Plan in Maude
Specifying an Active Network Programming Language

Mark-Oliver Stehr [1]

Fachbereich Informatik, Universität Hamburg, D-22527 Hamburg, Germany

Carolyn L. Talcott [2]

Computer Science Laboratory, SRI International, Menlo Park, CA 94025, USA

Abstract

PLAN is a language designed for programming active networks, and can more generally be regarded as a model of mobile computation. PLAN generalizes the paradigm of imperative functional programming in an elegant way that allows for recursive, remote function calls, and it provides a clear mechanism for the interaction between host and mobile code. Techniques for specifying and reasoning about such languages are of growing importance. In this paper we describe our specification of PLAN in the rewriting logic language Maude. We show how techniques for specifying the operational semantics of imperative functional programs (syntax-based semantics) and for formalizing variable binding constructs and mobile environments (CINNI calculus) are used in combination with the natural representation of concurrency and distribution provided by rewriting logic to develop a faithful description of the informal PLAN semantics. We also illustrate the wide-spectrum approach to formal modeling supported by Maude: executing PLAN programs; analyzing PLAN programs using search and model-checking; proving properties of particular PLAN programs; and proving general properties of the PLAN language.

1 Introduction

In [24] we have reported on our experience with the rewriting logic [23] language Maude [2] in the context of active networks. In that paper we have included a very brief overview of the application of Maude at two very different levels of the active network infrastructure, namely in the object-oriented specification of the AER/NCA protocol suite and in the specification of the PLAN active network

[1] Email: stehr@informatik.uni-hamburg.de
[2] Email: clt@cs.stanford.edu

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

programming language. In this paper we present the second application in greater depth and with a particular emphasis on the following two aspects: (1) the use of operational semantics techniques from programming language theory augmented with the CINNI explicit substitution calculus; and (2) the wide-spectrum approach to formal modeling supported by Maude. We begin with a brief introduction to active networks and PLAN.

What are Active Networks?

On the web site of the *Switchware Project* [11], a project concerned with the design and the implementation of an active network infrastructure, we find the following explanation:

> Active networks explore the idea of allowing routing elements to be extensively programmed by the packets passing through them. This allows computation previously possible only at endpoints to be carried out within the network itself, thus enabling optimizations and extensions of current protocols as well as the development of fundamentally new protocols.

Active networks are networks with nodes that do not operate according to a fixed scheme (e.g. as conventional routers) but are instead fully programmable and provide *execution environments* for programs that can be received from other nodes via the network. Active networks can be wired, wireless or hybrid networks. One may think of active networks as a generalization of conventional networks and as a step toward greater flexibility: Packets, which are *interpreted* by routers in conventional networks following rigid schemes, become programs, which are *executed* in active networks in a universal fashion. See [31] for a survey of active network research and the recent DARPA conferences on this subject [3,4].

What is PLAN?

The PLAN web site [10] introduces PLAN as a *Packet Language for Active Networks* with the following explanation:

> PLAN is a resource-bounded functional programming language that uses a form of remote procedure call to realize active network packet programming. It is part of the SwitchWare Project.

PLAN [13,12,25,14,19], is an imperative functional language similar to ML, but has a number of additional features, such as remote function execution and resource awareness. Remote function execution, means that functions can be invoked in such a way that the execution does not take place locally but in the execution environment of a different network node. To this end, the function call is treated as a so-called chunk, i.e. as a piece of data, which is transmitted to the destination node by means of a packet. Resource awareness refers to a mechanism which keeps track of computational resources and ensures that all PLAN programs are terminating. In addition, PLAN programs interact with their host nodes through service package interfaces. Basic services include provision of information about local network

topology, local node properties, time, and routing. Other possible services include resident data services for (time-limited) data storage and retrieval.

Our sources for the informal semantics of PLAN included (in addition to conversations with members of the Switchware team) the PLAN specification document [17] and the paper [19] (a fairly detailed description of an operational semantics), an abstract version of PLAN for reasoning about security [18], and the PLAN programmers guide [15]. We have specified a more general language that we call the extended PLAN Language (briefly xPLAN). xPLAN is based on the full call-by-value λ-calculus (also known as λ-calculus with eager evaluation) and unrestricted recursion, whereas the functional core language of PLAN is similar to a first-order fragment of ML but only allows a form of bounded recursion. This generalization leads to a syntactially simpler, more elegant model with many interesting possibilities for mobile code. The official PLAN language maps naturally to a subset of xPLAN defined by simple syntactic restrictions. The main restriction, which ensures termination of PLAN (and corresponding xPLAN) programs, is that recursive calls can only occur inside chunks, and the local or remote invocation of a chunk reduces the computational resources available by at least one unit. Furthermore, forwarding a packet to the next hop consumes one unit so that the standard hop counter scheme to avoid nontermination of routing is subsumed by this concept.

Our specification fully captures the intent of the specifications [17] and [18], but has the benefit of being both formal and executable. Furthermore, as we will illustrate below, this specification can be used at very different levels [5] ranging from execution of test configurations, symbolic search, and model checking analysis to verification of general properties of programs and of the language itself.

2 PLAN in Maude

Our specification is organized in three main parts: syntax; network; and semantics. The syntax part is a fairly direct formalization in Maude of the syntax of xPLAN as an algebraic data type. The network part models basic network concepts such as locations, addresses, connections, and routing, with the minimal detail needed for the PLAN specification. The semantic part is the heart of the matter. The *multilevel concurrency* of active networks is very directly reflected in the computation state which is structured to provide clear boundaries for the scope of effects and information access.

- A network configuration is modeled as a multiset containing nodes and packets.
- With each node we associate a multiset of processes local to the node, which serve as execution environments for programs and can themselves execute concurrently within the node.
- Each process encapsulates the local state of the execution environment together with an abstract reduction machine.

