URL: http://www.elsevier.nl/locate/entcs/volume26.html

Specifying the PLAN Network Programming Langauge

Pankaj Kakkar, Michael Hicks, Jon Moore, and Carl A. Gunter

Department of CIS
University of Pennsylvania
Philadelphia, U.S.A.

Abstract

We discuss how the specification of the PLAN programming language supports the design objectives of the language. The specification aims to provide a mathematically precise operational semantics that can serve as a standard for implementing interpreters and portable programs. The semantics should also support proofs of key properties of PLAN that would hold of all conformant implementations. This paper discusses two such properties. (1) Type checking is required, but interpreters are given significant flexibility about *when* types are checked; the specification must support a clear description of the possible behaviors of a network of conformant implementations. (2) It is essential to have guarantees about how PLAN programs use global resources, but the specification must be flexible about extensions in the network service layer. We illustrate on of kind of issue that will arise in using to specification to prove properties of the network based on the choice of services.

1 Introduction

The aim of this paper is to describe how the PLAN specification supports some of the design goals of the language. The specification itself is available from the web.[1] For purposes of motivation we will need to give some general introduction to the requirements, design, programming model, applications, and implementation of PLAN, but details are best obtained from the project home page.[2] PLAN is part of the SwitchWare Active Network Architecture; the language design was introduced in [8,9] and the overall architecture in [1]. In this paper we discuss two of the primary attributes of PLAN that are interesting for its specification, namely its type system and its model for reasoning about global properties based on available service layer functions. After general discussion of each of these, we describe the specification itself and use some PLAN programs to illustrate the issues.

[1] http://www.cis.upenn.edu/~switchware/PLAN/spec
[2] http://www.cis.upenn.edu/~switchware

Specifying a programming language carefully is a traditional challenge dating back at least to the specification of Algol 60 [4,17], possibly one of the first programming languages to be specified precisely by something different from an implementation. Many of the problems that arose with that specification [12] are still issues now, but there are also some new challenges brought on by the changing role of modern programming languages. Several of these apply to PLAN:

- Programming mobile agents.
- Programming networks.
- Using Domain Specific Languages (DSL's) and scripting languages.
- Proving properties of programs.

PLAN is a *mobile agent* programming language which moves across the network to evaluate on remote hosts. Such languages raise concerns of the kind addressed by the Java security system [18], so the specification of PLAN needs to address many of the same challenges as that of Java language [6] and its virtual machine [15]. As in Java, this has a profound influence on type checking and on the relationship between the language and its 'wire representation', that is, the form in which it is delivered to the remote host. A key decision with PLAN was to follow the Java pattern of specifying both a source-level programming language and a wire language. However, at this point only the source-level language is specified and the wire language is a topic of ongoing research. This gap is resolved in implementations by using parse trees of PLAN programs as a wire representation. This at least allows us to avoid the pitfall that caused some of the early security gaps in Java, namely the fact that transmitted byte code was not necessarily generated by a Java compiler and therefore might not have the guarantees of compiler target code [3].

PLAN is also an *active network* programming language [20,10,11]: its goal is to enable communicators to customize the function of the machines that route packets in a packet switched network. This entails a new set of issues, relating to *global resource control.* In the most common mobile agent application for Java, the web browser applet, a program in Java byte code is 'pulled' from a web page server and evaluated (only) on the machine that requested it. By contrast, a PLAN program can be expected to evaluate on possibly dozens of machines to which it is 'pushed' by a communicator. With such programs being pushed by many parties, it is essential to find ways to control network utilization. The IP protocol suite attempts to protect network resources by a number of means, among which is the very limited interface that IP provides to the routers. Moreover, an IP packet contains a *Time To Live (TTL)* field indicating the maximum number of routers the packet is allowed to visit. This TTL is typically decremented by each router that forwards the packet, and, if it reaches zero, the packet is discarded (with a notification to its source). This protects the network against routing loops that can arise from link and router failures, or buggy routing code. Active network programming environments, which are generally known as *Execution Environments (EE's)*, often provide some analog of the TTL field. However, since the goal of active networks is to provide a richer programming interface to the routers, the challenge of controlling resources is difficult to overcome. It should be possible to see, from the specification, independent of any particular conformant compiler and runtime implementation, how PLAN programs will protect the network.

