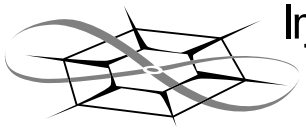
The University of Kansas



Information and
Telecommunication
Technology Center

Technical Report

Stack Local Packet Memory Interface Requirements

Stephen Ganje, Ed Komp,
Gary Minden, and Joseph Evans

ITTC-FY2003-TR-19740-06

February 2003

Defense Advanced Research Projects Agency and the
United States Air Force Research Laboratory,
contract no. F30602-99-2-0516

Abstract

This thesis extends the protocol component architecture developed for the Innovative Active Networking Services (IANS) project to include a language, a syntax checker, and a utility for dynamic analysis of interface requirements. The IANS components exchange information via a mechanism called Stack Local Packet Memory (SLPM). Each component can read or write elements in the SLPM. The tool described here ensures that information in the SLP memory is consistent for all paths through the protocol stack. A component programmer can use the output of this tool to identify problems in his/her specification and implementation and thereby address those issues more expeditiously.

Table of Contents

List of Tables

List of Figures

1 Introduction

Network protocols overcome many obstacles in order to harness the power of the network. These obstacles can take the form of bit-errors in packets, loss of packets, or overflow in the network. In Composite Protocols, components are developed to hurdle one or more of these obstacles each. Then these components are composed together to produce a composite protocol with the combined abilities of the joined components. At the University of Kansas, we have researched current models for composite protocols in order to define a model that allows components to be designed and implemented more quickly while simultaneously helping to increase the assurance that the new composite protocols developed from the components will have no negative impact on the network.

The composite protocol model used at the University of Kansas will be reviewed in Chapter One. Chapter Two will address the Interface Requirements of Stack Local Packet Memory and cover the definition of the SLPM specification language and syntax. The third chapter will introduce the utility for stack SLPM usage analysis. Finally, the fourth chapter will present related work and possible future work in this area.

1.1 KU Framework Model

In striving to define a model for component protocol definition, the goal in mind was to help enforce correctness and aid analysis. In order to achieve this, we focused on

defining a framework with a relatively simple and consistent composition operator. By being simple and consistent, we improve the ability to analyze many characteristics of individual components and how they interact as a whole when combined to form a composite protocol. After analysis of many composition methods, we decided on a linear stacking model that provides serialized event processing. Ensemble was chosen as the implementation architecture as it provided a linear component stacking approach coupled with a dual-FIFO queue event passing structure. Extensions to this architecture include a component model based on augmented finite state machines and a memory model.

1.2 Ensemble

The Ensemble architecture [6], developed at Cornell University, allows the flexible construction of layered group communications protocols. Ensemble was designed for creating a variety of distributed applications from a set of reusable components. Protocol components in Ensemble are called layers. Stacking these layers on top of each other creates composite protocols. Communication between two juxtaposed layers, one on top of the other, occurs through the use of two FIFO queues, one to pass information from the lower layer up, and the other to pass information from the upper layer down.

1.3 Component Finite State Machine Model

The component specifications follow an augmented finite state machine (AFSM) model [3, 5] that consists of a finite set of states with a finite set of transitions from one state to the next. In these models a transition is triggered by an event and a Boolean guard expression. The guard qualifies the transition according to certain conditions that can be tested with the component's accessible memory. In order to be selected, a transition must be both triggered by the specific event associated with it, and its guard must evaluate to true. Additionally, only one of the guards from the set of transitions based on the same event may evaluate to true. Once a transition is selected, the transition's corresponding action function is executed and the local memory is updated. Action functions are limited to simple sequences of non-branching statements through the proper use of guard expressions, synchronous transitions and synchronous states. A synchronous transition is one in which no event is associated with the transition; it is solely selected by the guard function's evaluation. A synchronous state is one in which there are only synchronous transitions to any next state.

A component consists of two finite state machines. The Transmit State Machine (TSM) processes events being sent from the application or component above. Similarly, the Receive State Machine (RSM) processes events arriving from the network or a lower component. The event processing of these two state machines is what defines the actions of the component they represent.

One of the main events that state machines process is the packet arrival event. This event is based on the packets that get sent through a stack for communication purposes with another protocol stack. Each packet event has two different types of memory associated with it, packet memory and stack local packet memory. The packet memory is sent to the peer of this component in another stack while the stack local packet memory can only be read or written to by components within the same stack. The action functions associated with this event are what decide a packet's course through each component and through the stack as a whole.

1.4 Memory Model

In order to make formal statements about a component, identifying all memory the component accesses is essential. Additionally, each memory's scope must be carefully detailed. As long as memory is localized within the boundaries of a component, there is a degree of safety as any problems are also localized to the component. Extra precaution should be taken with memory that extends outside of the component's domain as these memories interact with other components and entities outside of the component. We have identified the following four groups of memory in our framework model based on the memory's accessibility and scope:

1. Component Local Memory
2. Packet Memory
3. Global (external) Memory
4. Stack Local Packet Memory

The following diagram provides a graphic display of each category of memory relative to the host node, a protocol stack, and individual components.

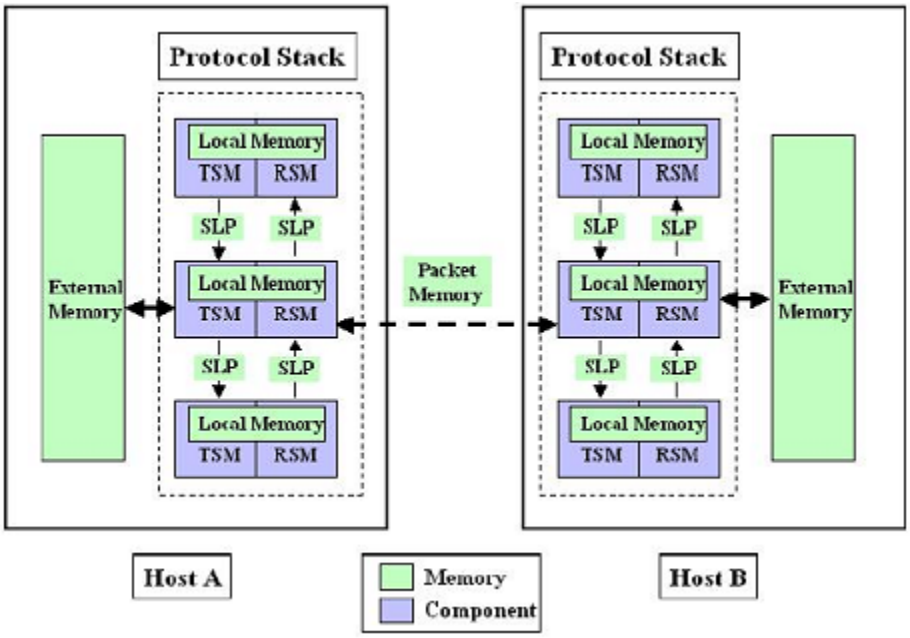


Figure 1: The different types of protocol memory and their locations within a stack.

1.4.1 Component Local Memory

Component local memory is internal to and accessible by a single component. This memory is accessed only through the action functions from the TSM and RSM of the component. This memory is separately instantiated at the sending and receiving hosts. If the component is part of a duplex protocol (transmitting information in both directions from the sender and receiver), then the TSM and RSM on a single host share the same component local memory.

The component developer determines the format and content of this memory for each protocol component specification. This memory is instantiated with the protocol component initialization. If an interface is defined to take initial parameters, a component can be tweaked in some ways to a user's specifications. All direct access to this memory, read and write, is strictly limited to a specific component instantiation.

1.4.2 Packet Memory

Packet Memory is used to transmit information to a component's peer in a stack that the current stack is communicating with. The component programmer defines this memory's format according to what is necessary to achieve the component's purpose. Some examples are the value of a checksum or a sequence number. The memory is accessible in its defined format only for the component and the component's peers. For any component below, all the previous packet memory appears as an array of bytes. This expression is permitted to allow components access to the data without allowing them to "peek" into specific fields. For example, a checksum component could run a checksum algorithm over the bytes or a fragmentation component could chunk the data into fragments, but neither would have direct knowledge of the information in the fields that previous components wrote. This constraint is applied to help keep components individualized with as few direct dependencies on other components as possible.

Packet memory is instantiated by setting fields in the user defined data structure during packet processing and then this structure is sent with the packet. The framework determines how this data is converted for transmission, but upon arrival to a peer, it is returned to the component's defined packet memory format and the component makes use of it.

1.4.3 Global (External) Memory

Regardless of attempts to avoid Global Memory and the dependencies it incurs, some protocols need access to this external memory. Global memory contains information that is shared among separate protocol stacks on the same host. For example, a node's routing tables are utilized by routing protocols such as RIP or OSPF, IP forwarding protocol components, and management and monitoring protocols. This memory has an arbitrarily wide scope and extent. The designers of all the protocols that utilize the external memory must first agree on its format, access rights, etc. Likewise, any modification to this shared memory by one protocol design may require additional modifications by any other components utilizing the memory.

