Specifying an Active Network Programming Language in Rewriting Logic

Mark-Oliver Stehr and Carolyn Talcott

July 29, 2003

In tutorial form we discuss our specification of PLAN, an active network programming language, in Maude, a formal specification language based on rewriting logic and membership equational logic. We will point out a number of challenges that we faced in this enterprise as well as our solutions and how we have formalized them using rewriting logic as a semantic framework.

Since we believe that the underlying paradigm of the PLAN language is interesting in its own right, our approach is based on a fairly general treatment that is in its essential parts independent of the concrete application of active networks for which the language was originally designed. In fact, we think of PLAN more generally as a programming language for mobile programs where the concept of location is abstract and does not necessarily coincide with the physical location in a network.

Furthermore, we feel that the techniques that we employed in this project and their combination, especially the use of rewriting logic, the modelling of a network, the treatment of multi-level concurrency, the use of an abstract reduction machine with explicit contexts, the treatment of bound variables and environments using an explicit substitution calculus, are worth to discuss, since they are equally applicable in many other contexts.

1 What are Active Networks ?

On the website of the *Switchware Project* [13] 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 enviromments* for programs that can be received from other other nodes via the network. An active network does not necessarily have a static topology, so that also unreliable, wireless and hybrid networks are subsumed by this notion. 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 conference on this subject [3].

2 What is PLAN ?

The PLAN website [12] 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.

From a general perspective, one of the central issues of active network research is the development of programming paradigms which provide a suitable level of abstraction to develop active network applications. A particular programming language providing such a new paradigm is PLAN, the Packet Language for Active Networks [15, 14, 26, 16, 22], which has been implemented as part of PLANet, an active internetwork architecture [17]. PLAN is an imperative functional language similar to ML, but has a number of unique additional features, such as remote function execution, resource awareness and anytime typechecking. Remote function execution, means that functions can be applied to arguments so that the execution does not take place locally but in the execution enviromnent 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. Additional possible services include resident data services for (time-limited) data storage and retrieval. For details we refer to the PLAN programmer's guide [18].

3 Overview

We give an executable formal specification of PLAN (in fact we specify a more general language that we call the PLAN frame language) using the lanugage Maude [7, 8] which is based on a version of rewriting logic [24] with a membership-equational sublogic [6]. The specification has been developed in a collaborative effort by Carolyn Talcott (Stanford Univerity), Mark-Oliver Stehr (SRI), Jose Meseguer (SRI), with support by the PLAN group, in particular Carl A. Gunter and Pankaj Kakkar (University of Pennsylvania) in the scope of the DARPA Active Network Program [1] and the ONR Coordinated Research in Adaptive Network Centric Computing [2]. In accordance with a reseach objective which has been first set in [32] our work provides a formal semantics for PLAN. Indeed our specification fully captures the intent of the specifications [20] and [21], but has the benefit of being both rigorously formal and executable. As shown in [30] our specification can be used as an active network model with execution environments for PLAN programs. More generally, a formal executable specification can be used at very different levels [11] ranging from formal modeling and execution via model-checking and narrowing analysis to formal verification. For instance, the use of abstract execution in this context is currently under investigation [30].

By exploiting the capabilities of rewriting logic, the Maude specification enhances the original PLAN specification in several aspects. For instance, the Maude specification directly reflects the concurrent nature of the network, meaning that each node makes only use of local information about the network topology and that the possibility of concurrent execution in different nodes is directly reflected in the specification.

A particularly interesting aspect of the problem and our specification is the presence of *multilevel concurrency*:

1. A network configuration is modeled as a multiset containing nodes and packets.
2. 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.
3. Each process encapsulates the local state of the execution environment together with an abstract reduction machine.