PLAN is a Domain Specific Language (DSL) for packet-level active network programming. Languages like Java, Algol 60, C++, ML, and so on aim to support a wide range of programming needs. By contrast, there are other languages that support a programming niche. LaTeX [13], for instance, is a successful language for programming typesetting of documents with mathematics; we used it to write this paper. There is a widely-used language called `make` [5] for programming builds from dependencies between software configuration items. AWK is a popular language for writing file processing functions. One probably would not use LaTeX, `make`, or AWK to program a management information system, a video game, or a runtime system, although C++ has been used for these diverse applications. The point is that these DSL's are good for programming in the niches for which they are designed, and make the job easier than a general purpose language would. To regain some generality, DSL's may rely on programs written in other languages. For instance, Perl is a *scripting language,* which means that it is designed to compose or 'glue' functions possibly written in other languages. As such, it is similar to PLAN, which is a scripting language for composing active network functions from the *service layer*. We will discuss issues related to the PLAN service layer later in the paper, but, in summary, properties of PLAN programs depend on the services made available for gluing in PLAN. Hence a specification of PLAN is also the specification of service layer composition. Also, the specification of PLAN must be adequate to support a rely/guarantee reasoning system capable of deriving properties guaranteed of the system as a whole from properties relied upon for the service layer. For example, it can be shown that a PLAN program will terminate if all of the services it invokes terminate and various other properties of the services can be relied upon.

The idea that one can specify a programming language precisely enough to prove mathematical properties of it and its programs is not new. Apparently much of the programming community has considered this to be a matter of secondary importance, since the specification and design of widely-used languages like C make this very difficult. However, with the advent of Java and Standard ML, there has been renewed interest in proving properties from the language specification and thereby ensuring them for conformant compilers and programs. One advance driving this direction is the need to ensure security and the willingness to use *garbage collection* to support a more abstract notion of state. These issues have been taken seriously by the active networking community, for instance, where most projects use Java or some variant of

ML. It was our hope in creating the specification of PLAN that it could be as precise as that of Standard ML [16] and, given the simplicity of PLAN, that it will be easy to understand.

What is the point of specifying PLAN: wouldn't a good implementation be sufficient? Most programming languages are specified so they can be *standardized,* and a typical goal of standardization is to enable portability of programs between different compilers, in particular, between compilers for different machine instruction sets. Since it is feasible to implement PLAN via an interpreter in a high-level language, such portability can be achieved without standardizing PLAN. However, as things have developed, it is typical for PLAN implementations to be in the same language as the service layer they intend to use, and there are PLAN implementions existing or underway for at least five languages[3] at the current time. Some of the goals of the specification are as follows:

(i) Serve as a means to communicate PLAN design ideas between implementors.

(ii) Assist portability of PLAN programs between implementations.

(iii) Enable common tutorials and test suites.

(iv) Provide a basis for describing safety properties of the service layer functions.

(v) Provide a basis for formal verification of PLAN properties.

This paper illustrates some of the challenges in achieving the last two objectives.

2 Static and Dynamic Type Checking

Most of the popular programming languages today employ some kind of type-checking mechanism. Types allow programmers to organize data in meaningful ways; type checking conversely provides guarantees to programmers that their program does not access memory that it is not supposed to. The same guarantee is also helpful to runtime systems.

There have traditionally been two modes of type checking:

Static type checking In the static mode of type checking, a program is checked for type correctness as part of the compilation or loading process, before it is executed. This is the mode employed by ML, Java, and C, for example.

Dynamic type checking In the dynamic mode of type checking, type checking does not happen until an attempt is made by the running program to treat data in a way that does not respect its type. This is the mode employed by the Scheme programming language.

[3] We are aware of work on Java, OCaml, C++, C, and typed assembly language.

There are pros and cons to both approaches. Static type checking allows certain kinds of errors to be detected at compile time. This aids debugging and the runtime system, since both can rely on certain guarantees about type correct code. However, to enable such filtering of errors certain type annotations are required from the programmer. These can become burdensome, and conservative restrictions may mean that programs are rejected by the type checker even if they would run without type errors. Dynamic type checking reduces these problems by using richer runtime information about types, but may allow programs with type errors to be evaluated. Dynamic checks can also be a drag on performance.