Global memory is outside the extent of the protocols that access it and must be instantiated and maintained by the node environment in which the protocols that access it execute. This memory must be instantiated on each node in which one or more of the memory sharing protocols execute.

Although we cannot avoid the use of this memory in all cases, there are some methods that can help make its use more tractable. The first step is to abstract the

access of global memory, reads and writes, through a functional interface. Secondly, each component specification must declare all external memory functions it utilizes. The second makes explicit the dependencies of a component, and the stack that contains the component, on external memory. The first allows definition, maintenance, and control of the external memory to be separated from the protocol stacks that use it. Through these restrictions we can allow components the access to global memory while simultaneously trying to keep the two as independent as possible.

1.4.4 Stack Local Packet Memory

Stack Local Packet Memory (SLPM) provides a mechanism for components within a protocol stack on the same host to share information. The memory is local to the stack, but is only accessible when a component has access to the packet to which the memory belongs. SLPM travels up or down a stack with a packet event and therefore its duration transcends synchronous transitions that are a part of the AFSM model.

In our architecture, SLPM is structured as an association list of name value pairs. The name is a regular string, while the component programmer defines the format of the value. Once originally defined, later components utilizing the memory must adhere to the same format. This unfortunately results in dependencies among components and foreknowledge of at least the format of the value of the SLPM that a component needs to use. This memory is instantiated by the use of an SLPM write in a component and is utilized by components invoking a read with the SLPM's name.

These accesses occurring in the wrong order can cause several problem issues in a stack and therefore should be carefully specified and analyzed to insure proper usage.

2 Interface Requirements

Idealistically, all protocol components are independent of each other and do not need direct interaction or knowledge beyond their own peer-to-peer interaction. Unfortunately, this ideal is unattainable in many instances for the kind of services we wish to provide. In these cases it is necessary to go beyond the knowledge obtainable through normal interaction and seek additional knowledge.

An example of this interaction is a component that tries to provide fragmentation for a given stack [16]. At first this seems quite simple, break a large packet into several smaller ones. It's upon delving a little deeper into the specification that the intricacies become clearer. What size should the smaller packets be? They should only be as large as the network interface is able to handle, so in some way we need to determine this information, the maximum transmission unit (MTU). If the designers try to get even more sophisticated, they might try to determine the MTU of the path from the source to the destination so that one fragmentation is sufficient to transmit a packet all the way to the destination. Looking at it in the specific light of composite protocols, this component may be stacked on one or more other components. At the time of fragmentation, it becomes clear that the MTU alone is insufficient for this component to function properly. It also needs to know the maximum header size that each component between itself and the sending interface will contribute so that it can properly fragment a packet to the correct size. Without this knowledge, this

component's attempt to provide fragmentation may fail horribly because the best it could do would be to guess the correct size. This is just one of the many obstacles we wish to address through the use of interface requirements.

An interface requirement is a characteristic of network protocols, but they occur because of the necessity for some services to have more knowledge than the basic knowledge they are innately provided as components. They must step outside of their bounds to access this information or have this information explicitly passed to them. We have carefully limited the possible outlets for these requirements to make them explicit when they are used in the design of a protocol component. By making them explicit, we achieve several results:

1. The protocol designer can determine if the interaction is necessary or if it is possible to manipulate the design to remove the interaction.
2. If possible, restrict the extent of the interaction.
3. Recognize that these interactions are most likely where problems will occur, and through this realization, better focus design and implementation.

In our architecture model, we have identified three types of interactions that fall in this category: initialization parameters, memory, and control events. Initialization parameters allow the application to pass basic one-time information to a component. This allows for init-time configuration and is therefore easy to spot, but the information is limited, as it cannot be dynamically updated throughout the execution. This would be the case of the application giving the fragmentation component a set size to fragment to, and the component would thereafter fragment packets to that size

regardless of any dynamic changes in the network. Another possible interface, and likely the most prevalent, is through memory. We have defined several different classes of memory, varying in scope and extent. Any memory access that is not local to the component must be accounted for, including but not limited to SLPM and global memory, and also the type of access, be it a read or a write. Global memory access is further constrained by requiring it to be through a functional interface. The last interface is through control events. Control events provide a mechanism for components to communicate between, but not among each other.

An interesting element of interface requirements is that they are not only well defined, but also explicitly defined. Because of this nature, it makes them more amenable to static code analysis to determine if the requirements are met. A straightforward scan could determine if the interaction existed and of what type, init parameter, memory read or write, or control event send or receive. When putting together a stack, this information could be used to help determine if the interface requirements were upheld, a writer for every reader, etc. Determining that a component does not have an interface requirement should be fairly complete, "No, this component has no reference to an exterior interface." If it does have a requirement, the answer can be a little more devious. If a reader only reads sometimes and a writer only writes sometimes, will those times be coordinated correctly? For this type of determination, we need more thorough proof. One possible solution, and the approach used in this thesis, is to enumerate the state paths in a stack of finite state machine specifications to determine if the requirements were actually met for each case.

It should be clear that interface requirements are an important characteristic of the interaction of protocols. The ability to show when they exist and are satisfied is an essential part of the correctness of any stack of components.

2.1 SLPM as an Interface Requirement

There are two parts of an SLPM interaction, a reader and a writer. The reader is requesting information from the SLP Memory, so it is therefore the source of an SLPM requirement. The writer will write information to the SLP Memory. Alone, it does nothing more than provide information (although unread information could be a sign that the information isn't needed or that something isn't quite right in the stack). However, in the case of a requirement being in place for the information being written, this write would satisfy that requirement.

A normal transition or a string of synchronous transitions is concluded by the use of a packet transfer function. For a regular transition, all SLPM writes that occur previous to the packet transfer carry through with the packet event to the next component. Synchronous transitions, however, must be mapped to transitions by concatenating each synchronous transition on the way to the packet transfer. Depending on the branching of the synchronous transitions, each list may result in one to several regular transitions. Each possible transition of a packet through a component can result in a different group of SLPM written and/or read. Because of these different groups, it can't be flatly stated that a component accesses a certain SLPM, although it is true that it does so in some instances. In order to fully qualify

the expression, it must be stated that a component accesses a given SLPM in a given transition.

Having a write of a specific SLPM come before a read satisfies the SLPM interface requirement. However, determining the chronological relationship between a read and a write is a difficult matter. If the TSM only sent packets down the stack, and the RSM only sent packets up the stack, a chronological order could be determined by having a write occur higher in the stack than the read in the TSM, and vice versa for the RSM. As the architecture stands, it is possible for a TSM to send a packet up the stack (the forwarding component returning a packet that was sent from and to the same host) and a RSM to send a packet down the stack (any component that transmits an ACK). Therefore, in order to determine the chronological order of SLPM reads and writes, the possible paths a packet can take through a stack must be mapped out. By observing each possible complete path of a packet, the chronological order can be determined for each path instance and the SLPM interface requirement's satisfaction or failure can be thereby derived.

The New Packet transfer functions cause some side effects in the satisfaction of an SLPM interface requirement. In the case of a peer-to-peer New Packet transfer, a new packet may originate from the given component somewhere in the middle of the stack. New Packet transfers that follow a previous packet arrival event also generate a new packet, but they have access to the old packet as well. Because a new packet is created, the previous packet's path through the system comes to an end and the new packet takes up where the old one left off. In these cases, any SLPM that belonged to

the old packet is forgotten and the SLPM starts over with the new packet. There are two instances that must be addressed for these cases. The first is that some components may find the need for the ability to write SLPM to these new packets. This cannot be accomplished through an ordinary write because the packet transfer function is the last expression to be evaluated in a transition. With a new packet transfer, any write made wouldn't make it to the next component. In order to accomplish this desired write, an additional SLPM access function, New Packet Write, has been created. New Packet Write achieves the same purpose as a normal write on a normal transfer, but is made specifically for a new packet transfer. The second case is the instance in which a component needs to use a new packet transfer, but doesn't intend to delete the SLPM from the old packet. To handle this case, the SLPM transfer was created. This procedure transfers any previous SLPM from the old packet to the SLPM of the newly created packet.

After this description, it should be clear that there is much more involved in the satisfaction of SLPM interface requirements in a stack than having a component that writes to the SLPM and having a component that reads from the SLPM in the same stack. However, while complicated, the use of SLPM is not beyond analysis.

