johnson, S. C., Lesk, M. E., & Kernighan, B. W. (1978). *The C programming language. Bell Sys. Tech. J, 57, 1991-2019.*
[61] Romellini, C., & Tonelli, F. (2005). CCXML: The Power of Standardization. *Loquendo, Sep*, *27*, 7.
[62] Rose, M. T., & Cass, D. E. (1987). ISO Transport Service on top of the TCP Version: 3.
[63] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., ... & Schooler, E. (2002). *SIP: session initiation protocol* (Vol. 23). RFC 3261, Internet Engineering Task Force.
[64] Schulzrinne, H. (1997). RTP payload for DTMF digits. In *Internet Draft, Internet Engineering Task Force*.
[65] Shannon, C. E. (1956). A universal Turing machine with two internal states. *Automata studies*, *34*, 157-165.
[66] Sharp, R. (2004). *Higher-level hardware synthesis* (Vol. 2963). Springer.
[67] Srinivasan, R. (1995). RPC: Remote procedure call protocol specification version 2.
[68] Srinivasan, R. (1995). XDR: External data representation standard.
[69] Tanenbaum, A. S. (1988). *Computer Networks, 2nd*. Prentice-Hall,

[70] Thompson, K., & Ritchie, D. M. (1975). *UNIX Programmer's Manual*. Bell Telephone Laboratories.

[71] Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society*, *42*(2), 230-265.

[72] Whiting P. G. & Pascoe R.S.V., "A History of Data-Flow Languages," IEEE Ann. Hist. Comput., 16:4 (1994), 38–59.

[73] Winterbottom, P. (1994, January). ACID: A Debugger Built From A Language. In *USENIX Winter* (pp. 211-222).

[74] Wolfinger, R., Reiter, S., Dhungana, D., Grunbacher, P., & Prahofer, H. (2008, February). Supporting runtime system adaptation through product line engineering and plug-in techniques. In *Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on* (pp. 21-30). IEEE.

[75] Xiaotao Wu & Schulzrinne, H."Programmable end system services using SIP", (2003). *ICC '03. IEEE International Conference on Communications (Volume 2)*

A US Patent 20090089770

Method and Apparatus for Performing Non Service Affecting Software Upgrades in Place

Shamilian; John H.; (Tinton Falls, NJ) ; Wood; Thomas L.; (Colts Neck, NJ), April 2, 2009

A.1 Abstract

The invention includes a method and apparatus for dynamically defining and instantiating an undefined portion of a graph, where the graph has a plurality of states and a plurality of state transitions. A method includes executing the graph where the graph comprises a defined portion and an undefined portion and a plurality of tokens traverse the graph executing functions, suspending the one of the tokens in response to the one of the tokens detecting the undefined portion of the graph, generating a new portion of the graph for the undefined portion of the graph, replacing the undefined portion of the graph with the new portion of the graph, and releasing the suspended token. The new portion of the graph is generated by generating at least one definition file for the undefined portion of the

graph and executing the at least one definition file to form thereby the new portion of the graph. The at least one definition file is generated by obtaining information adapted for defining the undefined portion of the graph and generating the at least one definition file using the obtained information.

A.2 Claims

1. A method for dynamically instantiating a portion of a graph having a plurality of states and a plurality of state transitions, comprising: executing the graph, wherein the graph comprises a defined portion and an undefined portion, wherein a plurality of tokens traverse the graph executing functions; in response to one of the tokens detecting the undefined portion of the graph, suspending the one of the tokens that detected the undefined portion of the graph; generating a new portion of the graph for the undefined portion of the graph; replacing the undefined portion of the graph with the new portion of the graph; and releasing the suspended token.
2. The method of claim 1, wherein generating the new portion of the graph comprises: generating at least one definition file for the undefined portion of the graph; and executing the at least one definition file to form thereby the new portion of the graph.
3. The method of claim 2, wherein generating the at least one definition file for the undefined portion of the graph comprises: obtaining information adapted for defining the undefined portion of the graph; and generating the at least one definition file using the obtained information.
4. The method of claim 3, wherein the information is obtained at least one of locally from a system on which the graph is instantiated and remotely from at least one network element in communication with the system on which the graph is instantiated.
5. The method of claim 1, wherein replacing the undefined

portion of the graph with the new portion of the graph comprises: reading the new portion of the graph into the graph.

6. The method of claim 1, wherein, upon being released, the released token traverses the new portion of the graph.
7. The method of claim 1, wherein at least one other token traverses the defined portion of the graph at least while the new portion of the graph is defined.
8. An apparatus for dynamically instantiating a portion of a graph having a plurality of states and a plurality of state transitions, comprising: means for executing the graph, wherein the graph comprises a defined portion and an undefined portion, wherein a plurality of tokens traverse the graph executing functions; means for suspending one of the tokens in response to the one of the tokens detecting the undefined portion of the graph; means for generating a new portion of the graph for the undefined portion of the graph; means for replacing the undefined portion of the graph with the new portion of the graph; and means for releasing the suspended token.
9. The apparatus of claim 8, wherein the means for generating the new portion of the graph comprises: means for generating at least one definition file for the undefined portion of the graph; and means for executing the at least one definition file to form thereby the new portion of the graph.
10. The apparatus of claim 9, wherein the means for generating the at least one definition file for the undefined portion of the graph comprises: means for obtaining information adapted for defining the undefined portion of the graph; and means for generating the at least one definition file using the obtained information.
11. The apparatus of claim 10, wherein the information is obtained at least one of locally from a system on which the graph is instantiated and remotely from at least one network element in communication with the system on which the graph is instantiated.
12. The apparatus of claim 8, wherein the means for replacing the

undefined portion of the graph with the new portion of the graph comprises: means for reading the new portion of the graph into the graph.