PLAN throws up some interesting questions in this regard. Should PLAN programs be statically checked or dynamically checked for type correctness? If static type checking is used, should it be done once before the packet is sent into the network, or at each node where the the program executes? The answer to the second question seems quite obviously to be that PLAN programs have to be type checked on every active router that they are executed on, since routers should not be required to 'trust' one another beyond a limited interface. The first question is more interesting. PLAN programmers would want to make sure that they catch any type errors in the programs they write before the programs actually go out into the network. This means that static type checking would help PLAN programmers. However, in order to protect network resources, PLAN programs are quite special-purpose and do not have looping constructs or recursion. Moreover, only a small part of a PLAN program may be executed on any given router (the examples later will illustrate why this is the case) so some of the performance benefits of static checking are lost. Dynamic type checking probably has better performance for the average PLAN program. In summary, it seems best for PLAN to be capable of static type-checking, but to allow dynamic type checking.

Active routers, in fact, have an entire spectrum of possibilities vis-a-vis type checking available to them, with static and dynamic type checking being the two extremes of this range. Routers could, for example, start the execution of the program with dynamic checking and statically type-check it at the same time, terminating the execution if a type error is found either dynamically or in the static check. Alternatively, one could statically check selected *fragments* of the program that looked like they would benefit from static analysis and check other parts dynamically. This is similar to optimizations for dynamically typed languages that omit runtime checks for parts of the program known to be type correct after an initial analysis phase.

Allowing such a range of possibilites creates new kinds of problems in programs that have effects. One of the ways in which PLAN programs cause effects is the `OnRemote` language primitive, which causes a packet containing a necessary subset of the PLAN program and a first invocation to be sent for execution on a specified network host. Consider the following PLAN program:

```
        fun foo () : unit =
(OnRemote (|foo| (), host1, ...);
 OnRemote (|foo| (), host1, ...);
 OnRemote (|foo| (), host1, ...);
 1 + true)
```

This program first sends three PLAN packets to `host1`, presumably a node in the network, which will all try to execute `foo ()` after getting there. Then it tries to add the integer `1` to the boolean value `true`, generating a type error. Static type checking will catch this error before execution while dynamic type checking will catch this error only after the three packets have been sent. Allowing for a range of alternative modes of type-checking means that any of 0, 1, 2 or 3 packets could have been sent to `host1`.

This offered an interesting challenge when writing the specification of PLAN. The specification needed to be such that static type checking was possible, and yet we needed to specify clearly the behavior when type errors were detected—without over-specifying when or how this is done. Specifiers of statically typed languages have it easy: they can just say how a type-correct program should be evaluated! Fortunately there is a tradition in semantics of giving rules for evaluation of non-type-correct programs so that a theorem can assert that the ill fate that could be encountered by such a program at runtime never happens to a program that is type correct. Our goal is to exploit this approach explicity in the PLAN standard to achieve the desired flexibility at the same time as we avoid the unacceptable option of leaving the behavior of programs with type errors unspecified.

The standard details a way by which static type-checking could be done. However, it also specifies a way of dynamically type-checking a PLAN program during execution, which lies at the other extreme of the range. This execution might give rise to several PLAN packets, which are collected in a multiset during the execution. If the PLAN program contains no type errors, then the execution completes if the services it calls also complete. All PLAN packets collected in the multiset are then sent off into the network. The interesting possibility is when the PLAN program contains type errors. In that case, the execution will terminate abnormally with a type error, and the standard specifies exactly what packets can be generated for transmission. The actual set of packets transmitted will depend on when exactly the type error is caught by a particular router; if the router was employing static type checking, none of the packets would be sent, and, if dynamic, possibly[4] all of them. With any alternative mode of type checking, some allowable subset of these messages will be sent. Thus it is known what messages *can possibly* be sent in the case that the program is not type correct, even if it is not known exactly which ones will be sent. We believe that this is a good constraint, since it errs on the safe side. We will demostrate this further with an example in Section 5. In practice, all of these guarantees are subject to other forms of failure such

[4] PLAN programs can be dropped by the network.

as dropped packets or failing nodes. These forms of failure are not directly modelled in the current specification, but we believe that it would be possible to do so.

3 Network Properties and the Service Layer

PLAN is intended to be used essentially as a scripting language to 'glue' together various library routines, called *services*. This second layer of programming involves another programming language, and in contrast to the packet language, this is intended to be a general-purpose language like Java, OCaml, or even C. Thus it is possible to write much more complex programs at the service layer than at the packet layer.

This extra complexity of programs that can be invoked by a PLAN program on a router makes guarantees about PLAN program behaviour more complex to express. As we mentioned before, a PLAN program that invokes no services is guaranteed to terminate. However, a PLAN program might invoke services (perhaps paying the price with some of its resource bound) that run indefinitely on the router, waiting for other packets to come in or listening for connections, in which case the termination property no longer holds. When making guarantees about PLAN programs it is essential to specify the exact assumptions about the service layer on which these guarantees rely.