2.2 Examples of Stack Local Packet Memory

Understanding SLPM and its possible uses may be easier by seeing current examples of SLPM in components. In the components written to date, the use of SLPM has taken on a few repeated forms. The first form is used for supplying

information to a component that comes later in a stack from the writing component. In the TSM a component higher in the stack provides information for a component lower in the stack and vice versa for the RSM. The second form occurs when a component is executed in the network and the packets get rerouted back down the stack after being received. Some of these components, TTL for example, find it necessary to conserve their packet memory from the RSM in order to reuse it for transmission in the TSM. In these cases, a write occurs in the RSM to be later read in the TSM. The final form is to reduce stack execution and improve efficiency. An example of this usage can be seen through the Multicast Forwarding component and the Replicator component. The multicast forwarding component has a list of hosts that the packet needs to be sent to. If the packet is replicated in this component that is near the top of a stack, each resultant packet must be processed by each lower component when the packets are actually just copies of the same original packet. To reduce this execution overhead, the multicast component writes a list of the hosts which the packet must be forwarded to into SLPM and transmits a single packet. Near the bottom of the stack, the packet arrives to the replicator component that reads the SLPM and sends a copy of the packet to each host in the list. The same packets could be sent without the use of SLPM, but this usage allows a stack to be more efficient by splitting a component's functionality into different components and avoiding repetitious execution of components between the two. Some examples of SLPM are as follows:

Incoming Interface – The IP address of the packet's incoming interface is stored for a Reverse Path Forwarding check in a later component.

Next Hop Address – This is used to set the next hop of the packet and also as the packet reaches the wire to know where the packet should be directed next.

Next Hop Address List – This SLPM is used by multicast components to set a group of next hops when a router must forward several copies of the same packet to different addresses.

Packet Destination – This SLPM is used to offset the lack of routing capability in components that send acknowledgments. For example, reliable components function through the use of ACK's or NAK's, however, these components do not keep track of to whom they should send the response. They expect this information to be handled by the component that does routing. Therefore the routing component writes this information to the packet and then later reads it from the ACK to know where to send the packet.

Packet Type – This is used by a component wishing to set the type of the packet to a unicast or a multicast. Depending on this setting, the packet is handled differently by the stack's forwarding components.

Reliable Sequence Number – In most cases, a reliable delivery component would not need to use this SLPM reference. However, in the instances of hop-by-hop reliability and reliable multicast where the reliable components are placed below the forwarding components, this sequence number must be maintained for retransmission of the packet.

Source Address – Occasionally components will process a packet differently if it originated from the same host as the component's stack. One example is the TTL component that sets the TTL to be the max when the packet originates from the same host, but reads the previously saved value when the packet originated elsewhere.

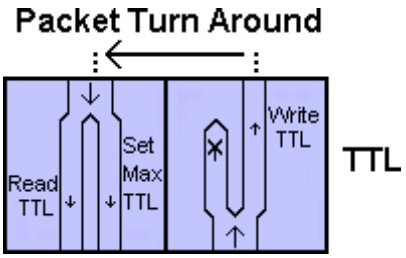


Figure 2: TTL is a component that writes to SLPM only to be read by itself.

TTL (Time To Live) – The time to live counter is decremented at each hop of a packet on its way to the final destination. However, this component must be placed below the forwarding component and therefore must save the packet's TTL in SLPM to be processed when the packet is forwarded. When it returns to the component, the TTL is read from the SLPM and transferred back to Packet Memory.

2.3 SLPM Requirement Specification Language

Modeling SLPM usage in components requires attention to many of the finer details of component structure. Each of the following elements plays an important role in SLPM usage within a stack and must be represented by the SLPM specification language.

2.3.1 State Machines and Transitions

Components are made up of finite state machines that respond to incoming packets by transitioning to new states and processing each packet accordingly. Each component is composed of two state machines. The Transmit State Machine (TSM) handles the packets that are received from an upper component while the Receive State Machine (RSM) handles the packets received from a lower component. Knowing the current state machine is important to determine the set of possible transitions out of the current state. It is also important to track in which state machine and transition an SLPM access occurs to better pinpoint the source of faults that occur.

2.3.2 SLPM Accesses

Access to SLPM can either be through a read or a write. Accesses of different types of SLPM are independent of each other. However, the order of reads and writes of the same SLPM can cause various issues of which a component programmer should be aware.

- Read with no previous Write – This is a failure case likened to attempting to utilize an un-instantiated variable and in the least case would cause an exception.
- Multiple Reads per Write – In most cases multiple reads aren't a problem, but the user should at least be made aware of the existence of such a case.
- Write that was never Read – In this case, a component writes to the SLPM, but no other component ever tries to read it. This may be an instance of where a component is trying to do more than it should.
- Previous Write was Overwritten – Some component wrote to the same SLPM as a previous component. This shouldn't cause a system failure, but an overwritten value may be a signal that one component is doing more work than it should or that the other may be unexpectedly interfering.
- Dirty Reading and Writing – In theses cases, a previously written value has been read and then later overwritten. This overwritten value may then be read by an even later component. This could cause a problem if the two separately read values were supposed to remain the same, however, the components could also be purposely reading and writing in pairs.

An SLPM write can be achieved through either a Write or, in the case of a New Packet Transfer, a New Packet Write. Previous SLPM may also be transferred from an old packet to a new one in the New Packet Transfers through the use of the special access SLPM Transfer. A read is achieved through the Read access.

2.3.3 Packet Transfer Functions

In order to transfer a packet between components, a component has access to various transfer functions. These functions determine a packet's next position in the stack relative to its current position. SLPM accesses are associated with the packet transfer that shares the transition in which they occur. The packet transfer functions are an important part of the SLPM language because they help determine the path of a packet through a stack.

- Packet Send – The normal transmission function that sends a packet to the next lower component, closer to the network wire.
- Packet Deliver – The normal delivery function that sends a packet to the next higher component, closer to the application utilizing the stack of components.
- New Packet Send and Deliver – Special cases of the normal send and deliver functions. These functions create a new packet to send in the corresponding direction up or down the stack. One example of the need for these functions is during fragmentation when several packets must be reconstructed and delivered as a completely new packet. Another use occurs when a component sends its peer a message after receiving an event other than a packet arrival (a timeout, etc.). Because the packets generated are new, they do not contain any SLPM unless a special access function, New Packet Write or SLPM Transfer, are also invoked in the same transition.
- Keep Packet – Saves a packet and its current SLPM for later transfer from the component's same state machine.

- Transmit and Deliver Kept Packet – The corresponding send and deliver functions to transfer a packet after it has been saved using Keep Packet.
- Drop Packet – Drops a packet on the floor. One of the ends of a packet's path through a stack. A packet that has been dropped is no longer accessible and transfers neither up nor down the stack. It is removed from existence.

2.3.4 SLPM Specification Syntax

The purpose of the specification syntax is to specify all of the above elements of a component and their relationship to each other. Through a structure that specifies these elements, it is possible to analyze and determine possible fault points of each component in relation to the others in a stack. The table below gives the syntax in Backus Naur Form. Examples of several components written in this specification syntax can be found in Appendix A.

Table 1: SLPM Specification Syntax

```
STACK         ::= [ <COMPONENT>+ ]

COMPONENT     ::= ( <NAME> [ <PKT_TRANSFER>+ ] )

PKT_TRANSFER ::=
| ((Transmit <TRANS_NUM> <EVENT>?) KeepPkt      [ <SLPMINTERFACE>* ])
  ((Transmit <TRANS_NUM> <EVENT>?) <KEPTPKTFUN>[ <SLPMINTERFACE>* ])
| ((Receive  <TRANS_NUM> <EVENT>?) KeepPkt      [ <SLPMINTERFACE>* ])
  ((Receive  <TRANS_NUM> <EVENT>?) <KEPTPKTFUN>[ <SLPMINTERFACE>* ])
| ((Transmit <TRANS_NUM> <EVENT>?) <TXTFUN>     [ <SLPMINTERFACE>* ])
| ((Receive  <TRANS_NUM> <EVENT>?) <RCVFUN>     [ <SLPMINTERFACE>* ])

(* No event defaults to PktArrival *)
TRANSITION    ::= (<SM> <TRANS_NUM> <EVENT>?)

SM            ::= Transmit | Receive

TRANS_NUM     ::= {1-9}{0-9}*

EVENT         ::= PktArrival | Timeout | Control

TXTFUN        ::= PktSend | NewPktSend | PktDeliver | DropPkt

RCVFUN        ::= PktDeliver | NewPktDeliver | PktSend
                  | NewPktSend | DropPkt

KEPTPKTFUN    ::= DlvrKeptPkt | TxtKeptPkt


SLPMINTERFACE ::= ( <ACCESS> <NAME> )
                  | ( Transfer <WILDCARD> )

ACCESS        ::= Read | Write | NewPktWrite

WILDCARD      ::= _

NAME          ::= {a-z | A-Z}{a-z | A-Z | - | _ }*
```

3 Mechanical Validation of SLPM Usage

The use of finite state machines as the execution method for components makes it possible to build a stack with a finite number of paths a packet can take through the system. By tracing these paths, we can pinpoint where errors might occur within the stack, and, having pinpointed them, can work in a directed manner towards either removing these bugs, or showing that the path with the error could not have occurred. By using a computer program to automate this process, we can quickly and efficiently determine these problem paths and work towards their elimination.