To specify the abstract machine we use a general approach suitable for functional languages with side-effects which is inspired by [10, 19, 23]. 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, i.e. a metavariable) and the expression to be reduced in this context. Furthermore, the specification uses the CINNI calculus [27] to specify the binding structure of the language and to formalize environments as explicit substitutions. A particular benefit of this approach is that environments can always be eliminated, resulting in a specification that does not have to deal with them explicitly at various places. Especially the delicate issue of binding and environment handling in the context of recursive remote function calls turns out to have a very elegant solution. In summary, the use of the abstract reduction machine in connection with CINNI has greatly increased the clarity and reduced the complexity of the specification w.r.t. earlier versions [28, 29], bringing us closer to the goal of a specification which is useful for application programmers, language designers/implementors and formal methods researchers.

4 The PLAN Frame Language

Our specification uses a representation of programs based on an imperative λ-calculus and presupposes that PLAN programs are mapped to this so-called frame language by a simple preprocessing stage. The representational distance between PLAN and the PLAN frame language is very small and indeed we feel that the level provided by the frame language is better suited for purposes of execution and formal reasoning about PLAN programs.

The abstract syntax of the frame language uses CINNI [27] notation (CINNI is a calculus of names and explicit substitutions, more details are given below) for bound variables. 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).

```
sort Var .

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

Furthermore there are constants for built-in data objects, and constants for functions, services, etc., which are classified into constructors and nonconstructors.

Note: We have formalized a representative sample of the official Plan operations. Axiomatizing the missing operations is straightforward and can be done on a by need basis.

```
sort Cstr NonCstr Const .
subsort Cstr < Const .
subsort NonCstr < Const .
```

Literals and Data Constructors:

```
op Bool_ : Bool -> Const .
op Int_ : MachineInt -> Const .
```

```
op String_ : String -> Const .
op Addr_ : Addr -> Const .
op Key_ : MachineInt -> Const .

op  Dummy : -> Const .
op  Pair : -> Cstr .
ops Nil : -> Const .
ops Cons : -> Cstr .
op  Chunk : -> Cstr .
```

Exception Constructors:

```
ops DivByZero ServiceNotPresent HdExcept TlExcept : -> Const .
ops NotEnoughRB HostNotLocal NoRouteEntry NotFound : -> Const .
```

Service Functions:

```
op  Equal : -> NonCstr .
ops Not And Or : -> NonCstr .
ops Plus Minus Mul Div : -> NonCstr .
ops Fst Snd : -> NonCstr .
ops Hd Tl : -> NonCstr .
ops Length Member Append Reverse : -> NonCstr .
ops Foldr Foldl : -> NonCstr .
op  Print : -> NonCstr .
ops ThisHost ThisHostIs : -> NonCstr .
ops GetRB GetSrcDev GetSource : -> NonCstr .
ops GetNeighbors GetDevToHost : -> NonCstr .
ops Eval OnNeighbor  : -> NonCstr .
ops DefaultRoute OnRemote OnRemote' : -> NonCstr .
ops GenerateKey Exists Get Put Delete SetLT SetGT : -> NonCstr .
```

The expressions, i.e. programs of the language, are built from constants and variables (also called atoms) using the constructs given below.

```
sort At Ex .
subsort Const < At < Ex .
subsort Var < At < Ex .
```

Function application with multiple arguments:

```
op __ : Ex ExList -> Ex .
```

Note the `__` stands for empty syntax – i.e. application is represented by juxtaposition of the function expression with the argument list.

Sequential composition:

```
op _;_ : Ex Ex -> Ex .
```

Conditional execution:

```
op If_Then_Else_ : Ex Ex Ex -> Ex .
```

Nonrecursive definitions with simultaneous binding:

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

Recursive definitions with simultaneous binding (used to define recusive and mutually recursive functions):

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

Lambda abstraction with multiple formal arguments:

```
op Lam`[_:_`]_ : IdList PlanTypeList Ex -> Ex .
```

Typing assertion:

```
op _:_ : Ex PlanType -> Ex .
```

Expression with associated exception handler:

```
op Try_Handle`[_`]_ : Ex Id Ex -> Ex .
```

Exception Raising Construct:

```
op Raise_ : Ex -> Ex .
```

In both the lambda abstraction and the typing assertion the type expressions are ignored by the interpreter. They serve as documentation and can be used by the typechecker as hints or constraints.

4.1 Chunks and Remote Function Invocation

The key capability which distinguishes PLAN from a standard functional programming language such as Scheme or ML is the possibility of invoking a function at a location different from the one on which the program is currently executing and hence to access services on the new location. Although similar, the capability of remote function invocation is somewhat different from the remote procedure calls that is widely used in today's imperative programming languages. With remote function invocation the function is executed on a remote location, gaining access to the services of the remote node, but it inherits its current execution environment and carries with it the code for the function invokation. In combination with the recursive let construct this allows us to write recursive mobile programs in a style very similar to the recursive functions we a familar with in functional programming.

Syntactically, remote function invocation is expressed by one of the following built-in services:

1. `OnNeighbor` ((`Chunk` (ex, exl)),
   (`Addr` $destex$), (`Int` $intex$), (`Addr` $devex$))
2. `OnRemote` ((`Chunk` (ex, exl)),
   (`Addr` $destex$), (`Int` $intex$), rf)))

The first argument of each of these is a chunk, i.e. a piece of data which encapsulates a function call. In the chunk ex should evaluate to the function to be executed and exl is the list of arguments to be passed to this function. The argument `destex` determines the location at which the execution should take place. `OnNeighbor` allows only execution on neigbouring locations and `devex` is the outgoing device over which the packet containing the chunk will be emitted. `OnRemote` makes use of the specified routing function `rf` to forward the packet to its destination (possibly via several hops). `intex` determines the part of the current process resources that is given to the emitted packet for carrying out its task, which must account for the fact that for each hop (one in the case of `OnNeighbor`) the amount of resources that remains available for the remote execution is decreased by one.

In addition to these two there is one further noteworthy built-in service that can be used to execute chunks. The service `Eval`, which is used as

`Eval` (`Chunk` (ex, exl)),

just executes the given chunk at the present location and deceases the resources by one.

It is important to point out that our PLAN frame language is more general than PLAN, and as a consequence only a subset of expressions in the sense above correspond to PLAN programs. For instance, in PLAN programs recursive references to functions can only occur inside the chunks which are arguments to `Eval`, `OnNeighbor`, or `OnRemote`, and to ensure termination executing a chunk consumes at least one unit of the resources.

4.2 Berkling's Notation and the CINNI Calculus

The CINNI calculus [27] addresses the general problem of representing higher-order languages with binding constructs in a first-order framework such as membership equational logic [25, 6] or rewriting logic [24].

Traditionally, there are two major approaches to representing an object language with binding constructs in first-order frameworks. We distinguish betweeen representations with names and representations based on de Bruijn indices. Representations with names have the very desirable feature of being close to the object language; their major drawback is that they lack a canonical way to treat names and potential name clashes. Calculi based on de Bruijn indices have the advantage of a canonical representation, but they are more abstract, since information about names is not represented, so that the gap between the object language and its representation is considerable.

A solution that is closely related to de Bruijn's representation [9] but has been developed independently by Berkling [4, 5] is a unification of named and indexed notation. Therefore, we devote the next section to an introduction and motivation of Berkling's representation, which serves as a basis for the CINNI substitution calculus.

Consider the standard treatment of binding constructs, say in the context of the λ-calculus, where α-equivalent terms, i.e. terms that can be transformed into each other by consistent renaming of bound variables, are identified, i.e. not distinguished for essential parts of the metatheory. An obvious first step towards a named representation is to give up this identification that we also refer to as α-equality. Unfortunately, this rather naive approach leads to the following difficulty that we refer to as *accidental hiding*.