The rules are grouped according to their scope. To specify the abstract machine we use a general approach suitable for functional languages with side-effects which is based on [8,16,22]. The main idea is that the reduction state of the abstract machine is a pair, consisting of a reduction context (i.e. an expression with a hole) and the expression to be reduced in this context. Furthermore, the specification uses the CINNI calculus [28] to specify the binding structure of the language. CINNI is a generic first-order calculus of explicit substitutions that is parametric in the object language and that does not abstract away the names of variables. [3] The specification is considerably simplified by formalizing environments directly as explicit substitutions, thereby eliminating the need to treat environments explicitly in multiple places. [4] It also gives an elegant solution to the subtle problems of binding and environment handling in the context of recursive remote function calls.

2.1 Syntax

The abstract syntax of xPLAN uses CINNI notation for bound variables. Defining (binding) occurrences of variables are represented as identifiers. A referencing occurrence of a variable is written $X\{n\}$ and refers to the n-th defining occurrence of X (counting from the inside and starting with 0). Presupposing a sort `Nat` of natural numbers, and a sort of identifiers `Id`, this is formalized by the declarations:

```
sort Var .
op _{_} : Id Nat -> Var .
```

The sort `Const` contains constants for built-in data objects, and constants for functions, services, etc., which are classified into constructors (sort `Cstr`) and non-constructors (sort `NonCstr`).

```
sorts Cstr NonCstr Const .
subsorts Cstr NonCstr < Const .
ops Nil  Dummy : -> Const .
ops Pair Cons Chunk Foldr : -> Cstr .
ops Foldr Foldl Hd Tl : -> NonCstr .
```

`Foldr` and `Foldl` provide the ability to iterate over a list. In contrast to general recursion using `LetRec` (see below), these two functions provide a form of bounded recursion that is always terminating and hence not charged against the computational resources available to the program. The basic data types of PLAN are modeled by injecting the corresponding Maude sort into the `Const` sort. Thus they are isomorphic to, but not confused with, the Maude sorts. Apart from the standard Maude sorts we presuppose a sort `Addr` of host addresses. The host address is not necessarily unique for a given host, because each host can have several network devices and each of these has an associated host addresss.

[3] This is in contrast to presentations of λ-calculus modulo α-conversion or presentations based on de Bruijn indices. In both of these representations the information about names is lost.

[4] This is in contrast to for instance SECD machines, which carry the environment as an explict component. In the explicit substitution approach environments are not accessible as a machine component but instead implicitly eliminated as soon as possible thanks to the CINNI equations.

```
op Bool_ : Bool -> Const .
op Int_ : Int -> Const .
op String_ : String -> Const .
op Addr_ : Addr -> Const .
op Key_ : Int -> Const .
```

There are constants for each of the service functions. Some examples are given below.

```
ops GetRB GetSource GetSrcDev : -> NonCstr . *** Proc. level
ops ThisHostIs GetNeighbors  : -> NonCstr .  *** Node level
ops OnNeighbor OnRemote : -> NonCstr .   *** Packet creation
ops Exists Get Put  : -> NonCstr .       *** Data Repository
```

The service calls `GetRB()`, `GetSource()`, and `GetSrcDev()` are used to access information about the current process, namely the remaining amount of computational resources, the address of the originating host, and the address of the network device at which the packet arrived that initiated the current process. The service `ThisHostIs`(a) checks whether a given address a refers to a network device local to the current note, and `GetNeighbors()` returns the list of neighbors of the current node. `OnNeighbor`$(chunk, dest, int, dev)$ invokes the given chunk $chunk$ at a neighbor $dest$ using dev as the outgoing device and passes on int of its resource units for sending the packet containing the chunk and for its execution on the remote node. `OnRemote` is similar but allows execution on an arbitrary nodes and hence may involve packet routing by means of a routing function that has to be passed as an additional argument. Finally, `Exists`(str, i), `Get`(str, i), and `Put`(str, i, val, exp) provide access to a resident data dictionary local to the current node, (str, i) being a composite access key, val the value to be stored, and exp the time till expiration.

Note that we use the classification into constructors and non-constructor only for constants that denote functions. It is done in order to identify the subset of the expressions that represent values. Roughly speaking, constants are values and constructors applied to lists of values are values. A non-constructor applied to any list of expressions is a non-value requiring one or more steps of evaluation. Also, a constructor applied to a list containing a non-value is a non-value. In the specification, values and non-values are formalized as sorts `Val < Ex` and `NonVal < Ex`, respectively.

The expressions of the language, are built from constants and variables using typical functional language constructs. The main constructs of xPLAN are given below. [5]

```
sort Ex .
subsorts Const Var < Ex .
op __ : Ex ExList -> Ex .
op If_Then_Else_ : Ex Ex Ex -> Ex .
op Lam`[_:_`]_ : IdList PlanTypeList Ex -> Ex .
```

[5] The are additional constructs for sequential execution and exception handling.

```
op Let`[_=_`]_ : IdList ExList Ex -> Ex .
op LetRec`[_=_`]_ : IdList ExList Ex -> Ex .
```

Note that __ stands for empty syntax, thus function application is represented by juxtaposition of the function expression with the argument list, and the backquote separates lexical tokens in the mixfix declarations. The meaning of these constructs is the standard one of call-by-value λ-caluclus, but function application and λ-abstraction are generalized to arbitrary n-ary functions (so that currying is not needed), and correspondingly a single (recursive) `Let` construct allows several *simultaneous* bindings.

For sake of brevity we have omitted the declarations of the sort `PlanType` of type annotations and the obvious declarations of the sorts `IdList`, `ExList`, `ValList`, and `PlanTypeList`. They denote lists over `Id`, `Ex`, `Val`, and `PlanType`, respectively, with inclusions `Id < IdList`, `Ex < ExList`, `Val < ValList`, `PlanType < PlanTypeList`. We always use a constructor `_,_` for list concatenation. Furthermore, we use a constant `empty-exl` for the empty list over `Ex`, and we extend the inclusion `Val < Ex` to `ValList < ExList`.