13. The apparatus of claim 8, wherein, upon being released, the released token traverses the new portion of the graph.
14. The apparatus of claim 8, wherein at least one other token traverses the defined portion of the graph at least while the new portion of the graph is defined.
15. A computer-readable medium storing a software program that, when executed by a computer, causes the computer to perform a method for dynamically instantiating a portion of a graph having a plurality of states and a plurality of state transitions, comprising: executing the graph, wherein the graph comprises a defined portion and an undefined portion, wherein a plurality of tokens traverse the graph executing functions; in response to one of the tokens detecting the undefined portion of the graph, suspending the one of the tokens that detected the undefined portion of the graph; generating a new portion of the graph for the undefined portion of the graph; replacing the undefined portion of the graph with the new portion of the graph; and releasing the suspended token.
16. The computer-readable medium of claim 15, wherein generating the new portion of the graph comprises: generating at least one definition file for the undefined portion of the graph; and executing the at least one definition file to form thereby the new portion of the graph.
17. The computer-readable medium of claim 16, wherein generating the at least one definition file for the undefined portion of the graph comprises: obtaining information adapted for defining the undefined portion of the graph; and generating the at least one definition file using the obtained information.
18. The computer-readable medium of claim 15, wherein replacing the undefined portion of the graph with the new portion of the graph comprises: reading the new portion of the graph into the graph.

19. The computer-readable medium of claim 15, wherein, upon being released, the released token traverses the new portion of the graph.
20. The computer-readable medium of claim 15, wherein at least one other token traverses the defined portion of the graph at least while the new portion of the graph is defined.

A.3 Description

A.3.1 Field of the Invention

The invention relates to the field of software applications and, more specifically, to performing upgrades of software applications.

A.3.2 Background of the Invention

Software applications, such as Voice-Over-IP (VoIP) call control as performed by a VoIP Media Gateway Controller (MGC), have been realized using an approach in which a high-level language is used to describe the data processing and message exchange behavior of the VoIP MGC. In a VoIP MGC, an execution engine accepts a high-level graph language file that is used by the execution engine to create internal data structures which realize the behaviors described in the input language file. The combination of the execution engine and the high-level graph language file (which describes the VoIP MGC behavior) results in a system that exhibits the desired VoIP MGC processing characteristics.

In many such software applications, the software application must be fully defined and implemented before the software application is deployed to the field. For example, each of the functions to be performed by the software application must be fully defined and implemented before deploying the software application to the field. Disadvantageously, this requirement often results in delayed deployment of software applications because additional time is required to define and implement the numerous functions of the software application, thereby resulting in delayed availability of

functions and services for use by customers.

A.3.3 Summary of the Invention

Various deficiencies in the prior art are addressed through the invention of a method and apparatus for dynamically defining and instantiating an undefined portion of a graph, where the graph has a plurality of states and a plurality of state transitions. A method includes executing the graph where the graph comprises a defined portion and an undefined portion and a plurality of tokens traverse the graph executing functions, suspending the one of the tokens in response to the one of the tokens detecting the undefined portion of the graph, generating a new portion of the graph for the undefined portion of the graph, replacing the undefined portion of the graph with the new portion of the graph, and releasing the suspended token. The new portion of the graph is generated by generating at least one definition file for the undefined portion of the graph and executing the at least one definition file to form thereby the new portion of the graph. The at least one definition file is generated by obtaining information adapted for defining the undefined portion of the graph and generating the at least one definition file using the obtained information.

A.3.4 Brief Description of the Drawings

The teachings of the present invention can be readily understood by considering the following detailed description in conjunction with the accompanying drawings, in which:

Figure A.1 depicts a high-level block diagram of a communication network;

Figure A.2 depicts a high-level block diagram of interaction between the call control element and border element of the communication network of Figure A.1;

Figure A.3 depicts the call flow graph depicted and described with respect to Figure A.2;

Figure A.4 depicts an exemplary graph function;

Figure A.5 depicts a graph including defined portions and undefined portions;

Figure A.6 depicts a method according to one embodiment of the

present invention; and

Figure A.7 depicts a high-level block diagram of a general-purpose computer suitable for use in performing the functions described herein.

To facilitate understanding, identical reference numerals have been used, where possible, to designate identical elements that are common to the figures.