Consider for instance the term

$$[X]\ (F\ A\ ([Y]\ (G\ B\ ([X]\ C(X)))))$$

for distinct names X and Y. Assume the subterm $C(X)$ contains X free. Then each free occurrence of X in $C(X)$ is *captured* by the inner lambda abstraction $[X]$, so that the name bound by the outermost lambda abstraction (which is also $[X]$) is hidden from the viewpoint of $C(X)$. Indeed there is no way to refer to the outermost binder within $C(X)$.

Hence, we are faced with the following problem: a calculus without α-equality is not only less abstract, which is an unavoidable consequence of giving up identification by α-conversion, but also, depending on the (accidental) choice of names, visibility of (bound) variables may be restricted. It is important to emphasize that visibility is not restricted in the original calculus with α-equality, since renaming can be performed *tacitly* at any time.

This suggests tackling this general problem by migrating to a more flexible syntax, where we express a binding constraint by annotating each name X with a natural number index i, written $X\{i\}$, that indicates how many `X`-binders should be skipped before we reach the one that $X\{i\}$ refers to. For instance, we write

$$[X]\ (F\ A\ ([Y]\ (G\ B\ ([X]\ C(X\{0\})))))$$

to express that $X\{0\}$ is bound by the inner $[X]$, and

$$[X]\ (F\ A\ ([Y]\ (G\ B\ ([X]\ C(X\{1\})))))$$

meaning that $X\{1\}$ is bound by the outermost $[X]$. To make the language a conservative extension of the traditional notation, we can identify X and $X\{0\}$.

Obviously, there is some similarity to a notation based on de Bruijn indices [9], but notice that there is an essential difference: the index m in the occurrence $X\{m\}$ is *not* the number of binders to be skipped; it states that we have to skip m binders *for the particular name X*, *not* counting binders for other names. Still a formal relationship to de Bruijn's notation can be established: if we restrict ourselves to terms that contain only a single name X, then we can replace each $X\{i\}$ by the index i without loss of information, and we arrive at de Bruijn's purely indexed notation. In other words, if we restrict the available names to a single one, we obtain de Bruijn's notation as a very special case. In this sense, Berkling's representation can be formally seen as a proper generalization of de Bruijn's notation. Pragmatically, however, the relationship to de Bruijn's syntax plays only a minor role, since a typical user will exploit the dimension of names much more than the dimension of indices. Hence, in practice the notation can be used as a standard named notation, with the additional advantage that accidental hiding and weird renamings (see below) are avoided.

In addition to the representation of the object language, another important issue is capture-free substitution, the main operation that is needed for terms of the object language. For a definition of substitution on terms with binding constructs, it turns out to be useful to treat substitutions as first-class citizens, since substitutions usually have to be adjusted as they are propagated through the term they are applied to. Since substitutions receive the same formal status as terms, the calculi that deal with substitution in this way are called *explicit substitution calculi.*

As a further motivation for the CINNI substitution calculus that we present below, consider the following example of a β-reduction step in the traditional λ-calculus with distinct names X and Y, again taking names literally, i.e. not presupposing identification by α-conversion:

$$(([X][Y]\ X)\ Y) \to [Z]\ Y$$

Clearly, the bound variable Y must be renamed to Z, a name different from Y, to avoid capturing of the free variable Y. Unfortunately, there is no canonical choice if all names should be treated as being equal. We call this phenomenon *weird renaming* of bound variables.

Strictly speaking, CINNI is a family of explicit substitution calculi, parameterized by the syntax of the language we want to represent. The generic calculus can be found in [27]. For our concrete application we have instantiated CINNI to the syntax of the PLAN frame language. Furthermore, 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). In this way the following specification has been obtained. Use use the variable S to ranges over substitutions (sort `Subst`).