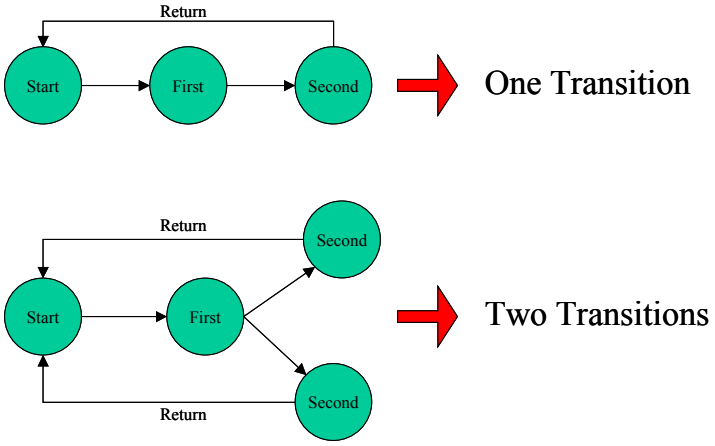


Figure 3: The concatenation of synchronous transitions into regular transitions.

3.1 Component Specification to SLPM Specification

The first step to analysis begins with the translation from a component specification to a SLPM specification. In order to start this process, each transition that calls a packet transfer function is determined along with any SLPM accesses that occur therein. However, the AFSM's used in the component specifications provide for

special synchronous transitions. These transitions are handled by creating a regular transition for each possible permutation of the synchronous transitions that produces a different result than the others. The figure above shows a few simple examples of how this is achieved. Unfortunately, as the transitions get more complicated, it takes more careful analysis to determine how many different transitions are actually produced as far as the SLPM is concerned. The figure below gives an example of a state machine with a loop that could produce infinitely many different transitions as far as states go. However, each group of transitions all have in common the write of SLPM A and therefore only the one transition results. The second example shows how two different transitions would result if there was also a write in the later half of the state machine. Additionally, if a SLPM value is read in a guard to determine the correct path to take, each resulting path must include the read of that SLPM as a part of their transition.

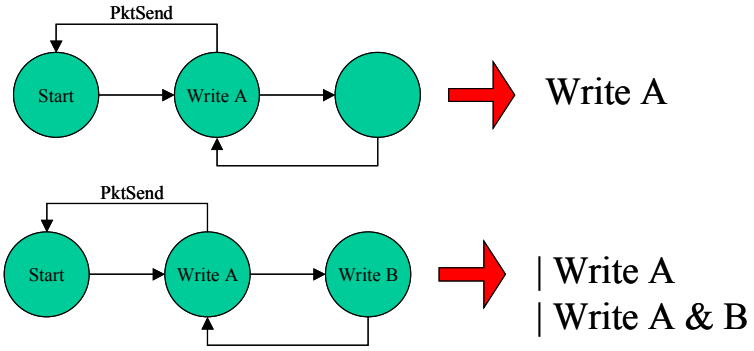


Figure 4: Transition concatenations that require more careful examination.

Once the transitions through a component have been determined, each one is recorded along with the state machine in which it occurs, the packet transfer function that determines the packet's next destination, and the SLPM accesses that occur previous

to the execution of the transfer function. When all of these have been recorded, the SLPM specification is complete.

3.2 Syntax Checking of SLPM Specification

An additional program has been created to check the syntax of an SLPM specification to aid in the writing of these specs. The syntax check is also applied before the application of the stack verifier to insure only components written in proper syntax are being used.

3.3 Packet Paths

Each possible transfer of a packet from one component to another gives a different possible path that a packet may travel through the component. Each step of the path includes the current component's name, state machine, transition, event, packet transfer function, and any SLPM accesses that occur before the packet transfer is called. Through the use of this information, it is clear that some paths through a component access certain pieces of SLPM, while others do not. By stacking these components together, it can be seen that there are a number of different possible paths through the same stack. The paths are based on branching that occurs from each packet transfer in the same state machine of a component that the packet entered. Some of these paths will be correct, while others may cause unwanted side effects that need to be analyzed. Below is an example stack of components with the different possible routes through each. Some components will only have one route through

them while others will have many. When a packet is dropped, the path is ended and the current SLPM is evaluated for warnings. When a packet is turned around, as in the forwarding component, the packet goes to the next component above or below, but to the opposite state machine.

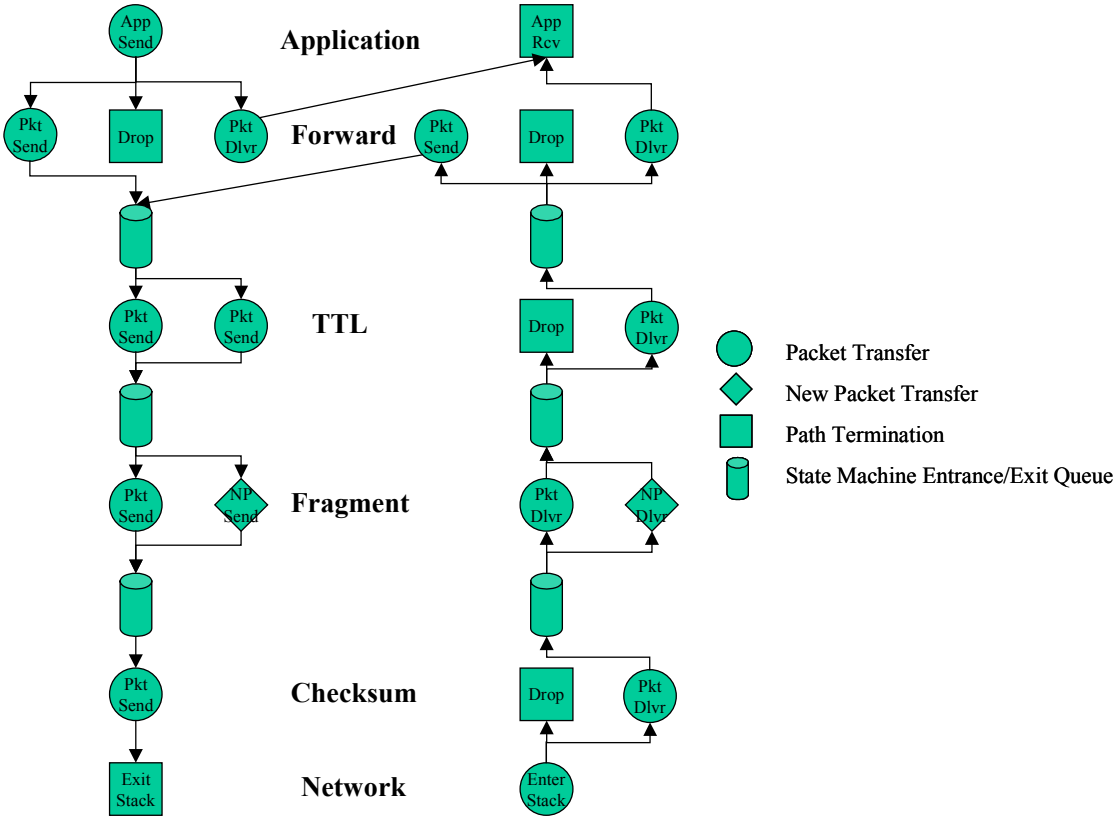


Figure 5: A stack model showing the alternate paths of a packet through a stack.

The AFSM's of components define transitions not only through incoming events, but also through the use of guards. A guard is a Boolean test that may be based on any information available to a component at the point in time it is processing an event. Because the only knowledge allowed to the SLPM specification is that of SLP Memory, these transitions appear non-deterministic as far as the SLPM is concerned.

As a result, each transition that matches the correct event must be searched as a possible path through a component. Unfortunately, this can result in paths that are invalid. If an invalid path produces no SLPM access faults, this causes no problems. However, if one does cause a fault, a warning message is produced with the path in question. In order to resolve this invalid fault, one would need to show that this path could never occur. This can be occasionally accomplished when the point at which the packet goes down the invalid path has a guard based on the SLPM. Otherwise it can become more complicated.

In the figure below are four possible trace paths through the previous stack. The first path is from the application sending a packet of data that gets fragmented in the fragmentation component. The second path is a packet from the network meant for the application that was not fragmented on transmission. The third path is a packet received from the network, but not meant for this stack. It is sent back down and out the stack to be forwarded to the next hop. The fourth path is from a packet that has exceeded its maximum number of hops in the network and is dropped by the TTL component.