A.3.5 Detailed Description of the Invention

The present invention enables undefined portions of software to be defined and instantiated, in place, without requiring the software to be stopped and restarted, thereby enabling existing, defined portions of software that are not being updated to continue to function while the undefined portions of software are being defined and instantiated. The present invention enables an undefined portion of an instantiated graph to be dynamically defined and instantiated while existing defined portions of the graph continue running. Although primarily depicted and described herein within the context of updating a VoIP call control element software application, the present invention is applicable to any software constructed using a graph language.

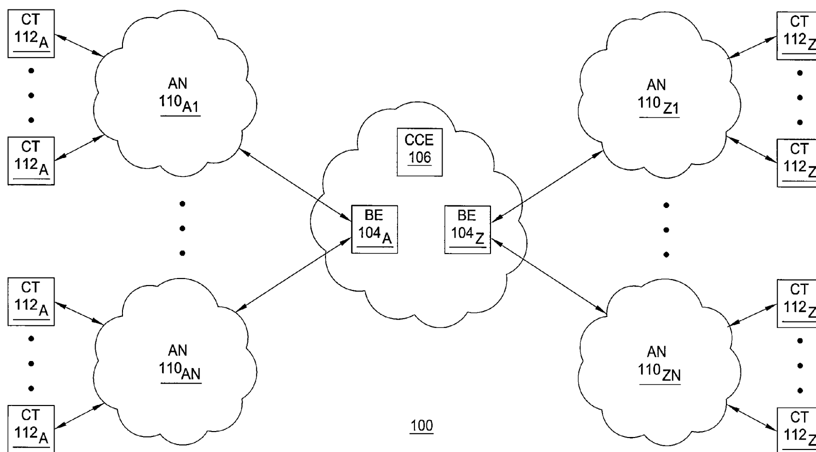


Figure A.1

Figure A.1 depicts a high-level block diagram of a communication network. Specifically, communication network 100 of Figure A.1 includes a core VoIP network 102, a plurality of access networks (ANs) 110.sub.A1-110.sub.AN and 110.sub.Z1-110.sub.ZN (collectively, ANs 110), and a plurality of customer terminals (CTs) 112.sub.A and 112.sub.z (collectively, CTs 112). The core VoIP network 102 includes a plurality of border elements (BEs) 104.sub.A and 104.sub.Z (collectively, BEs 104) and a call control element (CCE) 106. The core VoIP network 102 (including BEs 104 and CCE 106) and ANs 110 cooperate to provide end-to-end VoIP services to CTs 112. The core IP network 102 communicates with ANs 110 (via BEs 104) using various communication paths. The ANs 110 communicate with CTs 112 using various communication paths.

The core VoIP network 102 supports VoIP services. The BEs 104, which reside at the edge of core VoIP network 102, interface with CTs 112 over ANs 110. The BEs 104 perform signaling, call admission control, media control, security, and like functions, as well as various combinations thereof. For example, BEs 104 may be Media Gateways, VoIP Gateways, and the like, as well as various combinations thereof. The CCE 106 resides within core VoIP network 102. The CCE 106 interacts with BEs 104, as well as other components of core IP network 102, to provide VoIP-related functions and services, e.g., performing network wide call-control related functions as well as interacting with VoIP service related servers (e.g., media servers, application servers, and the like) as necessary. For example, CCE 106 may be a Media Gateway Controller (MGC), a softswitch, and the like (and may also be referred to as a call agent). The CCE 106 and BEs 104 may communicate using various protocols, such as Media Gateway Control Protocol (MGCP), Simple Gateway Control Protocol (SGCP), Megago (i.e., H.248), Session Initiation Protocol (SIP), H.323, and the like. Although primarily depicted and described herein with respect to a specific core network, the present invention is independent of the type of core network.

The ANs 110 may be packet-based ANs, TDM-based ANs, and the like, as well as various combinations thereof. A TDM-based access network may be used to support TDM-based customer terminals, such as customer endpoint devices connected via traditional

telephone lines. For example, a TDM-based access network may be the Public Switched Telephone Network (PSTN). A packet-based access network may be used to support IP-based customer endpoint devices and/or TDM-based customer endpoint devices. For example, a packet-based access network may be based on one or more of IP, Ethernet, Frame Relay (FR), ATM, and the like, as well as various combinations thereof. A packet-based access network (e.g., such as a cable network, Digital Subscriber Line (DSL) network, and the like) may include a terminal adapter (TA) such that the packet-based access network can support TDM-based customer endpoint devices. Although primarily depicted and described herein with respect to specific types of access networks, the present invention is independent of the type of access network.

The CTs 112 may be packet-based CTs, TDM-based CTs, and the like, as well as various combinations thereof. The CTs 112 may be customer network elements and/or customer endpoint devices. With respect to packet-based CTs, for example, CTs 112 may include customer network elements, such as VoIP gateways, VoIP routers, IP-PBXs, and like customer network elements supporting IP-based customer endpoint devices such as computers, IP phones, and the like, as well as various combinations thereof. With respect to TDM-based CTs, for example, CTs 112 may include TDM-PBXs, TAs (which provide interworking functions between TDM-based customer endpoint devices and the core IP network), and like customer network elements supporting TDM-based customer endpoint devices such as analog phones. Although primarily depicted and described herein with respect to specific types of customer terminals, the present invention is independent of the type of customer terminals supported.