2.2 Semantics

To specify the semantics of xPLAN we first explain how the global active network state is represented. We then discuss the reduction machine which is the basis for the operational semantics for the functional programming primitives. Finally, we discuss the transition rules and give representative examples for the main types of transitions.

2.2.1 The Active Network State

The global state of an active network is a configuration modeled as multiset whose elements are nodes, processes, packets, data sets, and a unique global key. The sort and constructor declarations are as follows. We assume sorts `Addr`, `Loc`, `Connection`, `Route` of host addresses, locations, connections (i.e. pairs of the form src `>>` $dest$), routes (i.e. pairs of the form $dest$ `via` con, meaning that $dest$ can be reached via the connection con), and sorts `AddrList`, `Connection List`, `RouteList` of corresponding lists.

```
sort Configuration .
sort Node Packet Process FreshKey Data DataItem .
subsorts  Node Packet Process FreshKey Data < Configuration .
op empty-conf : -> Configuration .
op __ : Configuration Configuration -> Configuration
        [assoc comm id: empty-conf] .
op Node : Loc AddrList ConnectionList RouteList -> Node .
op Packet : Addr Addr Addr Int Int Const
            Val ValList -> Packet .
op Process : Loc Addr Addr Int Int RedState -> Process .
op FreshKey : Int -> FreshKey .
```

```
op Data : Loc DataItemList -> Data .
op DataItem : String Int Val Int -> DataItem
```

A network node has the form $\texttt{Node}(l, devs, nbrs, rt)$. The location l serves as its identifier, $devs$ lists its network devices, $nbrs$ gives the connections to neighbors, and rt is the node's routing table. The network topology is given by the combined device and neighbors information of all of its nodes.

A packet in transit has the form $\texttt{Packet}(dest, fdest, orign, ssn, rb, rf, val, vall)$, where $dest$ specifies the next hop destination address on its route to the final destination $fdest$. Each packet has an originating packet, injected into the network by some application and has assigned a unique session key. ssn is the session key of the originating packet, and $orign$ is the address of the originating application. rb is the amount of computational resources available to the packet for its execution, and rf is the packet's preferred routing function. The final two arguments make up a chunk with function val, and (evaluated) arguments $vall$.

A process has the form $\texttt{Process}(l, orign, ardev, ssn, rb, rs)$. The process was created when a packet with node l as its final destination arrived. The address $ardev$ refers to the device at which the packet entered the node, $orign$, ssn, are the same as in the packet, rb is the remaining amount of computational resources, and rs is the reduction machine state (see below).

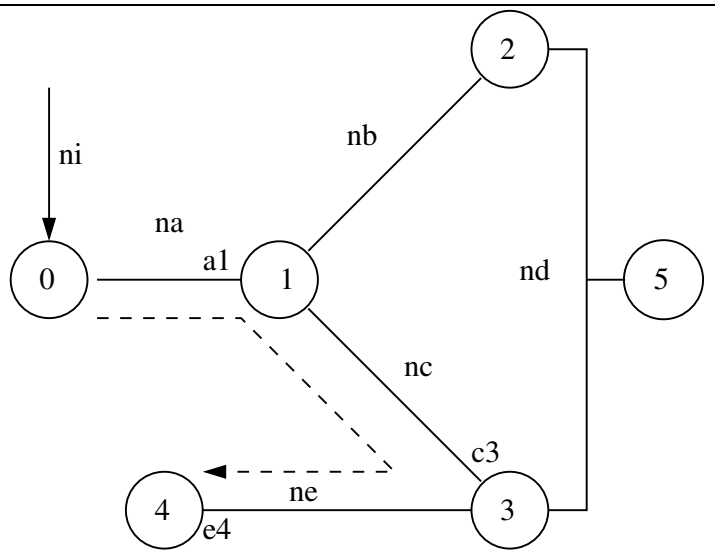
Admissible configurations have a single object of the form $\texttt{FreshKey}(key)$ used to generate fresh keys for sessions and controlled data sharing. The integer key is incremented each time a key is generated.

For the resident data services each node $\texttt{Node}(l, \ldots)$ has an associated data object $\texttt{Data}(l, dil)$ where dil is a list of data items. Data items have the form $\texttt{DataItem}(id, k, val, ttl)$, where (id, k) constitutes a composite key under which the value val is stored. The last argument ttl determines the time until expiration of the data item (present for future compatibility, since time advance is currently not modeled).

In Figure 1 we show an example network topology and a fragment of the definition of the Maude term `example-topology` representing an initial network configuration with this topology. The network has six nodes `l0`,...,`l5`, and six subnets `na`,...,`ne`,`ni`. In the term `example-topology` this topology is expressed by the `AddrList` and `ConnectionList` arguments of the `Node` constructs. The configuration also contains information about the next available fresh key and an initially empty data dictionary associated with each node.

2.2.2 Design of the Reduction Machine

When a packet arrives at its destination node a process is created to execute the invocation encapsulated by the chunk. The local execution of a process is specified by an abstract reduction machine. A simple and concise formalization of the reduction machine is crucial for the semantics to be useful for mathematical reasoning. We have used an approach called syntax-based semantics [8,20,30] to simplify the reduction machine and to obtain a very direct connection between the (partially




```
op example-topology : -> Configuration .
eq example-topology =
     FreshKey(10)
     Node(loc("l0"), (addr("i0"),addr("a0")),
          ((addr("a0") >> addr("a1"))),
          ((addr("a1") via (addr("a0") >> addr("a1"))),
           (addr("b1") via (addr("a0") >> addr("a1"))),
           ...))
     Data(loc("l0"), empty-dil)
     Node(loc("l1"), (addr("a1"),addr("b1"),addr("c1")),
          ((addr("a1") >> addr("a0")),
           (addr("b1") >> addr("b2")),
           (addr("c1") >> addr("c3"))),
          ((addr("a0") via (addr("a1") >> addr("a0"))),
           (addr("b2") via (addr("b1") >> addr("b2"))),
           ...))
     Data(loc("l1"), empty-dil)
     ...  .
```

Fig. 1. Example Topology

executed) program and the machine state. This approach uses (extended) program syntax to represent semantic entities. In particular, values are just a subset of expressions, and the control stack is represented by expressions with holes, called reduction contexts. Environments are represented using explicit substitutions in a suitable instance of the CINNI calculus.