The standard specifies exactly what services are part of a minimal PLAN implementation. All of these services are terminating, and none of them can leave state on the router they execute on. The minimal PLAN implementation then has the property that programs executing only on routers that run a minimal implementation will always terminate, will not leave any state on routers, and by implication, cannot communicate with each other within PLAN. They *may* communicate indirectly through higher-level operations at the endpoints that are creating and consuming the PLAN packets. This may seem restrictive, but it is an assumption that IP also imposes on packets.

Protocol designers and even general users may write services that do more than just return a result. For example, running a depth first search in the network might involve leaving a mark on the router to signify that the router has been visited. As we show in the example in Section 5, this can make programs simpler. However, reasoning about the programs and providing guarantees on their behavior now becomes harder. Consider some examples of the kinds of guarantees that might be expected. From the router's viewpoint, a PLAN program that leaves state on the router should not be able to destroy the file system on the router. From the point of view of PLAN packets, another PLAN packet should not be able to meddle with marks left by the former unless this is intended. While the standard does not provide any way of proving such requirements about services, it certainly provides a framework that can help specify exactly the properties that need to be proven, somewhat like verification conditions. In any case, without a precise specification of the

PLAN language semantics, it would be impossible to prove any properties of a network based on even the most precisely specified service layer.

4 An Overview of the Specification

PLAN is a strongly typed, functional scripting language. It provides basic types like integers and strings, simple datatypes like tuples and lists, as well as basic control flow based on conditionals and exceptions. The core of PLAN is its pair of primitives for remote evaluation and the concept of a *chunk* (short for 'code hunk') used to indicate what should be remotely evaluated. Communication consists of a sequence of remote evaluations on successive routers. As a simple language, PLAN has a succinct specification divided into a syntax, type system, and dynamic semantics. The dynamic semantics includes rules for local evaluation, transmission, and dynamic type checking.

4.1 Evaluation model

PLAN programs have a two-phased life cycle consisting of a *transit* phase and an *evaluation* phase. An initial PLAN packet contains a chunk, which encapsulates the program itself with an initial invocation. The packet also contains an *evaluation destination* among other things. In the transit phase, this packet is routed to the evaluation destination. This phase is modeled through a multiset rewriting system. One a packet reaches the evaluation destination it is evaluated, and type checked at any arbitrary time, but no later than the point at which the type error generating expression would have to be evaluated.

4.2 Network model

An IP internetwork consists of hosts on a collection of networks connected by routers. The routers typically have several *interfaces*, each interface connecting to a different network and with a different IP address for each. Packets usually travel through the internetwork with the help of routing tables maintained on the routers; such tables specify the next router to visit to get to a given destination. A PLAN active network (PLANet, [9]) is based on the same philosophy, with the difference that packets now contain programs that need to be executed rather than forwarded or routed.

The PLAN specification models this network through a multiset of *messages*. Each message is of the form `emit`(*dev, packet*), where *dev* is the interface that the *packet* has been *emitted* on. An interface is further modeled as a pair of *hosts*. The PLAN packet is routed to its evaluation destination through a sequence of rewrite rules on the multiset, somewhat analogous to a Chemical Abstract Machine [2]. Here is the rule that makes routing possible, for example:

$$\texttt{emit}((h,h'),(ch,h'',rb,rf)) \mapsto \{\texttt{emit}(dev,(ch,h'',rb-1,rf))\}$$
$$\text{where } (dev, nextHop) = rf(h',h'') \ \text{ if } h' \neq h''$$

To explain this rule without going into too much detail, a PLAN packet containing the chunk *ch*, and intended to be evaluated on h'' is emitted on the interface (h, h'). Since it is not yet at the evaluation destination, the routing function *rf* also specified in the packet is used to figure out the next hop and the interface that the packet needs to go to.

The state of the network changes with each rewrite, with some packets disappearing and others taking their place.