Although omitted for purposes of clarity, core VoIP network 102 may include various other VoIP components, such as core elements (CEs), Media Servers (MSs), VoIP-related Application Servers (ASs), and the like, as well as various combinations thereof. Although omitted for purposes of clarity, each of the ANs 110 may include various components depending on the type of access network and the type of customer terminals being supported (e.g., such as IP-PBXs, TAs, TDM-PBXs, routers, and the like, as well as various

combinations thereof). In other words, communication network 100 is merely provided for purposes of describing the present invention within the context of one of many possible applications of the present invention. The present invention may be used to provide software upgrades in place for any software capable of being controlled in accordance with the present invention.

As described herein, communication network 100 enables VoIP calls to be established between CTs 112. A call scenario is now described In order to illustrate the operation of communication network 100 in completing a VoIP call between CTs (e.g., one of the CTs 112.sub.A and one of the CTs 112.sub.z). A customer using CT 112.sub.A (calling party) initiates a call to a customer using CT 112.sub.z (called party). A call setup signaling message is sent from CT 112.sub.A to BE 104.sub.A, which sends a call setup signaling message to CCE 106. The CCE 106 examines the called party information and queries one or more VoIP service ASs to obtain information necessary to complete the call. Upon determining that BE 104.sub.Z must be involved in completing the call, CCE 106 sends another call setup signaling message to BE 104.sub.z, which forwards the call signaling setup message to CT 112.sub.z. If the call is accepted by the called party, a call acknowledgement signaling message is sent from CT 112.sub.z to CCE 106 via BE 104.sub.z. Upon receiving the call acknowledgement message, CE 106 sends a call acknowledgement signaling message toward the calling party and, further, provides information associated with the call to BEs 104.sub.A and 104.sub.z (so that the call data exchange proceeds directly between BEs 104.sub.A and 104.sub.z).

As such, CCE 106 supports call flows adapted to handle signaling required to complete calls between CTs 112. The call flows supported by CCE 106 are provided using call flow software implemented on CCE 106. As networks and services evolve, the set of call flows that must be supported by CCE 106 changes, requiring updates to various portions of the software implemented on CCE 106. Disadvantageously, in existing networks, updates to portions of the software implemented on CCE 106 require all software implemented on CCE 106 to be stopped and restarted, thus requiring the network operator to either (1) accept network downtime during the software

upgrade or (2) switch service from CCE 106 to a redundant CCE (omitted for purposes of clarity) in order to continue to provide call control functions during the software upgrade. The present invention advantageously enables portions of the software implemented on CCE 106 to be upgraded in place, thereby enabling unaffected portions of the software to continue to run on CCE 106 during the software upgrade. The operation of the present invention in performing software upgrades in place may be better understood with respect to FIGs. 2-8.

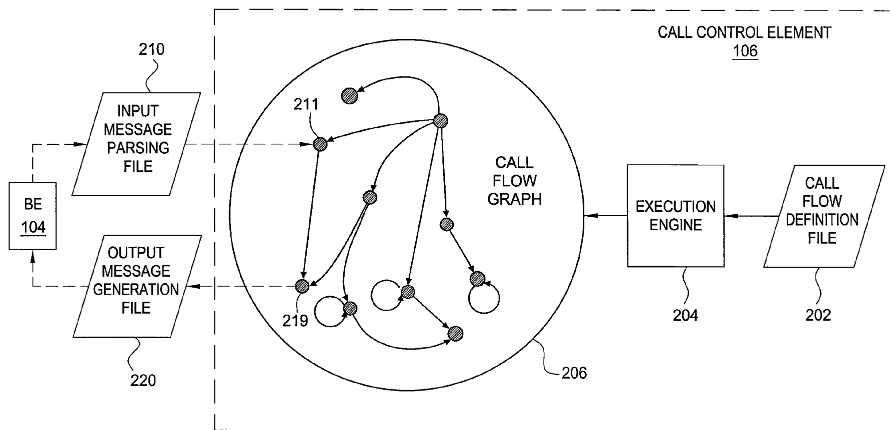


Figure A.2:

Figure A.2 depicts a high-level block diagram of interaction between the call control element and border element of the communication network of Figure A.1. As depicted in Figure A.2, a call flow definition file 202 is read into an execution engine 204 producing a call flow graph 206 (which may be more generally referred to herein as a call flow). The call flow definition file 202 explicitly defines the call flow in terms of states, state transitions, message translations, and the like. The execution engine 204 reads in the call flow definition file 202, processing the call flow definition file to produce call flow graph 206. The call flow graph 206 is software, including internal data structures, that runs on CCE 106 to realize the behaviors (e.g., functions and services), as described in call flow definition file 202, that need to be supported by CCE 106.