A reduction machine state has the form `RedState`(cx, ex) with a constructor:

```
op  RedState : Cx Ex -> RedState .
```

The reduction context component cx is an expression with a hole and ex is the expression that is the current focus of reduction.

The sort `Cx` contains expressions with any number of holes (including possibly none) in any position in which an expression could occur. Thus expression constructors are overloaded to construct contexts and there is an additional constant `?` to represent the hole:

```
sort Cx .
subsort Ex < Cx .
op '? : -> Cx .
```

Reduction contexts are a special form of contexts in which the holes correspond to positions where evaluation can take place. In the case of PLAN, which has a deterministic evaluation semantics, reduction contexts have a single hole and this hole is not in the scope of any binding operators. Redexes correspond to machine instructions, they can be immediately reduced. In the pure lambda-value calculus the redexes are lambda expressions applied to values: $(\lambda x.e)v$. In PLAN they also include non-constructors applied to value lists and let expressions in which all bindings are value expressions. Mathematical descriptions of deterministic evaluation using reduction contexts are based on a key lemma that says that an expression ex is either a value or it decomposes uniquely into a reduction context R and a redex r such that ex is the result of filling the hole in R with r (written $R[r]$) [8]. The inductive definition of the set of reduction contexts corresponds to peeling off basic reduction contexts one layer at a time until a redex is reached: $ex = R_0[....R_n[r]]$. These basic reduction contexts correspond to a control stack with R_n at the top. For example, the first layer of a PLAN application $ex = val(vall, nval, exl)$, where $vall$ is a value list and $nval$ is a non-value expression, is the reduction context $R = val(vall, \texttt{?}, exl)$ expressing the left to right evaluation order semantics. Most of the action occurs at the inner basic reduction context (top of the stack). For example, suppose the above application fills the hole of an outer reduction context R' so that $ex' = R'[ex] = R'[R[nval]]$. When the evaluation of $nval$ leads to a value val' the hole is filled with that value, and the resulting expression is redecomposed if it still contains a redex. The new decomposition is parametric in the outer reduction context, that is, it has the form $R'[R''[r']]$ where $R''[r']$ is the unique decomposition of $R[val']$.

In the following we use variables ex, ex', etc. to range over expressions (sort `Ex`) and variables cx, cx', etc. to range over contexts (sort `Cx`). The operation of hole filling is a special case of metavariable substitution (the hole being the only metavariable) and is generalized to allow filling of holes with contexts (context composition) and to apply to context lists (sort `CxList`), contexts being a special case. The process of hole filling is formalized by the following operation.

```
op <'?':=_>_ : Cx Cx -> Cx .
op <'?':=_>_ : Cx CxList -> CxList .
eq < ? := cx > ? = cx .
eq < ? := cx > const = const .
eq < ? := cx >(cx' cxl) = (< ? := cx >cx')(< ? := cx > cxl) .
   ...
```

A naive formalization of the reduction machine uses decomposition to determine the next reduction step, followed by hole filling to place the reduct in its context. This involves many operations of hole filling and decomposition. A more efficient formalization uses the observation that the reduction context layers correspond to a stack and represents this stack using a lazy hole filling operator (with no equations for simplification):

```
op  <<'?':=_>>_ : Cx Cx -> Cx .
op  <<'?':=_>>_ : Cx CxList -> CxList .
```

The rules for forming reduction contexts can be easily formalized using membership axioms or by direct construction. However, they are not needed to formulate the reduction rules. They more properly belong to an extension of the executable specification where properties of the semantics are to be proved. For example, the reduction machine maintains two invariants on `RedState`(cx, ex): (1) the cx component is a reduction context; and (2) the entire program (in its current stage of evaluation) is given by `<? := `ex`>` cx, i.e. by filling the hole in cx with the focus expression ex.

Apart from the meta variable substitution used for hole filling, a second notion of substitution is needed in the rules of our specification for object variables. This substitution cannot be reduced to a simple textual substitution, because it must respect the binding structure of the object language. Therefore, we use the CINNI family of explicit substitution calculi [28] instantiated to the syntax of xPLAN. We have slightly generalized the original CINNI substitutions to simultaneous substitutions by simply lifting all operators from `Id` to `IdList` (which represents a simultaneous binding). There is a basic explicit substitution constructor `[_:=_]`, two auxiliary constructors `shift` and `lift`, for relocation (by changing the variable indices), and an operation `__` for application of a substitution to an expression list (expressions being a special case).

```
sort Subst .
op [_:=_] : Id Ex -> Subst .
op [_:=_] : IdList ExList -> Subst .
op [shift_] : Id -> Subst .
op [lift__] : Id Subst -> Subst .
op [lift__] : IdList Subst -> Subst .
op __ : Subst Ex -> Ex .
op __ : Subst ExList -> ExList .

eq ([id := ex] (id{0})) = ex .                          *** C1
eq ([id := ex] (id{suc(m)})) = (id{m}) .                *** C2
eq ([shift id] (id{m})) = (id{suc(m)}) .                *** C3
eq ([lift id S] (id{0})) = (id{0}) .                    *** C4
eq ([lift id S] (id{suc(m)})) =
   [shift id] (S (id{m})) .                             *** C5

eq (S const) = const .                                  *** C6
```

```
eq (S (ex exl')) = ((S ex) (S exl')) .                    *** C7
eq (S (Lam [idl : typel] ex)) =
   (Lam [idl : typel]([lift idl S] ex)) .                 *** C8
...
```

2.2.3 The Transition Rules

The configuration evolves by means of local reduction machine rules and service rules. The latter are further split into process, network, packet, and data service rules.