4.3 Static typing

Static type checking in PLAN is done through a proof system that specifies how PLAN programs can be shown to be type correct. A type assertion to be proved by the rules is in one of the following two forms:

$$H \vdash definition\ list \Rightarrow H' \qquad \text{or} \qquad H \vdash expression \Rightarrow \tau$$

Here, H and H' are type environments and τ is a type. The first form of a static typing assertion specifies how a sequence of definitions are turned into a type environment; the second specifies how an expression is given a particular type under the assumptions contained in a given type environment. For example, the following rule specifies how the `OnNeighbor` primitive is typed:

$$\frac{\begin{array}{c} H \vdash exp_1 \Rightarrow \tau\ \texttt{chunkT} \quad H \vdash exp_2 \Rightarrow \texttt{hostT} \\ H \vdash exp_3 \Rightarrow \texttt{intT} \quad H \vdash exp_4 \Rightarrow \texttt{devT} \end{array}}{H \vdash \texttt{OnNeighbor}(exp_1, exp_2, exp_3, exp_4) \Rightarrow \texttt{unitT}}$$

4.4 Evaluation semantics and dynamic typing

The standard also needs to specify the other extreme of the typing mode range as well as how a PLAN program would be evaluated. The specification should make clear exactly:

- what the order of evaluation is, and secondly
- when a type error can be caught at the latest.

To facilitate these goals, and particularly the second, an abstract machine semantics are employed to specify evaluation and dynamic type checking. The abstract machine is an extension of the well-known SECD machine [14]. A state of the traditional abstract machine consists of a stack S, an environment E, a code sequence C and an activation frame stack D. This is extended by adding a multiset M to the state that contains the PLAN packets that are emitted as a result of the evaluation of the program, and some other bookkeeping terms.

The advantage of using an abstract machine semantics is that it breaks evaluation down into the smallest of steps. This facilitates pointing out exactly where during the evaluation of a program the type correctness of a sub-expression is needed; in other words, this specifies exactly the point at which a type error generating sub expression will be caught, if not earlier.

For example, here are the rules that specify how the `OnNeighbor` primitive is evaluated:

$$
\begin{aligned}
&(S, E, \texttt{OnNeighbor}\ (exp_1, exp_2, exp_3, exp_4) :: C, D)\\
&\quad \longrightarrow (S, E, exp_1 :: exp_2 :: exp_3 :: exp_4 :: \textbf{neighbor} :: C, D)
\end{aligned}
$$

$$
\begin{aligned}
&((val_d : \texttt{devT}) :: (val_{rb} : \texttt{intT}) :: (val_h : \texttt{hostT}) :: (val_{ch} : \tau\ \texttt{chunkT}) :: S,\\
&\qquad E, \textbf{neighbor} :: C, D, M, rb)\\
&\quad \longrightarrow ((\texttt{unitV} : \texttt{unitT}) :: S, E, C, D,\\
&\qquad M \cup \texttt{emit}(val_d, (val_{ch}, val_h, val_{rb}, \phi)), rb - \texttt{intVal}(val_{rb}))
\end{aligned}
$$

Here, **neighbor** is a special opcode. The first rule specifies that to evaluate the `OnNeighbor` primitive, the four arguments need to be evaluated first in a left to right order. Once the arguments are evaluated, the results will be on the stack as a typed value pair, at which point their types need to be exactly those specified in the second rule. If so, the machine specifies the result of evaluating the `OnNeighbor` primitive to be `unit` and adds a new message to M. If the types do not match, the machine stops ('hangs') with a type error. Otherwise a chunk val_{ch} with argument val_h is emitted through device val_d with a resource bound bound of val_{rb}. The resource bound assigned to this PLAN packet is decremented from the local resource bound.

When the machine stops with a type error, the set M specifies what messages could have been generated if type checking was in fact left until the last minute. However, a node on the network evaluating the PLAN program might have type checked the offending part of the program earlier, and therefore stopped its execution before all of the messages in M were generated. The specification does not specify exactly what messages could have been left out, however, it does guarantee that even in the presence of a type error in the program, no messages can be generated other than those in M when the modified SECD machine stops with a type error irrespective of when the type error was caught.

5 Examples

We now present two examples of actual PLAN programs. These programs implement a protocol, in two contrasting ways, to look for a particular host

Table 1
Standard Services for First PLAN Program

```
svc print : 'a -> unit
        (* converts given PLAN value into a string
         * and prints it on the console *)
svc getRB : unit -> int
        (* Returns the resource bound left with the packet *)
svc getNeighbors : unit -> (host * dev) list
        (* Returns a list of addresses of neighbors
         * together with the interfaces to reach them *)
svc thisHostIs : host -> bool
        (* Returns true if the given address is of the current node *)
svc getSrcDev : unit -> dev
        (* Returns the interface that the packet arrived on *)
svc length : 'a list -> int
        (* Returns the length of the list *)
svc member : 'a * 'a list -> bool
        (* Returns true if first arg is a member of second arg *)
svc thisHost : dev -> host
        (* Returns the network address of the given interface *)
svc thisHost : unit -> host list
        (* Returns all network addresses of the current node *)
```

in a network and to find routes to it. The network is such that individual nodes have no knowledge of the topology of the network except for knowing who their neighbors are and which interface they are connected through. The objective is for the source to find a route to a given destination without the aid of routing tables maintained by the network. Neither of the protocols we describe is really efficient, but they touch the right issues and do have a similar flavor to those being developed in areas like *mobile ad hoc* networking, where a collection of mobile wireless computers attempt to 'self configure' a network by acting as routers [19].