The execution engine 204 is adapted to perform various functions of the present invention such that an undefined portion of an instantiated graph may be detected and, upon being detected, may be defined and instantiated within the graph while the existing defined portions of the instantiated graph continue running. In other words, execution engine 204 is adapted to enable software to be defined and instantiated within a functioning system on-the-fly (i.e., while the other portions of the software that were previously defined and instantiated continue running to provide various functions. The execution engine 204 is adapted to define an undefined portion of a graph, to form thereby a new portion of the graph, and to replace the undefined portion of the graph with the new portion of the graph while the previously defined portions of the continue running. The execution engine 204 is adapted to perform various other functions associated with the present invention.

As depicted in Figure A.2, which depicts an example of a function that may be provided by CCE 106 using call flow 206, CCE 106 receives an input message parsing file 210 from BE 104 (e.g., as part of a VoIP control message). The CCE 106 processes the input message parsing file 210 using call flow graph 206. Specifically, input message parsing file 210 initializes a portion of the call flow graph 206 to an entry state (illustratively, state 211), from which call flow graph 206 transitions to an exit state (illustratively, state 219), thereby causing CCE 106 to generate an output message generation file 220 in response to the input message parsing file 210. The CCE 106 transmits the output message generation file 220 to BE 104 (e.g., as part of a VoIP control message). Thus, call flow graph 206 instantiated by CCE 106 using call flow definition file 202 enables CCE 106 to provide various functions and services. The operation of call flow graph 206 may be better understood with respect to Figure A.3. The operation of execution engine 204 in defining and instantiating an undefined portion of a graph may be better understood with respect to Figure A.4 – Figure A.6.

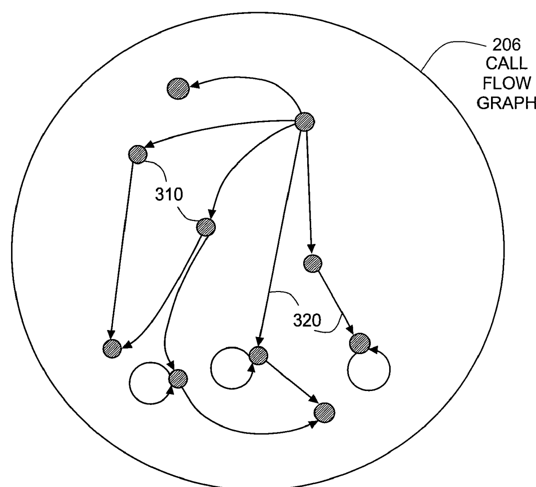


Figure A.3:

Figure A.3 depicts the call flow graph depicted and described with respect to Figure A.2. Specifically, call flow graph 206 is a graphical representation of call flow software. The call flow graph 206 includes a plurality of states 310 (collectively, states 310) and a plurality of state transitions 320 (collectively, state transitions 320). A multiplicity of tokens circulates within call flow graph 206, executing one or more functions during each of the state transitions 320 between states 310. For example, transition functions may include functions such as parsing incoming messages (e.g., from external devices, from other tokens, and the like), sending outgoing messages (e.g., to external devices, to other tokens, and the like), and the like, as well as various combinations thereof.

A graph may include one or more graph regions. A graph region is a set of one or more graph functions. A graph function is a function invoked at an entry state (e.g., using a "graph call" function), in which control is passed back to the entry state, rather than to an arbitrary next state, such that there is a set of states that are partitionable from other graph functions of the graph (i.e., partitionable starting with the entry state and including all states up to and including the state that uses a "graph return" to send control back to the entry state). For

purposes of clarity, the state transitions from return states to entry states (i.e., graph returns) are omitted from Figure A.3.

As such, a graph function includes an entry state and a returning state (which, for some functions, may be the same state as the entry state), and, optionally, one or more intermediate states. In other words, a graph function includes one or more states (i.e., a graph function may, in some instances, consist of a single state). The operation of a graph function may be better understood with respect to Figure A.4, which depicts an exemplary graph function which may be representative of any graph function of the call flow graph of Figure A.3 or any other graph composed of graph functions.

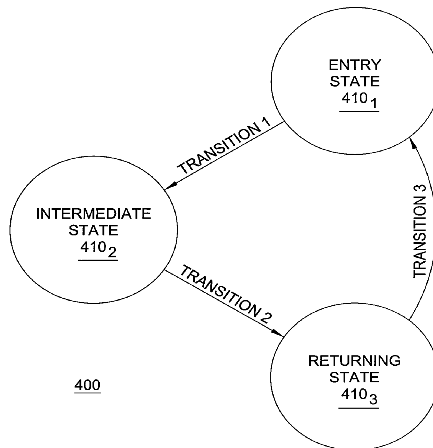


Figure A.4:

Figure A.4 depicts an exemplary graph function. As depicted in Figure A.4, graph function 400 of Figure A.4 includes an entry state 410.sub.1, an intermediate state 410.sub.2, and a returning state 410.sub.3. The graph function 400 is entered at entry state 410.sub.1. The "graph call" method invoked at entry state 410.sub.1 pushes its own state number onto the graph stack (for use later by the "graph return" method invoked by the returning state 410.sub.3). As graph function 400 executes, graph function 400 transitions from entry state 410.sub.1 to intermediate state 410.sub.2 (denoted as transition 1), transitions from intermediate state 410.sub.2 to returning state 410.sub.3 (denoted as transition 2), and transitions from returning

state 410.sub.3 to entry state 410.sub.1 (denoted as transition 3). The "graph return" method invoked at returning state 410.sub.3 pulls the next state from the graph stack.