Reduction machine rules. There are two kinds of reduction machine rules: *control rules* that move the focus to the next relevant redex; and *reduction rules* that perform the actual reductions.

```
rl [args]: RedState(cx, (val (vall', nval', exl')))
    =>
    RedState(<< ? := (val (vall', ?, exl')) >> cx, nval') .

rl [up]: RedState(<< ? := cx >> cx', val)
    =>
    RedState(cx', < ? := val > cx)  .

rl [beta] RedState(cx, ((Lam [idl : typel] ex) vall))
    =>
    RedState(cx, [idl := vall] ex) .
```

The rule `args` is a control rule moving the focus to the next unevaluated argument in a function application. The rule `up` moves the current focus toward the top (viewing the term as a tree) if the current focus is a value. It is the only rule that uses the "non-lazy" version of context hole filling. The rule `beta` corresponds to the standard beta-reduction rule of the call-by-value lambda calculus. To give a flavor of how CINNI handles substitution we show the reduction and simplification of a lambda application (omitting type annotations) using the `beta` rule and the CINNI equations given earlier.

```
((Lam [x] (Lam [x] (x{0} x{1})) ) x{0})
 => ([x := x{0}] (Lam [x] (x{0} x{1}))              *** beta
 = (Lam [x] [lift x [x := x{0}]] (x{0} x{1}) )        *** C8
 = (Lam [x] ([lift x [x := x{0}]]x{0}
             [lift x [x := x{0}]]x{1}))
 = (Lam [x] (x{0} [shift x]x{0}))             *** C1 C2 C4 C5
 = (Lam [x] (x{0} x{1})                                *** C3
```

Thus the original `x{0}` has become `x{1}` to maintain its reference to an external binding.

Process Service rules use information held in the process but outside the reduction machine state. For example, application of `GetRB` returns the resource bound, i.e. the remaining computational resources, of the current process.

```
rl  Process(l, orign, ardev, ssn, rb,
      RedState(cx, (GetRB empty-exl)))
    =>
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Int rb))) .
```

Network service rules use the nodes local network information. For example, the service function `ThisHostIs` checks whether a given address is one of the nodes network devices.

```
rl  Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (ThisHostIs (Addr a))))
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Bool (contains(devs,a))))) .
```

Data service rules manipulate the nodes resident data storage. For example, the service function `Put` adds or updates a data item.

```
rl  Data(l,dil)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (Put ((String str),(Key key),
                         val,(Int ttl)))))
    =>
    Data(l,put(dil,str,key,val,ttl))
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, Dummy)) .
```

Packet rules include rules for *emitting*, *delivering*, and *routing* packets in transit. The PLAN construct `OnNeighbor` is one of the two possibilities to initiate a remote function call which is given by a chunk `Chunk`$(val, vall)$. As we can see below, the execution of `OnNeighbor` leads to the emission of a packet which encapsulates this chunk.

```
crl Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, rb,
      RedState(cx, (OnNeighbor ((Chunk (val,vall)),
                        (Addr dest),(Int int),(Addr dev)))))
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, ardev, ssn, (rb -int),
      RedState(cx, Dummy))
    Packet(dest, dest, orign, ssn, (int - 1), NoRoute,
           val, vall)
    if connection(devs,nbrs,(dev >> dest)) and
       (rb >= int) and (int > 0) .
```

Notice that the current amount of resources `rb` of the executing process is de-

creased by the amount given to the emitted packet, and that amount is then decreased by one corresponding to the use of one unit for the first hop. The routing function component of the packet is set to an irrelevant constant `NoRoute` above, because `OnNeighbor` can only send packets to immediate neigbors. The more general `OnRemote` service allows remote invocation on arbitrary locations and allows the user to specify a routing function which is passed along in the packet.

When a packet reaches its destination (next hop agrees with final destination) a process is created to evaluate the contained chunk.

```
crl Node(l,devs,nbrs,rt)
    Packet(dest, fdest, orign, ssn, rb, rf, val, vall)
    =>
    Node(l,devs,nbrs,rt)
    Process(l, orign, dest, ssn, rb,  RedState(?,(val vall)))
    if (dest == fdest) and contains(devs,dest) .
```

There are also rules to route packets not yet at their destination, termination rules to remove processes that have completed their task, and exception handling rules for generating, propagating and handling runtime exceptions.

3 Using the Maude Specification of PLAN

Spelling out the details in a formal notation forces one to clarify concepts and to make explicit many implicit assumptions, but there is no guarantee that the specification is correct (represents the intended model) or usable. Thus, a specification must be subjected to further examination and tests. Like system requirements, whether or not a formal specification is correct is subjective and cannot be mechanically checked. However, one can derive consequences (predictions) and compare these to observed or desired properties. In this section, which extends a corresponding section in [24], we recall several general properties of PLAN programs and we discuss a specific program and its analysis in some detail.

As part of the validation of PLAN in Maude we proved a number of general properties of PLAN programs implied by the Maude specification:

(t) Termination: Assuming all packets are eventually delivered (fairness), if a packet is injected into the network with a fresh session identifier, then all processes with that session identifier terminate execution with a reduction state having one of the following forms:
  (t1) a non-value purposely left unevaluated in the current specification, such as `Print val`;
  (t2) a non-value that cannot be executed because it would cause a runtime type error.

(ni) Noninterference: Packets injected into a network with no pre-existing (accessible) data elements execute independently—that is, execution of packets with different session identifiers can be considered separately, since the only mechanism for interaction is shared access to data elements.