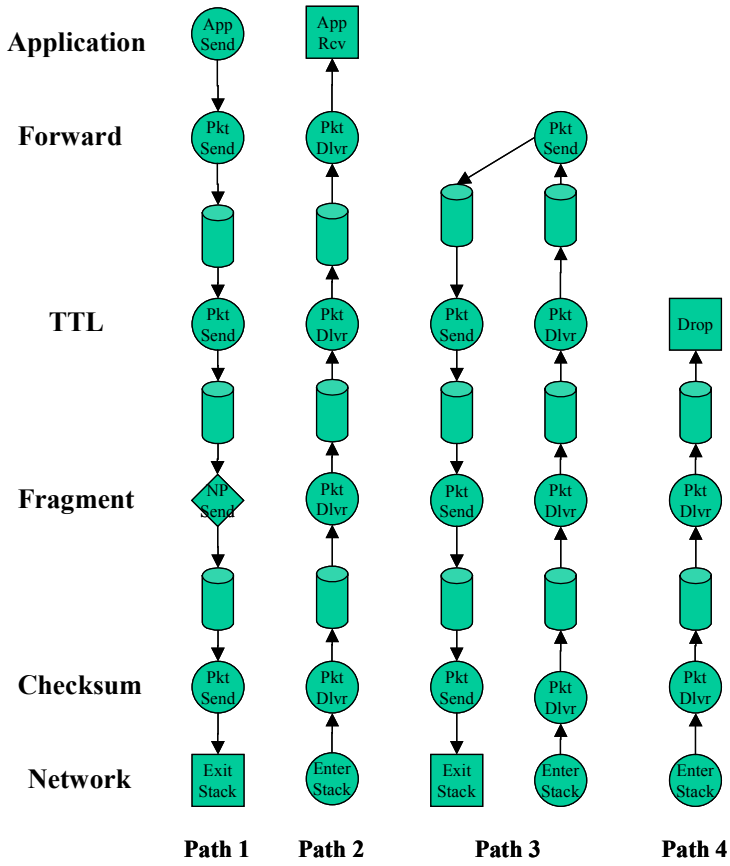


Figure 6: Four possible paths through the previous stack.

3.4 SLPM Path Validation Algorithm

The validation of a stack begins exactly where a packet enters the system, at the top of a stack for a transmission, at the bottom of a stack for a receipt, or somewhere in the middle for a peer-to-peer message originating in a non-packet arrival event like timeout. From these points, the initial paths are created as if a single path had entered the top component's TSM, the bottom component's RSM, and the next component in line from the peer-to-peer message. The branching in each path is calculated by creating a new path for each possible transition out of the current state machine. The

current SLPM interface requirements for each path are then calculated by merging any SLPM accesses with the accesses that have already occurred. A path ends if the corresponding packet leaves the stack, a packet loop is detected, the packet is dropped, or a failure occurs from trying to read a non-written SLPM value. Each path is extended until one of the end conditions is met. Once all paths have terminated, the warnings and failures that were produced by the stack are displayed according to the path in which they occurred. The table below shows events that occur throughout the validation and how the program responds to them.

Table 2: List of events that may occur in a packet path.

Event	Definition	Actions
Read Miss	A component tries to read a SLPM value that hasn't been written yet	1. Flags a failure 2. Ends the path
Read Hit	A component reads a SLPM value that was previously written	1. Records a successful read of the given SLPM
Multiple Read Hit	The SLPM value read has been read previously by another component	1. Flags a warning 2. Records a successful read of the value.
Read Hit – Possibly Dirty	The SLPM value that was read has been overwritten, but had been read before being overwritten. The values may not be the same.	1. Flags a warning 2. Records a successful read of the value
Write	A component writes a value to a SLPM name	1. Records a successful write to the given SLPM
Overwrite	A component writes a value to a SLPM name that already had a stored value.	1. Flags a warning 2. Records a successful write to the given SLPM
Overwrite Previously Read Value	A component overwrites a SLPM value that was previously written and read	1. Flags a warning 2. Records a successful write to the given SLPM
Packet Loop Detected	Given the provided transitions between components, an infinite packet loop occurs in the stack	1. Flags a failure 2. Ends the path

3.5 Evaluating the Validation Program's Output

To aid in understanding how the program can be used, a sample output is given in Appendix B for the following stack of components: Forwarding, TTL, Fragmentation, Checksum, and Below Bottom. The Below Bottom component is the component that attaches the stack to the network wire, and is therefore below the bottom of a stack of components. According to the output, this stack of components produces only one packet path with a failure and is the first one displayed.

```
ttl Read TTL: Read of 'TTL' in ttl without previous Write
```

This line states that in the TTL component there was a failure from trying to read the TTL SLPM field. Each warning message gives the component in which the warning occurred, the access type that was attempted, and the SLPM name that was to be accessed. After this, a more detailed message is delivered. In order to determine where this failure occurred, we next turn to the path the packet took through the stack.

```
Comp: forward        | Transn: (Transmit, 4, PktArrival)
    | TFunc: PktSend | SLPM:   (Write NextHopAddr)(Write SrcAddr)
Comp: ttl            | Transn: (Transmit, 2, PktArrival)
    | TFunc: PktSend | SLPM:   (Read SrcAddr)(Read TTL)
```

Here we see that the TSM of the TTL component in transition 2 attempted to read both the source address and the TTL SLPM's, but the TTL value hadn't yet been written by any previous component. By observing the component specification we can more clearly understand this error. In the TTL component, the SLPM value source address is read first to determine if the packet originated in this stack or is coming from another stack (i.e. previously turned around by the forwarding component). If the packet does come from this stack, then the max TTL value

provided at initialization is set, but if it originated in another stack then the packet has been received first and the TTL RSM has written the TTL value to the SLPM. This value would then be read by the component and retransmitted. In this case, since the packet originated from the same stack and was passed directly from the application to the forwarding component to the TTL component, the correct transition, according to the guard based on SLPM, is the one that does not attempt to read the TTL value from SLPM. Therefore this path would never occur and is invalid. However, if the guard statement in the implementation was flawed, this failure might result. Pinpointing this possible trouble can help give foresight to the programmer if bugs in the implementation do occur.

The previous path also created a warning, but because we have shown that this path is invalid, the warning is also dispelled. The rest of the displayed paths contain only warnings, and some of them are just variations of each other. The second path is invalid for the same reason as the first, when the packet is coming from being received, the second transition in the TTL TSM is called, not the first. In fact, the third, fourth, and fifth paths can all be dispelled for the same reason. Paths six through nine warn that the TTL and Source Address SLPM's are written but never read. This is ok because in all of these cases the packet is being either dropped in forward, or delivered to the application because it is meant for this stack. Paths ten through fifteen are each permutations of the same warning to let you know that the Source Address SLPM is read more than once. We are already aware of this because a read occurs in the TTL component and in the Below Bottom component. If this

double read is ok, then these last few warnings are also dispelled. In all, there were three distinctly different warnings, but we were able to evaluate and show that they either belonged to an invalid path or that they weren't a serious concern. This stack is therefore validated as far as the SLP Memory is concerned. If this information hadn't been known beforehand, this tool would have at least made the user aware of these issues, and being aware is an important element of safety.

3.6 Notes on Design of the Validation Program

The program utilizes a breadth first search to find all the packet traversal paths with failures or warnings. At each step of the search, the completed paths are filtered to remove non-problem paths and the remaining paths are stored. Once all of these paths are accumulated, they are sorted to order the paths with the most failures first and the most warnings second for display. In the example previously given, there were three distinct problems identified, however there were fifteen outputs. In order to reduce some of these repetitions, an output filter function is provided as an element of the stack validator. A second output is provided in Appendix B when the second level of filtering was applied. The first two paths fall under the first original problem as opposed to five paths, the next path is under the second problem as opposed to four paths, and the last path is under the third problem as opposed to 6 paths. This level of filtering is still useful for analysis and much easier to read, producing only four paths opposed to fifteen paths without filtering. There are three additional levels of filtering but should only be used to get an idea of what is happening inside the stack and not

for analysis. The first three filters are normal matching filters while the last two also filter if a path's warnings are a subset of another path's warnings. Because of the higher levels of filtering, one warning path represents multiple paths' warnings. In these cases, just because this path is dispelled of one warning doesn't mean the other paths with the other warnings it represents necessarily are. Care should be taken in these instances.

The selection of Ocaml [4, 13] as the programming language was based on a number of factors. These include strong list manipulation utilities, pattern matching, and higher order functions. Additionally, the Ensemble system and KU model have both been written in Ocaml and a utility that will be used with this system can benefit by utilizing the same language. Some of the data structures to be represented also have their initial design in the language and therefore may be more easily represented and manipulated by using the same language.

The program is more or less a model checker for stacks defined by the SLPM specification syntax. There is a question of whether it would have been wiser to implement a general-purpose model checker like SPIN to achieve the same purpose. The answer in this instance is probably not, but the concepts of process algebra and the temporal logic of actions were of vital importance to the program's creation [9, 12]. SPIN can easily show a path in which a failure occurred, but it is designed to also deal with infinite states where showing one failure is usually good enough. As our specification model uses finite state machines, we would like to see all the possible failures to be able to examine each instance. In order to utilize a model checker like

SPIN, two objectives would have to be achieved. First of all, a generic form of a component would have to be created so that each component could be translated into the guts of the generic component specification. Secondly, the ability to dispel or ignore failures and warnings after they have been initially presented would be necessary to determine all of them. The first objective is achievable, but the second may prove to be more difficult to achieve dynamically given the constraints of a model checker.