As described herein, a graph region may include one or more graph functions, and each graph function is a set of states and associated state transitions that are partitionable from other graph functions of the graph. Thus, an incomplete graph may be instantiated such that the graph includes defined portions and undefined portions. The defined portions of the graph may be executed to provide various functions, even though the graph includes one or more undefined portions that have not yet been defined. The undefined portions of the graph may be defined and instantiated on-the-fly while the defined portions continue running. A graph including defined portions and undefined portions is depicted and described herein with respect to Figure A.5.

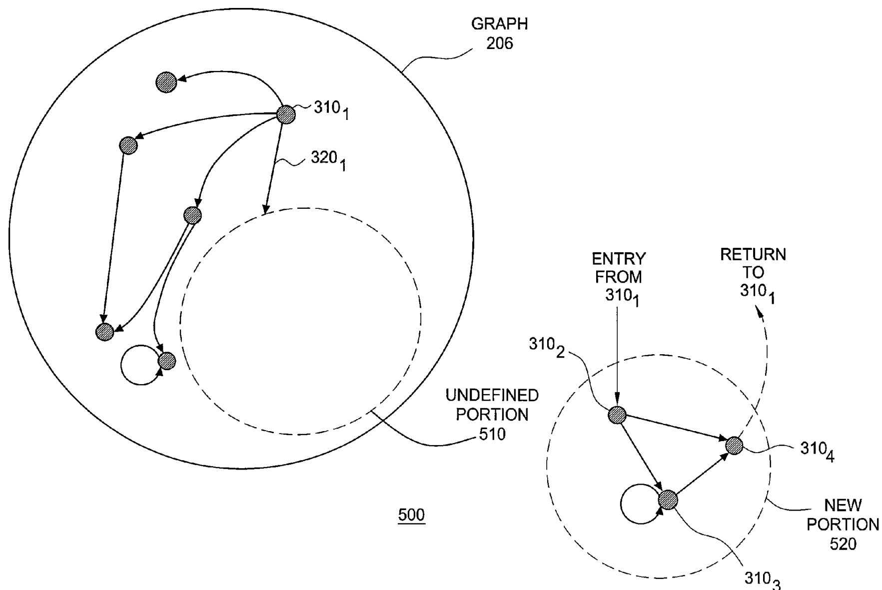


Figure A.5:

Figure A.5 depicts a graph including defined portions and undefined portions. As depicted in Figure A.5, a portion of call flow graph 206 is undefined (denoted as undefined portion 510), which the remaining portions of call flow graph 206 are defined. The defined

portions of call flow graph 206 include a state (denoted as state 310.sub.1) which has an undefined state transition (denoted as state transition 320.sub.1). The state transition 320.sub.1 is a transition from defined state 310.sub.1, however, there is no corresponding state to which a transition may be made (i.e., the state transition 320.sub.1 has not yet been defined and, thus, is associated with undefined portion 510 of call flow graph 206). Thus, a token preparing to traverse the undefined state transition 320.sub.1 may not make such a transition because the next state is undefined.

Using the present invention, in response to the token detecting the undefined portion 510 of the call flow graph 206 (i.e., detecting that there is no state to which a transition may be made along state transition 320.sub.1), the token is suspended, and the undefined portion 510 of call flow graph 206 is defined to form a new portion of the graph (denoted as new portion 520). As depicted in Figure A.5, new portion 520 includes a plurality of states (states 310.sub.2, 310.sub.3, and 310.sub.4) and state transitions adapted to perform various functions. In other words, functionality that was not previously instantiated, but which is now required, is defined and instantiated on-the-fly as needed. A method for dynamically defining and instantiating an undefined portion of an instantiated graph, while the defined portions of the graph continue running, is depicted and described herein with respect to Figure A.6.