(r) Resource requirement: For a PLAN program to visit each node of a network by repeatedly sending packets to all neighbors (one to each) it is sufficient to start with $rb > 2w^d$, where d is the diameter—the length of the longest path between nodes, and w is the width—the maximum number of neighbors of any node. To have k units left at every terminal point, it is sufficient to start with $rb > (k+2)w^d$.

The termination results are a bit more general than stated in that they allow, parametrically, for certain extensions of the language. Proofs of these results will appear in a forthcoming paper.

3.1 Testing and Analyzing Particular PLAN Programs

To test the usability of the specification from the programmer's point of view, we selected several PLAN programs and subjected them to a spectrum of formal analysis techniques. The general approach for these exercises was to

(i) represent the program as a Maude term (a simple syntactic modification, which could be automated);

(ii) define a suite of test configurations, each determined by a network configuration and program input—also represented as Maude terms;

(iii) run the test configurations using the Maude interpreter;

(iv) further analyze the possible computations of the test configurations using Maude's search and model checking tools; and

(v) prove properties of interest for arbitrary network configurations and inputs (using ordinary mathematical reasoning based on the formal model).

As a concrete illustration, we will use one of the route finding programs published in [19]. The Maude term for this program is shown in Figure 2. The program has two main functions: `find`, which does a forward search for the node with the destination address, and `goback`, that returns to the source by the inverse route and `Prints` the route found. The forward search, like Hansel and Gretel, drops crumbs to mark the way back, by storing at each node visited a backpointer, i.e. the address of the network device it used when leaving the previous node. When a packet containing an invocation `find-prog-2`$(find-dest)$ is injected at some node in the network with a given destination address $find-dest$, the computation is initialized by determining the address of the starting node (using the `GetSource` service), by generating a fresh key for labeling data (using the `GenerateKey` service), and by an initial call of the `find` function with this information. The network is then flooded with packets which propagate themselves from nodes that have not been previously visited. To this end, the `find` function first uses the resident data service `Exists` to check if an entry associated with the current key exists in the local dictionary of the current node. If this is not the case, the node has not been previously visited. Hence a new entry in the local dictionary under the same key is created using `Put` to store the backpointer. Next it is checked using

```
ops find-prog-2 : Addr -> Ex .

eq find-prog-2(find-dest) =
(LetRec ["goback" = Lam [("k","route")
                         : (TKey, (TList TAddr))]
  (If (ThisHostIs (GetSource empty-exl))
     Then (Print "route"{0})
     Else (Let ["nexthop" = (Get ((String ""),"k"{0}))]
          (Let ["d" = (GetDevToHost "nexthop"{0})]
          (Let ["newroute" = (Cons ("d"{0},"route"{0}))]
            (OnNeighbor
             ((Chunk ("goback"{0}, ("k"{0},"newroute"{0}))),
              "nexthop"{0}, (GetRB empty-exl), "d"{0})))))))]
(LetRec ["find" = Lam [("dest","previous","k")
                       : (TAddr,TAddr,TKey)]
  (If (Exists ((String ""),"k"{0}))
   Then Dummy
   Else ((Put ((String ""), "k"{0},
               "previous"{0}, (Int 200)));
     (If (ThisHostIs "dest"{0})
      Then ("goback"{0} ("k"{0},Nil))
      Else (
        (Let ["neighbors" = (GetNeighbors empty-exl)]
        (Let ["srcdev" =  (GetSrcDev empty-exl)]
        (Let ["childrb" = ... ] *** divide up rb
        (Let ["sendchild" = *** emit a find packet
           Lam [ ("n","u")
               : ((TPair TAddr TAddr),TUnit) ]
           (OnNeighbor ((Chunk ("find"{0},
                         ("dest"{0},(Snd "n"{0}),"k"{0}))),
            (Fst "n"{0}), "childrb"{0}, (Snd "n"{0})))]
        (Foldr ("sendchild"{0},"neighbors"{0},Dummy)) )))))
      )))]
  ("find"{0} ((Addr find-dest), (GetSource empty-exl),
              (GenerateKey empty-exl))) )) .
```

Fig. 2. A PLAN program for route discovery

`ThisHostIs` if the destination has been reached, and if this is the case the route is reported back to the source by calling `goback`, which recursively follows the backpointers until the source is reached and the route can be `Print`ed, which is assembled on the way. Otherwise, the auxiliary function `sendchild` is called in the body of `find` for each neighbor (using `Foldr` to iterate over the list of neighbors), and `sendchild` itself recursively invokes `find` on the given neighbor's address using the `OnNeighbor` construct. The remaining computational resources are equally distributed among all neighbors (the corresponding amount is computed in `childrb`).

As an example execution, we start a process on node `l0` with destination `e4` (the address of the interface of node `l4` on network `ne`).

```
rew example-topology
    Process(loc("l0"), addr("i0"), addr("i0"), 1, 100,
      RedState(?, find-prog-2(addr("e4")))) .
```

The resulting final configuration returns the route `(a1, c3, e4)` (cf. Fig. 1) by `Print`ing the corresponding PLAN list at the starting node. In this final configuration we also see how the data dictionary was used.

```
result Configuration:
    FreshKey(11)
    Data(loc("l0"), DataItem("", 10, Addr addr("i0"), 200))
    Data(loc("l1"), DataItem("", 10, Addr addr("a0"), 200))
    Data(loc("l2"), DataItem("", 10, Addr addr("b1"), 200))
    Data(loc("l3"), DataItem("", 10, Addr addr("c1"), 200))
    Data(loc("l4"), DataItem("", 10, Addr addr("e3"), 200))
    Data(loc("l5"), DataItem("", 10, Addr addr("d3"), 200))
    ...
    Process(loc("l0"), addr("i0"), addr("a0"), 1, 4,
      RedState(?,Print (Cons (Addr addr("a1"),
                       Cons (Addr addr("c3"),
                       Cons (Addr addr("e4"),Nil))))))
```