```
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 .
eq  ([id := ex] (id{suc(m)})) = (id{m}) .
ceq ([id := ex] (id'{m})) = (id'{m}) if id =/= id' .

eq  ([shift id] (id{m})) = (id{suc(m)}) .
ceq ([shift id] (id'{m})) = (id'{m}) if id =/= id' .

eq  ([lift id S] (id{0})) = (id{0}) .
eq  ([lift id S] (id{suc(m)})) = [shift id] (S (id{m})) .
ceq ([lift id S] (id'{m})) = [shift id] (S (id'{m})) if id =/= id' .

eq  ([empty-idl := empty-exl] ex') = ex' .
ceq ([(id,idl) := (ex,exl)] ex') =
    ([id := ex][idl := [shift id] exl] ex') if idl =/= empty-idl .

eq  ([lift empty-idl S] ex') = (S ex') .
ceq ([lift (id,idl) S] ex') =
    ([lift idl [lift id S]] ex') if idl =/= empty-idl .

eq  (S const) = const .
eq  (S (ex exl')) = ((S ex) (S exl')) .
eq  (S (ex ; ex')) = ((S ex) ; (S ex')) .
eq  (S (If ex Then ex' Else ex'')) =
    (If (S ex) Then (S ex') Else (S ex'')) .
eq  (S (Let [idl = exl] ex')) =
    (Let [idl = (S exl)] ([lift idl S] ex')) .
eq  (S (LetRec [idl = exl] ex')) =
    (LetRec [idl = ([lift idl S] exl)] ([lift idl S] ex')) .
eq  (S (Lam [idl : typel] ex)) =
    (Lam [idl : typel] ([lift idl S] ex)) .
eq  (S (Try ex Handle [id] ex')) =
    (Try (S ex) Handle [id] ([lift id S] ex')) .

eq  (S empty-exl) = empty-exl .
ceq (S (ex , exl')) = ((S ex) , (S exl')) if exl' =/= empty-exl .
```

We can now define the explicit substitution version of the β-*rule* by

$$([X]\ N)\ M \rightarrow_{\beta} [X := M]\ N$$

Notice that weird renaming of bound variables as in the previous example is avoided with the new notion of β-reduction. For instance, we have

$$(([X][Y]\ X\{0\})\ Y\{0\}) \rightarrow^{*} ([Y]\ Y\{1\}).$$

5 Formalizing the Semantics of PLAN

5.1 Specifying Concurrent Systems in Rewriting Logic

Rewriting logic is a specification language for structured transition systems (more precisely transition categories, but the additional categorical structure can be ignored for our purposes). In general a specification in rewriting logic consists of two parts:

1. An equational specification of the underlying state space. This is done by a introducing a collection of sorts and their constructors together with their structural equational properties. Typically, there is one sort (called Configuration below) which represents the entire state space. Other sorts are often used to represent local subspaces.

2. The specification of the local state transition rules. These rules are local in the sense that their are rewrite rules to be applied to (typically small) parts of the system's state. All rules are applied modulo the structural equational properties of the state space. Concurrency is reflected by the fact that several rules can be applied concurrently in a single step.

Part (2) of the specification usually requires data sorts and functions which are provided by the equational part of the specification. In the case of Maude the equational part can make use of membership equational logic, i.e. Horn clause logic over equalities (written $M = N$) and sort memberships (writtem $M : s$).

In the following we describe the design of these two parts in our specification of PLAN.

5.2 Part 1: The Active Network State

The overall state of the modelled active network is called a configuration and is specified as follows:

```
sort Configuration .
```

```
subsort Node < Configuration .
subsort Process < Configuration .
subsort Packet < Configuration .
subsort FreshKey < Configuration .
subsort Data < Configuration .

op empty-conf : -> Configuration .
op __ : Configuration Configuration -> Configuration
        [assoc comm id: empty-conf] .
```

In other words, a configuration is a multiset – axiomized as a commutative monoid – and for convenience each element of the sorts `Node`, `Process`, `Packet`, `FreshKey`, and `Data` is viewed as a singleton configuration. The constructors for these sorts and their intended meanings as as follows:

1. Each network node is represented by an object

$$\texttt{Node}(l, devs, nbrs, rt),$$

of sort Node with a constructor

```
op Node : Loc AddrList ConnectionList RouteList -> Node
```

which specifies the location l (sort `Loc`), which serves as a node identifier, the network devices of the node $devs$ (sort `AddrList`), all connections to neighbors $nbrs$ (sort `ConnectionList`), and the node's routing table rt (sort `RouteList`). Altogether all nodes in the configuration contain the information about the network topology.

2. Each process on a node is represented by an object

$$\texttt{Process}(l, orign, ariv, ssn, rb, \texttt{RedState}(cx, ex)),$$

of sort Process with a constructor