In our first example, *stateless* route discovery, a route is collected in the packet itself while the packet and its children search for the destination. This program is given in Appendix A and uses the services in Tables 1 and 2. Active packets based on this service interface do not change the state of routers. The services in Table 1 are part of the PLAN standard. Services in Table 2 are not part of the standard yet, but as we gain experience writing more PLAN programs we will be able to decide whether or not to add them to the standard. (Another thing that will be added to the standard is *parametric polymorphism*. A Hindley-Milner style inference system has been implemented for the OCaml version of the interpreter. This will enable a sensible typing of services like `reverse` and `append`.)

In our second example, route discovery *with state*, a route is collected by

Table 2
Additional Services for First PLAN Program

```
svc consfirst : (host*dev) * (host list) -> host list
        (* consfirst(hd,l) is (#1 hd)::l *)
svc reverse : host list -> host list
svc append : host list * host list -> host list
svc getDevtoHost : host -> dev
        (* Given a host address of a neighbor,
         * find the interface to get there *)
```

Table 3
Additional Services for Second PLAN Program

```
svc exists : int -> bool
svc find : int -> host
svc add : int * host -> unit
svc getUniqueKey : unit -> int
        (* Dictionary management services *)
```

leaving ‘bread crumbs’ (backward pointers) on routers as the packet and its children search for the destination. This program is given in Appendix B and uses the services of the first program together with those in Table 3. The services in Table 3 are not part of the standard.

5.1 Stateless route discovery

This program finds all non-looping routes between the source and destination modulo exhaustion of its resource bounds. It works by keeping a list of nodes that have already been visited, and builds a route to the destination as it traverses the network. At each node, the program generates several copies of itself and sends these children to all of its neighbors that have not been visited before. Once it finds the destination, the program uses the route it built while getting there to go back to the source, and reports the route found. It is called a stateless protocol because it leaves no state on the routers as it traverses the network; instead it carries state around with itself as arguments to functions. After reporting the route however, the program tries to add the integer 1 and the `unit` value, generating a type error. We now have several scenarios. If all nodes employ dynamic type checking, the program will run as expected, and no type errors will be detected until *after* the routes get reported. Thus the type error made no difference to the functionality of the program except by generating an extra type error after the result had been reported. If, on the other hand, at least the source node does static type checking, the program will not run at all, since the source will type-check the program before trying to evaluate it, will generate a type error, and then will terminate the program without sending any messges into the network. This is the other extreme. In the case that some nodes, but not the source, employ static type checking, some of the packets may come back and report routes, but no routes will be reported that contain nodes that employ static type checking, since those nodes would not have run the program at all. In particular, if the destination being searched for does type checking statically, no routes will be reported.

There are, of course, several other scenarios. However, the common theme in these and other scenarios is that the worst that can happen is routes not being reported even though they exist - clearly a better scenario than if a type incorrect program generated never-ending messages in the network, or if it caused security loopholes. Hence our assertion that although the specification cannot specify exactly the set of messages generated in the network in the presence of a type error, it restricts the possible set of messages, guarantees that these messages will still be valid PLAN packets (although perhaps containing type-incorrect PLAN programs) and errs on the safe side.

5.2 Route discovery with state

This program works in almost the same way as the previous one, except it keeps track of visited nodes by leaving state on them. This is done by adding a dictionary entry on visited routers that points to the host that the packet came from. When the packet starts at the source, it generates a globally unique integer key by using a service function. Upon arrival at a router that is not the destination, the packet checks the local dictionary for an entry left using the same key. The existence of such an entry implies that the router has been visited before; in which case that packet terminates. If such an entry does not exist, the program sends child packets, as in the first example, to all neighbors. When the program does get to the destination, it goes back to the source using the entries that were left on each router. This program will discover fewer routes than the previous one, but it is more efficient since it avoids revisiting nodes already covered by packets using the same key.