Given that multiple `find` processes can be executing concurrently, possibly several on the same node, we might ask if one process could overwrite data written by another. This could happen if two processes on a node are both waiting to `Put` data with the same key. Using the Maude search command we find a state reachable from the above initial configuration in which this situation occurs.

```
search [1] i222e4 =>+
cnf:Configuration
Process(l:Loc, src:Addr, idev0:Addr, sn:Int, rb0:Int,
 RedState(cx0:Cx,
  (Put (String "", Key k0:Int, Addr prev0:Addr, Int 200))))
Process(l:Loc, src:Addr, idev1:Addr, sn:Int, rb1:Int,
 RedState(cx1:Cx,
  (Put (String "", Key k1:Int, Addr prev1:Addr, Int 200)))) .
```

`i222e4` is a constant defined to be the above initial state. The infix `=>+` says to search for states reachable after one or more rewrites, and the term on the righthand side is a pattern to be matched. A solution is found with two processes executing on node `l3`, one coming from address `c1` and the other coming from address `d2`.

All of the example runs using Maude's default execution strategy produced a single path from source to destination. We wondered if in general at most one path would be discovered. We used the newly developed Maude model checking capability [6,7] to find a counter-example. The interface to the Maude model checker is embodied in the MODEL-CHECKER module. This module defines syntax for

LTL (Linear Temporal Logic) formulas built over a sort `Prop`. It also introduces a sort `State` and a satisfaction relation $\models$ on states and LTL formulas. To use the model checker, the user defines particular states and propositions and axiomatizes satisfaction on these states and propositions. The LTL semantics lifts satisfaction from propositions to arbitrary LTL formulas. We declared `Configuration` to be a subsort of `State` and define a property `PrintTwice` satisfied by a configuration in which there are two processes printing results on the packet source node. In order to get an answer quickly we used a simple 3-node network. The actual Maude model-checking command was

```
init-a1 |= [] ~ PrintTwice .
```

where `init-a1` starts the `find` program in the 3-node network, and the expression `[] ~ PrintTwice` is a temporal logic formula that is satisfied only if no reachable configuration satisfies the `PrintTwice` property. The model-checker returned a counterexample showing a possible computation in which two distinct paths from source to destination were returned (`Printed`) to the source.

As an example of analysis by mathematical reasoning, we have proved some correctness properties of the program. Informally, these properties are:

(f1) If the find program started at node loc with destination $dest$ prints $path$ at the source, then $path$ is a path from loc to $dest$.

(f2) If the find program started at node loc with destination $dest$ is given sufficient resources and there is a route from loc to $dest$, then eventually there will be at least one process that prints a path at the source node loc.

The proof is simplified by making use of the termination and non-interference properties of PLAN programs stated above. Note that the resources needed in (f2) can be computed using the general result (r). The proof also makes use of a simple form of program specialization that allows one to express a particular PLAN program as a set of rewrite rules that use only the node level service rules of the interpreter. The specialization process itself makes use of a form of "abstract execution" (see also [29]) in the sense that Maude is used to partially execute specifications without assuming a fixed model, which makes the proof entirely independent of e.g. the network topology. The `find` proof is the subject of a forthcoming paper.

4 Conclusions

The formal specification of the PLAN semantics clarifies a number of issues that remain vague or unsatisfactory in the original mathematical specification [17] such as: the scope of names and the notion of binding (in particular in connection with recursive programs), the handling of environments (especially when packets are shipped), the treatment of side-effects in the iterators `Foldr` and `Foldl`, the mechanism of exception handing, and the concurrent and distributed nature of packet execution.

By treating a less restrictive language xPLAN the semantics was simplified

without sacrificing the essential features of PLAN (the proper PLAN subset is characterized by a simple type system). Furthermore, our specification captures the general idea of a programming language for mobile computation based on an imperative λ-calculus with features such as recursive function calls with a simultaneous change of location.

The syntax-based semantics approach has been used to give operational semantics to langages with functional, imperative and/or concurrent features: program equivalance for Scheme-like languages [8,9,26]; program equivalence in actor languages [1,21]; uniform semantics and program equivalence for a family of higher-order imperative languages [30]; interaction equivalence of specification diagrams [27]; and tool for developing operational semantics and interpreters for programming languages [32]. With the exception of [32], these efforts have not developed executable semantics or automated analyses. To the best of our knowledge the the combination of explicit substitutions and reduction contexts is new. It has subsequently been used in a Maude implementation of Specification Diagrams (personal communication from Prasanna Thati).

Furthermore, the unique combination of functional programming and concurrency on different levels makes the PLAN specification an interesting case study for the use of rewriting logic as a unifying semantic framework. On the conceptual level rewriting logic is general enough to bridge the gap between these different aspects, and on the practical level it comes with an efficient implementation in the Maude language, so that our specification is actually an executable prototype of an active network programming environment, which at the same time can serve as a basis for formal theorem proving.

We emphasize that testing a specification as we did is an extremely important part of the process of developing a formal model. In fact, the specification presented here is the second major version. The first version we developed served to clarify many issues and to fill in many gaps. However, it was too low-level, making mathematical analysis overly complex. For example, the reduction machine was based on a state representation similar to the SECD machine used in the paper specification. This involved defining a number of new state constructors and special purpose transition rules just to manipulate these. Also the network model is slightly more abstract compared with the previous version, where network devices were clearly distinguished from their addresses.

Acknowledgments This work has been partially supported by DARPA through Air Force Research Laboratory Contract F30602-97-C-0312, by NSF under grants CCR-9900326 and CCR-9900334, and by Office of Naval Research Contract N00012-99-C-0198. We would like to thank the members of the Switchware team, especially Carl A. Gunter, Pankaj Kakkar, and Jonathan Smith, for the fruitful collaboration, and we are also indebted to José Meseguer for his advice. Last but not least, we appreciate numerous suggestions from the anonymous referees, which considerably improved the present paper.