```
op Process : Loc Addr Addr MachineInt MachineInt
             RedState -> Process
```

which specifies the location l (sort `Loc`), i.e. the node on which it is running, the originator address $orign$ (sort `Addr`), i.e. the address of the application which injected the program (more precisely the initiating packet) into the network, the arrival device of program $arriv$ (sort `Addr`), i.e. the device at which it entered the current node, a session key ssn (sort `MachineInt`), which is unique to each injected program, the resource bound rb (sort `MachineInt`), which specifies the remaining resources, and most importantly the reduction machine state `RedState`(cx, ex) (sort `RedState`), which contains the program (which may already be partially executed) and its current continuation state. Notice that there can be an arbitrary number of processes running at a single location, i.e. associated with a network node.

3. Each packet in transit between nodes is represented by an object

$$\texttt{Packet}(dest, fdest, orign, ssn, rb, rf, val, vall)$$

of sort `Packet` with a constructor

```
op Packet : Addr Addr Addr MachineInt MachineInt Const
            Val ValList -> Packet
```

which specifies the immediate destination address *dest* (sort `Addr`) of the packet, the final destination address *fdest* (sort `Addr`), i.e. the address of the next hop, the originator address *orign* (sort `Addr`), i.e. the address of the application which injected the program into the network, a session key *ssn* (sort `MachineInt`), the resource bound *rb* (sort `MachineInt`), the routing function *rf* (sort `Const`), and the information of a chunk which contains an invocation, i.e. a function *val* (sort `Val`) with (evaluated) arguments *vall* (sort `ValList`).

4. A generator of fresh keys is represented by an object

$$\texttt{FreshKey}(key)$$

of sort `FreshKey` with a constructor

```
op FreshKey : MachineInt -> FreshKey .
```

To ensure uniquenes of keys there should be only one such object in the configuration.

5. For the resident data services each node has an associated data object

$$\texttt{Data}(l, dil)$$

of sort `Data` with a constructor

```
op  Data : Loc DataItemList -> Data .
```

which specifies the physical location *loc* (sort `Loc`), a list of data objects *dil* (sort `DataItemList`) which is essentially an association list. The items in this list are constructed using

```
op DataItem : String MachineInt Val MachineInt -> DataItem
```

where the first two arguments (sort `String` and `MachineInt`) constitute a key under which the the third argument (sort `Val`) is stored. The last argument determines the time until expiration of the data item (time advance is currently not modelled in the specification).

Further Remarks:

In the specification we identify each network device of a node with its network address (for example an IP address). As a consequence the PLAN types `Host` and `Dev` are represented by the type `Addr` of addresses.

At moment there is no neighbor discovery mechanism or means for the set of neighbors to change. It shouldn't be hard to add this if/when it becomes interesting to do so.

5.3 Part 2: The Transition Rules

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 specfied by an abstract reduction machine which explicitly keeps track of the context in which an expression is reduced.

A function invocation is executed on an abstract reduction machine in the context of information contained in the packet (such as resource bound, source, address of arrival interface ...). All this information constitutes the state of a process (see above). The machine state, which is part of the process, is an object

$$\mathtt{RedState}(cx, ex)$$

of sort `RedState` with a constructor

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

which specifies the (functional) reduction context and the expression to be reduced. The purpose of the reduction machine is to perform execution of programs in a controlled way, since the order of execution is essential in an language with side-effects like PLAN. (note for instance, the resident data services provide a notion of storage).

The configuration evolves by means of the following rules:

1. **Reduction machine rules** for the interpretation of functional programs (rules that use only the reduction machine state). There are two kinds of reduction machine rules:

   (a) **Control rules**, which focus on the next relevant redex and update the context accordingly.
   Example: A function is applied to a list of arguments. Arguments should be evaluated from left to right, i.e. the first nonvalue argument becomes the new focus.

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

(b) **Reduction rules**, which perform the actual reduction without changing the context.

Example: A λ-abstraction is applied to a list of evaluated arguments.

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

In the righthand side of the rule `[idl := vall]` represents a substitution (see Section 4.2) mapping elements of idl (an `IdList`) to corresponding to corresponding elements of *vall* (a `ValList`).

Among the reduction rules we also have rules for exception handling, i.e. for propagating and catching exceptions.

Furthermore, amnong the reduction rules we have the *local service rules* such as the following for the local service `Hd`.