The validation program does have its limitations. Packet loops may occur if a component that can deliver a packet from the TSM and a component that can send a packet from the RSM are placed in the same stack. Through normal component execution these loops may be deterministic and eventually end, however, because of the non-deterministic nature of the paths that must be followed by the program, these become infinite loops. These loops can be detected, but they cannot be followed for the purpose of validating the path. If the loop is supposed to occur in normal operation, then this could pose a deficiency in the stack validation. However, assuming that a packet looping in a stack is something to be avoided, this would not then be a deficiency. Another limitation is that an order for same component accesses of the same SLPM is not defined. If a component reads and writes an SLPM that has been previously written, this could be interpreted as a read and a possibly dirty write, or an overwritten write and a successful read. Because of how state machines can be written, the order of the SLPM accesses may be different each time. This could cause a failure in the case that a component tried to write and read from the same SLPM

without a previous write. Nevertheless, the SLPM shouldn't be utilized for purposes that can be handled through Local Memory and this shouldn't be an issue.

3.7 Case Analysis: DVMRP Stack

In the previous example given, a relatively simple stack is shown to have its SLPM interface requirements validated because a stack problem did not exist. A specific incident where the SLPM analysis tool has caught a potential stack problem can be seen in output trace three of Appendix B. In this multicast routing stack, the DVMRP components usually do their own routing and generate their own peer-to-peer packets for communication with each other. When they create their own message, they write the source address and next hop address into the SLPM in order to direct the packet they are sending out. On the other hand, when a packet is coming from a higher component, they just pass it along assuming that some other component has already written the needed information to SLPM. This works fine if they are used as they are intended to be, separate and with no application running on top. However, if a component were to be placed above these components that just did a normal send without adding the correct SLPM, the packet would never get beyond the below_bottom component when it tries to read the SLPM that isn't there.

A separate instance highlighted through output three is that the TTL component's SLPM never arrives to be read again in the same component. This occurs because there is no component that turns a packet around to be re-routed into the network. Because of this, it is clear that the TTL component is taking up processing time, but is

never really doing anything in this particular stack. Through this determination, we see that the TTL component can actually be removed from this stack, resulting in a slightly more efficient protocol stack. The removal of this component, demonstrated by output trace four, removes these warning paths from the output while not interfering with the reproduction of the problem previously presented due to other components.

4 Conclusion

This thesis presents the SLPM component specification syntax and a utility for stack SLPM usage validation. Chapter 1 introduces a framework model for composite protocols used at the University of Kansas that includes an augmented finite state machine based component model and a detailed memory model. This introduction presents the framework upon which the utility is applied. Chapter 2 defined Interface Requirements and demonstrated the importance of their satisfaction within a stack. SLPM is introduced as an interface requirement that must be satisfied in stacks based on our framework model. A SLPM specification syntax is defined and some components are translated into this syntax in Appendix A. Chapter 3 presented the steps in the development of the SLPM stack validation program and explained how to use the output to pinpoint possible bugs in a component specification. This chapter will discuss similar work that has been done in the area of component protocols, summarizes the main results of the work presented in this thesis, and discusses the possibilities for future work.

4.1 Related Work

The idea of developing protocols from modular components has a long history. In fact, basic courses in networking introduce network concepts based on the OSI seven-layer model and protocols are designed to work with different upper and lower layers.

Ensemble [6] is a group communications system designed for constructing a variety of distributed applications from a set of re-usable components. It builds upon the Horus and Isis systems [15] and is written in Objective Caml (Ocaml) [13] a dialect of the functional language ML. Additional work done based on the Horus and Ensemble systems makes use of temporal logic to define and prove properties of protocol stacks [11]. Although much of the group communication is built-in to the system, Ensemble's model of component composition is very consistent with our framework specification's model. Additionally, being written in a functional language like Ocaml helps improve the chances for closer analysis of protocol components through the use of theorem provers, like Nuprl, as was done in [7].

The X-Kernel [10] is an operating system kernel that provides an architecture for constructing, implementing and composing network protocols. The key idea behind the X-Kernel architecture is to split the traditional protocol stack, which has a simple linear topology and complex per-node functionality, into a complex protocol graph consisting of individual protocols called micro-protocols and virtual protocols.

Cactus [17], based on the previous Coyote system [1, 2, 8], has a two-level model. Protocol components, termed microprotocols, are combined together with a runtime system to form a composite protocol. A composite protocol is composed with other protocols in a normal hierarchical manner (using X-Kernel) to form a network subsystem. Although our desired component granularity is similar to that of Cactus, its event triggered execution model was not quite what we were looking for as far as analysis. However, this same model, written in C, does have potential for multi-

processing and provided an interesting alternative approach to component composition.

4.2 Results and Future Work

This thesis has resulted in the production of a syntax checker and a dynamic stack analysis tool for SLPM usage. These tools have been used to show that current stacks are making proper usage of SLPM, and if not, have helped point out potential misuse. In the case of output three in Appendix B, they have even demonstrated that a component wasn't being utilized and, through the removal of said component, improved the packet processing efficiency of the stack.

Future work in this area may extend to the analysis of other stack interface requirements, like global memory or control events. Further extensions in the specific area of SLPM interface requirements might include a method for automatic translation from component specifications to SLPM syntax specifications. The utility discussed in this thesis provides a mechanism to analyze a stack's usage of Stack Local Packet Memory, but must be used with component specifications that have been written in the SLPM specification syntax. The SLPM specifications in this thesis have been translated by hand from previous component specifications. However, current component specifications have been written in XML, so there is potential that these specifications could be translated directly to the SLPM specification syntax. Unfortunately, pitfalls to this translation might develop in a couple of areas. First of all, part of the XML specification is direct Ocaml code in the form of helper

functions. Helper functions are a good programming practice in most cases, yet in this instance they can make translation more difficult by abstracting the access from the main body of code. This might be overcome by imposing restrictions on helper functions to disallow packet transfers and SLPM access inside the functions. Needed values could be read in the main body and passed in as parameters and any packet transfers could occur upon the return of the helper function. An alternate issue is that names can be "let bound" to values in the Ocaml code. An unintentional let binding of a value to a SLPM syntax keyword could also cause conflicts to direct keyword parsing since the syntax for Ocaml and the SLPM syntax are independent of each other. This may imply the need for a more sophisticated parsing technique. Although difficult, an extension in this area is not impossible. This extension would eliminate the necessity to create an SLPM component specification by hand because it could be achieved at runtime given the original component specification. Additionally, the component's Ocaml code will be automatically generated from the XML specifications, so the translation to the SLPM syntax for analysis will also be very consistent with the actual implementation.

Appendix A: Components in SLPM Specification Syntax

Packet Forwarding Stack Components:

```
(forward
  [((Transmit 1) PktDeliver [])
   ((Transmit 3) DropPkt    [])
   ((Transmit 4) PktSend    [(Write NextHopAddr)
                             (Write SrcAddr)])
   ((Receive  1) PktDeliver [(Write SrcAddr)])
   ((Receive  3) DropPkt    [(Write SrcAddr)])
   ((Receive  4) PktSend    [(Write SrcAddr)
                             (Write NextHopAddr)])])

(ttl
  [((Transmit 1) PktSend    [(Read SrcAddr)])
   ((Transmit 2) PktSend    [(Read SrcAddr)
                             (Read TTL)])
   ((Receive  1) DropPkt    [])
   ((Receive  2) PktDeliver [(Write TTL)])])

(fragment
  [((Transmit 1) PktSend       [])
   ((Transmit 3) NewPktSend    [(Transfer _)])
   ((Receive  2) PktDeliver    [])
   ((Receive  6) NewPktDeliver [])])

(checksum
  [((Transmit 1) PktSend    [])
   ((Receive  2) PktDeliver [])
   ((Receive  3) DropPkt    [])])


 (below_bottom
  [((Transmit 1) PktSend    [(Read NextHopAddr)
                             (Read SrcAddr)])
   ((Receive  1) PktDeliver [])])
```

Multicast DVMRP Specific Components:

```
(graft
  [((Transmit 1) PktSend [])
   ((Transmit 2 Timeout) NewPktSend [(NewPktWrite SrcAddr)
                                     (NewPktWrite NextHopAddr)])
   ((Receive 1)  DropPkt    [])
   ((Receive 2)  PktDeliver [])])

(prune
  [((Transmit 1) PktSend [])
   ((Transmit 2 Timeout) NewPktSend [(NewPktWrite SrcAddr)
                                     (NewPktWrite NextHopAddr)])
   ((Receive 1)  DropPkt    [])
   ((Receive 2)  PktDeliver [])])

(spanningtree
  [((Transmit 1) PktSend [])
   ((Transmit 2 Timeout) NewPktSend [(NewPktWrite SrcAddr)
                                     (NewPktWrite NextHopAddr)])
   ((Receive 1)  DropPkt    [])
   ((Receive 2)  PktDeliver [])])

(route_exchange
  [((Transmit 1) PktSend [])
   ((Transmit 2 Timeout) NewPktSend [(NewPktWrite SrcAddr)
                                     (NewPktWrite NextHopAddr)])
   ((Receive 1)  DropPkt    [])
   ((Receive 2)  PktDeliver [])])

(neighbor_discovery
  [((Transmit 1) PktSend [])
   ((Transmit 2 Timeout) NewPktSend [(NewPktWrite SrcAddr)
                                     (NewPktWrite NextHopAddr)])
   ((Receive 1)  DropPkt    [])
   ((Receive 2)  PktDeliver [])])
```