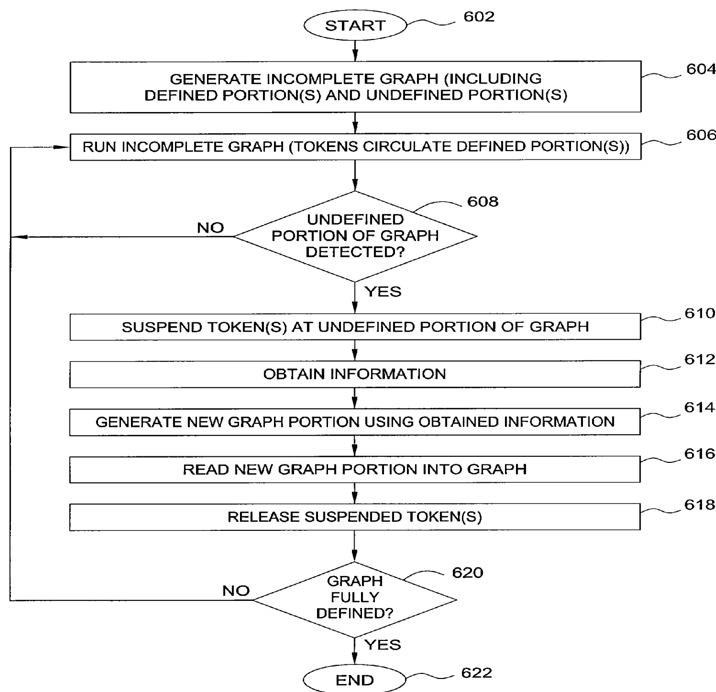


Figure A.6:

Figure A.6 depicts a method according to one embodiment of the present invention. Specifically, method 600 of Figure A.6 includes a method for defining and instantiating an undefined portion of a graph. As described herein, method 600 of Figure A.6 enables an undefined portion of a graph to be instantiated on-the-fly (i.e., while defined portions of the graph continue running to provide functions and services). Although primarily depicted and described as being performed serially, at least a portion of the steps of method 600 of Figure A.6 may be performed contemporaneously, or in a different order than depicted and described with respect to Figure A.6. The method 600 begins at step 602 and proceeds to step 604.

At step 604, an incomplete graph is generated. The incomplete graph is generated by running one or more definition files through an execution engine. The incomplete graph includes defined portions and undefined portions. A defined portion of the incomplete graph

includes defined states and state transitions, and tokens may traverse the defined portion of the graph to execute one or more functions. An undefined portion of the incomplete graph does not include any defined states and, as such, when a token encounters an undefined portion of the graph, the token is unable continue traversing the graph to provide functions (i.e., there is no defined state to which the token can transition in order to provide a function).

At step 606, the incomplete graph is run (i.e., the software is operating and one or more tokens traverse the defined portions of the graph to provide one or more defined functions supported by the defined portion of the graph). At step 608, a determination is made as to whether an undefined portion of the graph is detected. The undefined portion of the graph is detected by a token (or tokens) as the token traverses the incomplete graph. If an undefined portion of the graph is not detected, method 600 returns to step 606 (i.e., the tokens continue to traverse the graph until encountering an undefined portion of the graph). In an undefined portion of the graph is detected, method 600 proceeds to step 610.

At step 610, the token (or tokens) at the undefined portion of the graph is suspended. The other tokens of the graph may continue to run (unless one or more of those tokens encounter the same undefined portion of the graph or a different undefined portion of the graph). In other words, defined functions of the graph may continue to operate while any undefined functions of the graph are dynamically defined and instantiated on-the-fly in real time. Thus, the present invention enables undefined functions of the graph to be defined and instantiated without impacting defined functions of the graph (i.e., without taking the system down or switching system functions to redundant hardware while undefined functions of the graph are defined).

At step 612, information is obtained. The information is obtained by the execution engine. The information may be obtained locally (e.g., reading information maintained on the local system on which the execution engine is implemented) and/or remotely (e.g., information maintained on other systems that is retrieved by the system on which the execution engine is implemented, e.g., information running on the network). The information may includes

the definition of the states in the new graph, the set of correspondence states, those that are in both the old and the new graph, a state in the old graph and the corresponding state in the new graph, and like information, as well as various combinations thereof.

At step 614, a new graph portion is generated. The new graph portion is generated using the obtained information. In one embodiment, the new graph portion may be generated using information associated with one or more of the defined portion(s) of the graph. In one such embodiment, for example, the new graph portion may be generated using information associated with the defined portion of the graph traversed by the token before the token encountered the undefined portion that is now being defined. In one embodiment, the new graph portion is generated by creating or updating one or more definition files, and processing the definition file(s) to generate the new graph portion.

At step 616, the new graph portion is read into the graph. In one embodiment, the new graph portion may replace the state at which the undefined portion of the graph was detected. In one embodiment, the new graph portion may provide one or more state transitions, such that the suspended token(s) may proceed from the state in which the token(s) was suspended to one or more additional states defined in the new graph portion (or at least to loop at the state in which the token(s) was suspended, where no additional states are added). At step 618, the suspended token(s) is released. The suspended token continues to traverse the graph, including any previously defined portions of the graph, as well as the new graph portion.

At step 620, a determination is made as to whether the graph is fully defined. In one embodiment, the determination is made as to whether the graph is fully defined may be omitted. In this embodiment, the execution engine continues to monitor for a determination that an undefined portion of the graph is detected (i.e., since the execution engine may or may not know whether or not the graph is fully defined or still has undefined portions). If the graph is not fully defined, method 600 returns to step 606, at which point the incomplete graph continues running, with tokens traversing the graph to execute various defined functions. If the graph is fully

defined, method 600 proceeds to step 622, where method 600 ends.