This program uses services that leave state on routers. It thus enables communication between different PLAN programs that know the unique integer key being used to leave state. However, this key could be discovered and used by other PLAN packets (or other network agents) to maliciously corrupt the state being used by the computation. While an obvious solution is to create a new datatype for unforgeable keys, it is worth mentioning that most routing protocols in use in the Internet use analogous techniques, sending passwords in cleartext for example. Of course, what works well enough in the Internet may not work well in an active network. This issue is a topic of ongoing investigation.

5.3 Reasoning about global properties

Aside from efficiency and the resources demanded from the network and nodes by these two protocols, what differences are there between them? A key difference of the kind we would like to use the specification of PLAN and the specification of the services to study is how the network changes as a result of changes in the services. In the first protocol, packets do not communicate and it is therefore possible to show that a packet sent into the network will behave the same regardless of what other packets are in the network. Of course, in practice, there are issues such as whether the presence of heavy traffic causes packets to be dropped, but at the level we are modelling, packets in this protocol are unaffected by other packets. On the other hand, this is not true of interface used by the second protocol. Because packets can communicate by leaving state on routers, packets in the network could interact with this communication state. For instance, a packet may `find` a `host` left by another packet or fail to do so, depending on what another packet did. A more subtle issue arises if packets are allowed to *change* the bindings left at the nodes, as the FBAR protocol in [9] does. In this case labels are used as bread crumbs, but 'configuration' packets are allowed to adjust flow paths dynamically to exploit changes in network traffic. In this case packets not created by the 'owner' of the flow could alter it accidentally or maliciously. In other cases, considerably more power is offered by the interface, such as the ability to change the queueing strategy of a router [9]. The potential interactions between packets becomes more complex in the presence of such a strong interface.

The goal of the specification is to provide a pivot around which it is possible to analyze the effects of the choice of services on the guarantees of the network. An example of the kind of analysis we would like to be able to carry out is given in [7], where the flow of data in the network is analyzed to answer questions like what properties of an agent in the network would allow a cleartext password to be discovered. Ultimately we must go beyond this and understand how to study properties of the network in the presence of cryptographic protections. For example, we need to understand how an interface to authorization may affect network guarantees for a specific protocol, such as labelled routing where changes to labels require cryptographic authentication and authorization. The specification of PLAN is intended to enable this form of analysis.

Acknowledgements

We would like to thank researchers in the SwitchWare group for feedback on the PLAN specification. In particular, Luke Hornof provided insights on the range of possibilities for type-checking strategies. We also thank Peter Hu and Yuan Shi for comments on drafts of the specification. This research was supported by DARPA Contract N66001-96-C-852.

A Stateless Route Discovery

```
(* Function invoked to go back to the source with the route we just found *)
fun goback (remaining:(host*host) list, route:(host*host) list) : unit =
    if (remaining = [])               (* Source reached *)
        then (print (route);          (* Report route *)
              1 + ())                 (* Type error! *)
        else                          (* else keep going *)
            let
                val nextHop:host = #2 (hd remaining)
            in
                (|goback| (tl remaining, route),
                 nextHop, getRB (), getDevtoHost (nextHop))
            end

(* Function invoked to find all possible routes to destination *)
fun find (dest:host, route:(host*host) list, visited:host list):unit =
    (* First argument is destination to be reached *)
    (* Second argument is route we are accumalating *)
    (* Third argument is a list of visited network addresses *)
    let

        (* Get the neighbors' addresses, weed out already visited ones *)
        val neighbors:(host*dev) list = getNeighbors ()
        val neighborHosts:host list =
            foldr (consfirst, neighbors, [])
        fun addIfNew (n:host*dev, l:(host*dev) list):(host*dev) list =
            if (not (member (n, visited)))
                then n :: l
                else l
        val newNeighbors:host list =
            foldr (addifNew, neighbors, [])

        (* Get a list of all of the current node's addresses *)
        val myaddrs = foldr (consfirst, thisHost (), [])

        (* Find out what interface we came in on, will be used to go back *)
        val srcdev:dev = getSrcDev ()

        (* The amount of resource bound each child gets *)
        val childRB:int =
            if (length (newNeighbors) = 0)
                then getRB ()
                else (getRB ()) / (length (newNeighbors))

        (* Send a child packet to given host *)
        fun sendChild (n:host*dev, u:unit):unit =
            OnNeighbor
                (|find| (dest,
                         (thisHost (srcdev), thisHost (#2 n)) :: route,
                         append (myaddrs, visited)),
                 #1 n, childRB, #2 n)
    in
        if thisHostIs (dest)
           then goback (route, reverse (route))
           else foldr (sendChild, newNeighbors, ())
    end

fun startSearch (dest:host):unit = find (dest, [], [])
```