Appendix B: Output of the SLPM Stack Verifier

Output 1 – Packet Forwarding Stack All Outputs

```
for stack filter-level 0:
 | forward
 | ttl
 | fragment
 | checksum
 | below_bottom

These SLPM were Accessed:

  NextHopAddr
    Read - below_bottom
    Write - forward

  SrcAddr
    Read - below_bottom | ttl
    Write - forward

  TTL
    Read - ttl
    Write - ttl


  _
    Transfer - fragment

Packet Transfer Path 1:
  Comp: forward        | Transn: (Transmit, 4, PktArrival)
      | TFunc: PktSend | SLPM:   (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl            | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend | SLPM:   (Read SrcAddr)(Read TTL)
Failures: 1
  ttl Read TTL: Read of 'TTL' in ttl without previous Write
Warnings: 1
  forward Write NextHopAddr:  Path terminated before this Interface
                              Requirement was matched
```

```
Packet Transfer Path 2:
  Comp: below_bottom     | Transn:(Receive,  1, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: checksum         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: fragment         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: ttl              | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:  (Write TTL)
  Comp: forward          | Transn:(Receive,  4, PktArrival)
      | TFunc: PktSend   | SLPM:  (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl              | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read SrcAddr)
  Comp: fragment         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: checksum         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: below_bottom     | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 2
  below_bottom Read  SrcAddr: 'SrcAddr' has been read previous to
                              below_bottom
  ttl          Write TTL:     Path terminated before this Interface
                              Requirement was matched

Packet Transfer Path 3:
  Comp: below_bottom     | Transn:(Receive,  1, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: checksum         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: fragment         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: ttl              | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:  (Write TTL)
  Comp: forward          | Transn:(Receive,  4, PktArrival)
      | TFunc: PktSend   | SLPM:  (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl              | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read SrcAddr)
  Comp: fragment         | Transn:(Transmit, 3, PktArrival)
      | TFunc: NewPktSend| SLPM:  (Transfer _)
  Comp: checksum         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: below_bottom     | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 2
  below_bottom Read  SrcAddr: 'SrcAddr' has been read previous to
                              below_bottom
  ttl          Write TTL:     Path terminated before this Interface
                              Requirement was matched
```

```
Packet Transfer Path 4:
  Comp: below_bottom      | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: NewPktDeliver| SLPM:
  Comp: ttl               | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)
  Comp: fragment          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 2
  below_bottom Read  SrcAddr: 'SrcAddr' has been read previous to
                              below_bottom
  ttl          Write TTL:     Path terminated before this Interface
                              Requirement was matched
```

```
Packet Transfer Path 5:
  Comp: below_bottom      | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: NewPktDeliver| SLPM:
  Comp: ttl               | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)
  Comp: fragment          | Transn: (Transmit, 3, PktArrival)
      | TFunc: NewPktSend | SLPM:   (Transfer _)
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 2
  below_bottom Read  SrcAddr: 'SrcAddr' has been read previous to
                              below_bottom
  ttl          Write TTL:     Path terminated before this Interface
                              Requirement was matched

Packet Transfer Path 6:
  Comp: below_bottom      | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: ttl               | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write SrcAddr)
Failures: 0
Warnings: 2
  forward Write SrcAddr:  Path terminated before this Interface
                          Requirement was matched
  ttl     Write TTL:      Path terminated before this Interface
                          Requirement was matched
```

```
Packet Transfer Path 7:
  Comp: below_bottom      | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: ttl               | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive, 3, PktArrival)
      | TFunc: DropPkt    | SLPM:   (Write SrcAddr)
Failures: 0
Warnings: 2
  forward Write SrcAddr:  Path terminated before this Interface
                          Requirement was matched
  ttl     Write TTL:      Path terminated before this Interface
                          Requirement was matched

Packet Transfer Path 8:
  Comp: below_bottom             | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver         | SLPM:
  Comp: checksum                 | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver         | SLPM:
  Comp: fragment                 | Transn: (Receive, 6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment                 | Transn: (Receive, 6, PktArrival)
      | TFunc: NewPktDeliver      | SLPM:
  Comp: ttl                      | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver         | SLPM:   (Write TTL)
  Comp: forward                  | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver         | SLPM:   (Write SrcAddr)
Failures: 0
Warnings: 2
  forward Write SrcAddr:  Path terminated before this Interface
                          Requirement was matched
  ttl     Write TTL:      Path terminated before this Interface
                          Requirement was matched
```

```
Packet Transfer Path 9:
  Comp: below_bottom             | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver        | SLPM:
  Comp: checksum                 | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver        | SLPM:
  Comp: fragment                 | Transn: (Receive, 6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment                 | Transn: (Receive, 6, PktArrival)
      | TFunc: NewPktDeliver     | SLPM:
  Comp: ttl                      | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver        | SLPM:   (Write TTL)
  Comp: forward                  | Transn: (Receive, 3, PktArrival)
      | TFunc: DropPkt           | SLPM:   (Write SrcAddr)
Failures: 0
Warnings: 2
  forward Write SrcAddr:  Path terminated before this Interface
                          Requirement was matched
  ttl     Write TTL:      Path terminated before this Interface
                          Requirement was matched

Packet Transfer Path 10:
  Comp: below_bottom     | Transn:(Receive,  1, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: checksum         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: fragment         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: ttl              | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:  (Write TTL)
  Comp: forward          | Transn:(Receive,  4, PktArrival)
      | TFunc: PktSend   | SLPM:  (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl              | Transn:(Transmit, 2, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read SrcAddr)(Read TTL)
  Comp: fragment         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: checksum         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: below_bottom     | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom
```

```
Packet Transfer Path 11:
  Comp: below_bottom     | Transn:(Receive,  1, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: checksum         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: fragment         | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: ttl              | Transn:(Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:  (Write TTL)
  Comp: forward          | Transn:(Receive,  4, PktArrival)
      | TFunc: PktSend   | SLPM:  (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl              | Transn:(Transmit, 2, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read SrcAddr)(Read TTL)
  Comp: fragment         | Transn:(Transmit, 3, PktArrival)
      | TFunc: NewPktSend| SLPM:  (Transfer _)
  Comp: checksum         | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: below_bottom     | Transn:(Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:  (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom

Packet Transfer Path 12:
  Comp: below_bottom     | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: checksum         | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:
  Comp: fragment         | Transn: (Receive,  6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment         | Transn: (Receive,  6, PktArrival)
      | TFunc: NewPktDeliver | SLPM:
  Comp: ttl              | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver| SLPM:   (Write TTL)
  Comp: forward          | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend   | SLPM:  (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl              | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend   | SLPM:   (Read SrcAddr)(Read TTL)
  Comp: fragment         | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: checksum         | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:
  Comp: below_bottom     | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend   | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom
```

```
Packet Transfer Path 13:
  Comp: below_bottom      | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: pre-NewPktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  6, PktArrival)
      | TFunc: NewPktDeliver| SLPM:
  Comp: ttl               | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)(Read TTL)
  Comp: fragment          | Transn: (Transmit, 3, PktArrival)
      | TFunc: NewPktSend | SLPM:   (Transfer _)
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom

Packet Transfer Path 14:
  Comp: forward        | Transn: (Transmit, 4, PktArrival)
      | TFunc: PktSend | SLPM:   (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl            | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend | SLPM:   (Read SrcAddr)
  Comp: fragment       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend | SLPM:
  Comp: checksum       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend | SLPM:
  Comp: below_bottom   | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom
```

```
Packet Transfer Path 15:
  Comp: forward           | Transn: (Transmit, 4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)
  Comp: fragment          | Transn: (Transmit, 3, PktArrival)
      | TFunc: NewPktSend | SLPM:   (Transfer _)
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom
```