References

[1] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1997.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *Maude: Specification and Programming in Rewriting Logic*. SRI International, January 1999. `http://maude.csl.sri.com`.

[3] *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*. IEEE, January 2000.

[4] *DARPA Active Networks Conference and Exposition (DANCE)*. IEEE, May 2002.

[5] G. Denker, J. Meseguer, and C. L. Talcott. Formal specification and analysis of active networks and communication protocols: The Maude experience. In *DARPA Information and Survivability Conference and Exposition (DISCEX'00)*, pages 251–265. IEEE, January 2000.

[6] S. Eker. Maude 2.0 alpha release notes, 2002.

[7] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. Submitted for publication, 2002.

[8] M. Felleisen and D. P. Friedman. Control operators, the SECD-machine, and the λ-calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[9] M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[10] C. A. Gunther et al. A packet language for active networks. `http://www.cis.upenn.edu/~switchware/PLAN/`.

[11] C. A. Gunther et al. The switchware project. `http://www.cis.upenn.edu/~switchware/`.

[12] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Network programming using PLAN. In *Proceedings of the 1998 Workshop on Internet Programming Languages (IPL'98), Part of IEEE International Conference on Computer Languages (ICCL'98), Chicago, IL, May 1998*, May 1998. `http://www.cis.upenn.edu/~switchware/papers/progplan.ps`.

[13] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. PLAN: A Packet Language for Active Networks. In *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming, Baltimore, Maryland, September 1998*, pages 86–93. ACM, 1998. `http://www.cis.upenn.edu/~switchware/papers/plan.ps`.

[14] M. Hicks and A. D. Keromytis. A secure PLAN. In S. Covaci, editor, *Active Networks, First International Working Conference, IWAN '99, Berlin, Germany, June 30 – July 2, 1999, Proceedings*, volume 1653 of *Lecture Notes in Computer Science*,

pages 307–314. Springer-Verlag, June 1999. http://www.cis.upenn.edu/~switchware/papers/iwan99.ps. Extended version at http://www.cis.upenn.edu/~switchware/papers/secureplan.ps.

[15] M. Hicks, J. T. Moore, and P. Kakkar. Plan programmers guide for plan version 3.2. http://www.cis.upenn.edu/~switchware/PLAN/docs-ocaml/guide.ps, July 2001.

[16] F. Honsell, I. A. Mason, S. F. Smith, and C. L. Talcott. A Variable Typed Logic of Effects. *Information and Computation*, 119(1):55–90, 1995.

[17] P. Kakkar. The specification of PLAN. http://www.cis.upenn.edu/~switchware/PLAN/spec/spec.ps, 1999.

[18] P. Kakkar, C. A. Gunther, and M. Abadi. Reasoning About Secrecy for Active Networks. In *13th IEEE Computer Security Foundations Workshop (CSFW'00), 3 – 5 July 2000, Cambridge, England, Proceedings*, 2000. http://www.cis.upenn.edu/~switchware/papers/csfw.ps.

[19] P. Kakkar, M. Hicks, J. T. Moore, and C. A. Gunter. Specifying the PLAN networking programming language. In *HOOTS'99, Higher Order Operational Techniques in Semantics Paris, France, September 30 and October 1, 1999, Proceedings*, volume 26 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999. http://www.elsevier.nl/locate/entcs/volume26.html.

[20] I. A. Mason and C. L. Talcott. Programming, transforming, and proving with function abstractions and memories. In *Proceedings of the 16th EATCS Colloquium on Automata, Languages, and Programming, Stresa*, volume 372 of *Lecture Notes in Computer Science*, pages 574–588. Springer-Verlag, 1989.

[21] I. A. Mason and C. L. Talcott. Actor languages: Their syntax, semantics, translation, and equivalence. *Theoretical Computer Science*, 220:409 – 467, 1999.

[22] I. A. Mason and C. L. Talcott. Feferman–Landin Logic. In W. Sieg, R. Sommer, and C.L. Talcott, editors, *Reflections on the Foundations of Mathematics: Essays in honor of Solomon Feferman*, Lecture Notes in Logic, pages 299–344. Association of Symbolic Logic, 2002.

[23] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[24] J. Meseguer, P. C. Ölveczky, M.-O. Stehr, and C. L. Talcott. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In *DARPA Active Networks Conference and Exposition (DANCE), San Francisco*. IEEE, May 2002.

[25] J. T. Moore, M. Hicks, and S. M. Nettles. Chunks in PLAN: Language support for programs as packets. Technical report, Department of Computer and Information Science, University of Pennsylvania, April 1999. http://www.cis.upenn.edu/~switchware/papers/planchunks.ps.

[26] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3/4):287–358, 1993.

[27] S. F. Smith and C. L. Talcott. Specification diagrams for actor systems. *Higer-Order and Symbolic Computation*, 2002. To appear.

[28] M.-O. Stehr. CINNI – A Generic Calculus of Explicit Substitutions and its Application to λ-, σ- and π-calculi. In K. Futatsugi, editor, *The 3rd International Workshop on Rewriting Logic and its Applications Kanazawa City Cultural Hall, Kanzawa Japan, September 18–20, 2000, Proceedings*, volume 36 of *Electronic Notes in Theoretical Computer Science*, pages 71 – 92. Elsevier, 2000. `http://www.elsevier.nl/locate/entcs/volume36.html`.

[29] M.-O. Stehr. Programming, specification, and interactive theorem proving: Towards a unified language based on equational logic, rewriting logic and type theory. Ph.D. thesis, forthcoming, 2002.

[30] C. L. Talcott. Reasoning about functions with effects. In *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1996. `http://www-formal.stanford.edu/MT/96hoots.ps.Z`.

[31] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[32] Yong Xiao, Amr Sabry, and Zena Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher-Order and Symbolic Computation*, 14(4):387–409, 2001.