Therefore, since the present invention enables a software application to be deployed in stages (e.g., with an initial set of functions being supported at the time of deployment and additional functions being defined and instantiated on the software application as needed), the present invention thereby reduces the initial costs of deploying the software application and delaying the costs of deploying additional functions and services. In other words, for a system targeted to support numerous functions, a portion of the functions may be made available to customers, with the remaining functions being undefined at the time of deployment and being defined and instantiated within the software application later, as needed.

Thus, since control is explicit and data is explicit, an undefined region of an instantiated graph can be isolated from remaining portions of that graph, thereby enabling the execution engine to restrict the flow of tokens into the undefined portion of the graph while the undefined portion of the graph is dynamically defined and instantiated on-the-fly. By isolating the undefined portion of the graph, computations in the other portions of the graph may continue to proceed while the undefined portion of the graph is defined and instantiated. Thus, since control over the undefined portion of the graph is so explicit, the undefined portion of the graph may be defined and instantiated in a manner that obviates the need to perform many functions typically required to be performed during software deployments, such as monitoring what is on the stack, monitoring what is on the heap, determining what is invalidated and where it is invalidated (e.g., how far down the stack), and the like.

Furthermore, since the present invention enables software of a system to be defined and instantiated in place, i.e., without switching to a redundant system or taking the system down during the upgrade, the present invention significantly reduces the number of protection elements required to protect functions provided by the system. For example, instead of deploying two pieces of hardware for each function that requires protection, the present invention enables one additional piece of hardware to be deployed to support multiple functions (i.e., the one additional piece of hardware may be

oversubscribed), thereby reducing the amount of hardware that must be deployed for a system and, thus, reducing the cost of deploying that system.

For example, for a system currently supporting nine functions but intended to eventually support more than nine functions, since the present invention enables one (or two, or three, and so on) function to be defined and instantiated while the nine existing functions continue to operate, rather than requiring additional redundant hardware sufficient to support all ten functions when only one function is being defined and instantiated, additional hardware sufficient to support only a subset of the ten functions may be required. Thus, for example, the system may be deployed with additional hardware sufficient to support one or two functions (e.g., for use during upgrades, or in case of a failure condition in which one or two functions need to be switched to the redundant hardware).

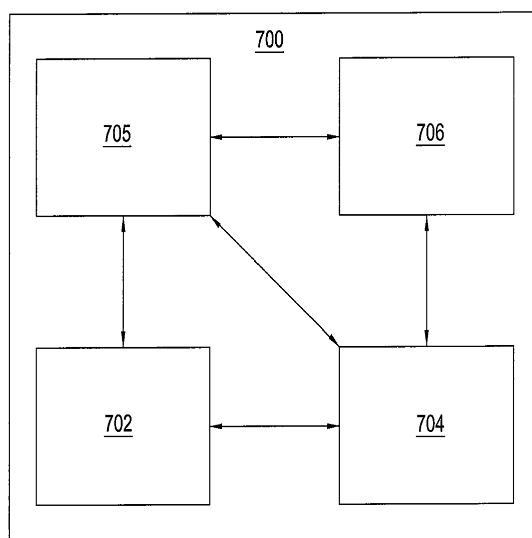


Figure A.7:

Figure A.7 depicts a high-level block diagram of a general-purpose computer suitable for use in performing the functions described herein. As depicted in Figure A.7, system 700 comprises a processor element 702 (e.g., a CPU), a memory 704, e.g., random access memory

(RAM) and/or read only memory (ROM), an execution engine 705, and various input/output devices 706 (e.g., storage devices, including but not limited to, a tape drive, a floppy drive, a hard disk drive or a compact disk drive, a receiver, a transmitter, a speaker, a display, an output port, and a user input device (such as a keyboard, a keypad, a mouse, and the like)).

It should be noted that the present invention may be implemented in software and/or in a combination of software and hardware, e.g., using application specific integrated circuits (ASIC), a general purpose computer or any other hardware equivalents. In one embodiment, the present execution engine process 705 can be loaded into memory 704 and executed by processor 702 to implement the functions as discussed above. As such, execution engine process 705 (including associated data structures) of the present invention can be stored on a computer readable medium or carrier, e.g., RAM memory, magnetic or optical drive or diskette, and the like.

It is contemplated that some of the steps discussed herein as software methods may be implemented within hardware, for example, as circuitry that cooperates with the processor to perform various method steps. Portions of the present invention may be implemented as a computer program product wherein computer instructions, when processed by a computer, adapt the operation of the computer such that the methods and/or techniques of the present invention are invoked or otherwise provided. Instructions for invoking the inventive methods may be stored in fixed or removable media, transmitted via a data stream in a broadcast or other signal bearing medium, and/or stored within a working memory within a computing device operating according to the instructions.

Although primarily depicted and described herein with respect to a VoIP call agent, a VoIP call flow graph, and associated VoIP call agent functions of the VoIP call flow graph, the present invention may be used to update software in any application in which the software is represented using a graph language.

Although various embodiments which incorporate the teachings of the present invention have been shown and described in detail herein, those skilled in the art can readily devise many other varied embodiments that still incorporate these teachings.