Output 2 – Packet Forwarding Stack Filtered Outputs

```
For stack filter-level 2:
 | forward
 | ttl
 | fragment
 | checksum
 | below_bottom

These SLPM were Accessed:

  NextHopAddr
    Read - below_bottom
    Write - forward

  SrcAddr
    Read - below_bottom | ttl
    Write - forward

  TTL
    Read - ttl
    Write - ttl


  _
    Transfer - fragment

Packet Transfer Path 1:
  Comp: forward        | Transn: (Transmit, 4, PktArrival)
      | TFunc: PktSend | SLPM:   (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl            | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend | SLPM:   (Read SrcAddr)(Read TTL)
Failures: 1
  ttl Read TTL: Read of 'TTL' in ttl without previous Write
Warnings: 1
  forward Write NextHopAddr:  Path terminated before this Interface
                              Requirement was matched
```

```
Packet Transfer Path 2:
  Comp: below_bottom      | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: ttl               | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)
  Comp: fragment          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 2
  below_bottom Read  SrcAddr: 'SrcAddr' has been read previous to
                              below_bottom
  ttl          Write TTL:     Path terminated before this Interface
                              Requirement was matched

Packet Transfer Path 3:
  Comp: below_bottom      | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: ttl               | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write SrcAddr)
Failures: 0
Warnings: 2
  forward Write SrcAddr:  Path terminated before this Interface
                          Requirement was matched
  ttl     Write TTL:      Path terminated before this Interface
                          Requirement was matched
```

```
Packet Transfer Path 4:
  Comp: below_bottom      | Transn: (Receive,  1, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: checksum          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: fragment          | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:
  Comp: ttl               | Transn: (Receive,  2, PktArrival)
      | TFunc: PktDeliver | SLPM:   (Write TTL)
  Comp: forward           | Transn: (Receive,  4, PktArrival)
      | TFunc: PktSend    | SLPM: (Write NextHopAddr)(Write SrcAddr)
  Comp: ttl               | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read SrcAddr)(Read TTL)
  Comp: fragment          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom
```

Output 3 – Multicast DVMRP Stack

```
For stack filter-level 3:
 | graft
 | prune
 | spanningtree
 | route_exchange
 | neighbor_discovery
 | ttl
 | fragment
 | checksum
 | below_bottom

These SLPM were Accessed:

  NextHopAddr
    Read  - below_bottom
    NewPktWrite - graft | neighbor_discovery | prune |
                  route_exchange | spanningtree

  SrcAddr
    Read  - below_bottom | ttl
    NewPktWrite - graft | neighbor_discovery | prune |
                  route_exchange | spanningtree

  TTL
    Read  - ttl
    Write - ttl


  _
    Transfer - fragment

Packet Transfer Path 1:
  Comp: graft              | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: prune              | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: spanningtree       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: route_exchange     | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: neighbor_discovery | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: ttl                | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend     | SLPM:   (Read SrcAddr)(Read TTL)
Failures: 2
  ttl Read SrcAddr: Read of 'SrcAddr' in ttl without previous Write
  ttl Read TTL:     Read of 'TTL' in ttl without previous Write
Warnings: 0
```

```
Packet Transfer Path 2:
  Comp: graft              | Transn: (Transmit, 2, Timeout)
      | TFunc: NewPktSend  | SLPM:   (NewPktWrite NextHopAddr)
                                     (NewPktWrite SrcAddr)
  Comp: prune              | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: spanningtree       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: route_exchange     | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: neighbor_discovery| Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: ttl                | Transn: (Transmit, 2, PktArrival)
      | TFunc: PktSend     | SLPM:   (Read SrcAddr)(Read TTL)
Failures: 1
  ttl Read TTL: Read of 'TTL' in ttl without previous Write
Warnings: 1
  graft NewPktWrite NextHopAddr: Path terminated before this
                                 Interface Requirement was matched

Packet Transfer Path 3:
  Comp: graft               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:
  Comp: prune               | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:
  Comp: spanningtree        | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:
  Comp: route_exchange      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:
  Comp: neighbor_discovery | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:
  Comp: ttl                 | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend      | SLPM:   (Read SrcAddr)
Failures: 1
  ttl Read SrcAddr: Read of 'SrcAddr' in ttl without previous Write
Warnings: 0
```

```
Packet Transfer Path 4:
  Comp: graft              | Transn: (Transmit, 2, Timeout)
      | TFunc: NewPktSend  | SLPM:   (NewPktWrite NextHopAddr)
                                     (NewPktWrite SrcAddr)
  Comp: prune              | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: spanningtree       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: route_exchange     | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: neighbor_discovery| Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: ttl                | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:   (Read SrcAddr)
  Comp: fragment           | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: checksum           | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:
  Comp: below_bottom       | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend     | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 0
Warnings: 1
  below_bottom Read SrcAddr:  'SrcAddr' has been read previous to
                              below_bottom

Packet Transfer Path 5:
  Comp: below_bottom       | Transn: (Receive, 1, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: checksum           | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: fragment           | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: ttl                | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:   (Write TTL)
  Comp: neighbor_discovery| Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: route_exchange     | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: spanningtree       | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: prune              | Transn: (Receive, 2, PktArrival)
      | TFunc: PktDeliver  | SLPM:
  Comp: graft              | Transn: (Receive, 1, PktArrival)
      | TFunc: DropPkt     | SLPM:
Failures: 0
Warnings: 1
  ttl Write TTL:  Path terminated before this Interface Requirement
                  was matched
```

Output 4 – Multicast Stack with TTL Removed

```
For stack filter-level 3:
 | graft
 | prune
 | spanningtree
 | route_exchange
 | neighbor_discovery
 | fragment
 | checksum
 | below_bottom

These SLPM were Accessed:

  NextHopAddr
    Read - below_bottom
    NewPktWrite - graft | neighbor_discovery | prune |
                  route_exchange | spanningtree

  SrcAddr
    Read - below_bottom
    NewPktWrite - graft | neighbor_discovery | prune |
                  route_exchange | spanningtree


  _
    Transfer - fragment

Packet Transfer Path 1:
  Comp: graft             | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: prune             | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: spanningtree      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: route_exchange    | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: neighbor_discovery| Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: fragment          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: checksum          | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:
  Comp: below_bottom      | Transn: (Transmit, 1, PktArrival)
      | TFunc: PktSend    | SLPM:   (Read NextHopAddr)(Read SrcAddr)
Failures: 2
  below_bottom Read NextHopAddr: Read of 'NextHopAddr' in
                                 below_bottom without previous Write
  below_bottom Read SrcAddr:     Read of 'SrcAddr' in below_bottom
                                 without previous Write
Warnings: 0
```

Bibliography

[1] Nina T. Bhatti. *A System for Constructing Configurable High-Level Protocols*. Ph.D. Dissertation. University of Arizona Computer Science Department, December 1996.

[2] Nina T. Bhatti, Matti A. Hiltunen, Richard D. Schlichting, and Wanda Chiu. *Coyote: A System for Constructing Fine-Grain Configurable Communication Services*. ACM Transactions on Computer Systems, 1997.

[3] Egon Borger, Robert Stark, and Joachim Schmid. Chapter 2: *Java and the Java Virtual Machine: Definition Verification and Validation*. Springer-Verlag New York, Inc, 2001.

[4] Emmanuel Chailloux, Pascal Manoury, Bruno Pagano. *Developing Applications With Objective Caml*. O'REILLY & Associates, 2000.

[5] Yuri Gurevich. *Sequential Abstract State Machines Capture Sequential Algorithms*. ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111.

[6] Mark Hayden. *The Ensemble System*. Ph.D. Dissertation. Cornell Computer Science Department, January 1998.

[7] Jason Hickey, Nancy Lynch, and Robbert van Renesse. *Specifications and Proofs for Ensemble Layers*. TACAS '99, May 1999, Springer.

[8] Matti A. Hiltunen. *Configurable Fault-Tolerant Distributed Services*. Ph.D. Dissertation. University of Arizona Computer Science Department, July 1996.

[9] Gerard J Holzmann. *Design and Validation of Computer Protocols*. Englewood Cliffs, NJ: Prentice Hall, 1991.

[10] Norman C. Hutchinson and Larry L. Peterson. *The X-Kernel: An Architecture for Implementing Network Protocols*. IEEE Transactions on Software Engineering, 1991.

[11] David Karr. *Specification, Composition, and Automated Verification of Layered Communication Protocols*. Ph.D. Dissertation. Cornell Computer Science Department, January 1997.

[12] Leslie Lamport. *Specifying Systems*. Preliminary Draft, Leslie Lamport, March 2002 http://research.microsoft.com/users/lamport/tla/index.html.

[13] Xavier Leroy. The Objective Caml System Release 3.06. INRIA, France, August 2002 http://caml.inria.fr/ocaml/htmlman/.

[14] Gary Minden, Ed Komp, Stephen Ganje, Magesh Kannan, Sandeep Subramaniam, Shyang Tan, Srujana Vallabhaneni, and Joseph Evans. *Composite Protocols for Innovative Active Services*. DARPA Active Networks Conference and Exposition (DANCE 2002), San Francisco, USA, May 2002.

[15] Robbert van Renesse, Ken Birman, Roy Friedman, Mark Hayden and David Karr. *A Framework for Protocol Composition in Horus*. Proceedings of the 1995 Principles of Distributed Computing, August 1995.

[16] Richard W Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1994.

[17] Gary T. Wong, Matti A. Hiltunen, and Richard D. Schlichting. *A Configurable and Extensible Transport Protocol*. Proceedings of the 20th Annual Conference of IEEE Communications and Computer Societies (INFOCOM 2001), Anchorage, Alaska, (April 2001), pages 319-328.