B Route Discovery with State

```
(* Function invoked to go back to the source using the back pointers *)
(* On the way, build up the route to report *)
fun goback (k:int, route:(host*host) list) : unit =
    if (thisHostIs (getSource ()))              (* Source reached *)
        then print (route)                      (* Report route *)
        else                                    (* else keep going *)
            let
                val nextHop:host = find (k)
                val d:dev = getDevtoHost (nextHop)
            in
                OnNeighbor
                    (|goback| (k,
                               (thisHost (d), thisHost (getSrcDev ())) :: route),
                     nextHop, getRB (), d)
            end

(* Function invoked to find all possible routes to destination *)
fun find (dest:host, previous:host, k:int):unit =
    (* First argument is destination to be reached *)
    (* Second argument is the host previously visited *)
    (* Third argument is a key used to store back pointers *)
    let

        (* Get the neighbors' addresses *)
        val neighbors:(host*dev) list = getNeighbors ()
        val neighborHosts:host list =
            foldr (consfirst, neighbors, [])

        (* Find out what interface we came in on *)
        val srcdev:dev = getSrcDev ()

        (* The amount of resource bound each child gets *)
        val childRB:int =
            if (length (newNeighbors) = 0)
                then getRB ()
                else (getRB ()) / (length (newNeighbors))

        (* Send a child packet to given host *)
        fun sendChild (n:host*dev, u:unit):unit =
            OnNeighbor
                (|find| (dest, thisHost (#2 n), k),
                 #1 n, childRB, #2 n)
    in
        if exists (k)                   (* Were we here before? *)
            then ()                     (* If so, die *)
        else if thisHostIs (dest)       (* otherwise, is this our dest? *)
            then goback (k, [])         (* If so, go back *)

        (* Otherwise, check if we're still at source *)
        (* We don't need to add a backpointer if that is the case *)
        else if thisHostIs (getSource ())
            then foldr (sendChild, newNeighbors, ())
            else (addEntry (k, previous);
                  foldr (sendChild, newNeighbors, ()))
    end

fun startSearch (dest:host):unit =
    let
        val k:int = getUniqueKey ()
    in
        find (dest, getSource (), k)
    end
```

References

[1] D. Scott Alexander, William A. Arbaugh, Michael Hicks, Pankaj Kakkar, Angelos Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The switchware active network architecture. *IEEE Network Magazine*, 12(3):29–36, May/June 1998. Special issue on Active and Controllable Networks.

[2] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Annual Symposium of Programming Languages*, pages 81–94, 1990.

[3] Drew Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, April 1997.

[4] P. Naur (Ed.). Report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:299–314, 1960.

[5] Stuart I. Feldman. Make—a program for maintaining computer programs. *Software—Practice and Experience*, 9:255–265, 1979.

[6] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison Wesley, 1996.

[7] Carl A. Gunter and Pankaj Kakkar. Reasoning about secrecy and integrity for active networks. `http://www.cis.upenn.edu/~switchware/papers/secrecy.ps`, July 1999. Submitted to POPL.

[8] Mike Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998. `www.cis.upenn.edu/~switchware/papers/plan.ps`.

[9] Mike Hicks, Jonathan T. Moore, Scott Alexander, Carl A. Gunter, and Scott Nettles. Planet: An active network testbed. `www.cis.upenn.edu/~switchware/papers/planet.ps`, February 1998.

[10] IEEE Computer Society. *IEEE Network: Special Issue on Active and Controllable Networks*, volume 12, 1998.

[11] IEEE Computer Society. *IEEE Computer: Special Issue on Active Networks*, volume 32, April 1999.

[12] D. E. Knuth. The remaining troublespots in ALGOL 60. *Comm. ACM*, 10(10):611–617, 1967.

[13] Leslie Lamport. *LaTeX: A Document Preparation System.* Addison Wesley, 1994.

[14] P. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[15] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[16] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* The MIT Press, 2nd edition, 1997.

[17] P. Naur and M. Woodger (Eds.). Revised report on the algorithmic language ALGOL 60. *Communications of the ACM*, 6:1–20, 1963.

[18] Scott Oaks. *Java Security.* O'Reilly, 1998.

[19] Elizabeth M. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, pages 46–55, April 1999.

[20] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A Survey of Active Network Research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.