```
rl  RedState(cx, (Hd (Cons (val,val'))))
    =>
    RedState(cx, val) .
```

This service is local in the sense that the rule refers only to the information in the reduction state. *Nonlocal service rules*, which are discussed next, can make use of additional information, e.g. information about the process or about the network.

2. **Process Service rules** (rules that aditionally use the information in the process).

   Example: The service function `getRB` returns the resource bound of the current process.

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

3. **Network service rules** (rules that additionally use the information of the accociated node).

   Example: The service function `ThisHostIs` checks if a given address is a local address (i.e. a network device) of the node on which the current process is running.

```
rl  Node(l,devs,nbrs,rt)
    Process(l, orign, ariv, ssn, rb,
      RedState(cx, (ThisHostIs (Addr a))))
```

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

4. **Data service rules** (rules that additionally use the resident data storage associated with the process).

   Example: The service function `Delete` removes a data item from the store associated with the current process.

```
crl Data(l,dil)
    Process(l, orign, ariv, ssn, rb,
      RedState(cx, (Delete ((String str),(Key key)))))
    =>
    Data(l,delete(dil,str,key))
    Process(l, orign, ariv, ssn, rb,
      RedState(cx, Dummy))
    if contains(dil,str,key) .
```

5. Rules for **emitting**, **delivering**, and **routing** packets in transit:

   Example: The service function `OnNeigbor` emits a packet.

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

   The current resource of the executing packed (`rb`) is decreased 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.

   Example: A packet delivered to a node may become a process.

```
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) .
```

> Delivery rules apply to packets whose next hop address is the destination address. In this case a process is created on the destination node to evaluate the packet. Routing rules apply to packets not yet at their destination.

Further Remarks: The semantics has been designed to allow for maximal concurrency and laziest type-checking. Other execution/implementation strategies can be obtained by defining Maude strategies at the meta level, leaving the basic specification to declare what could be possible.

5.4 Design of the Reduction Machine

As explained above the reduction machine state is a core part of each process and as we can see from is constructor

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

a reduction machine state

$$\texttt{RedState}(cx, ex)$$

has two components, a reduction context cx, which is an expression with a hole and the potential redex ex, a expression which represents the current focus of reduction.

Contexts are specified in full anaolgy to expressions, but they have `?` as an additional constant to represent the hole:

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

In the following we use variables ex, ex', ex'', etc. to range over expressions (sort `Ex`) and variables cx, cx', etc. to range over to range over contexts (sort `Cx`).

The reduction machine maintains the invariant that the entire program (in its current stage of evaluation) is given by $cx[ex]$ which we write as `<? :=` ex `>` cx in Maude, i.e. cx with the hole replaced by ex.

We have axiomatized the sort `Cx` to include expressions with any number of holes (including possibly none), in any position in which an expression could occur, corresponding to the traditional notion of context used in the lambda calculus. Reduction contexts contain exactly one occurrence of the hole and this occurs only in the position where execution can take place next. This subsort can be axiomatized by membership equations or defined directly by further overloading the expression constructions. One can then show that the contexts arising in reduction states are reduction contexts. This additional information is essential for proving properties of the Plan semantics and of Plan programs, but is irrelevant as far as the interpreter is concerned.

Earlier we have encounted substutition of object language variables which are defined using CINNI. Viewing a hole as a metavariable, we are concerned with a second kind of substitution here. Hole filling is exactly metavariable substitution (there is only one metavariable called a hole), which is nothing else then straightforward textual substitution (there are no binding constructs for metavariables in the object languages). Furthermore, it is clear that holes can be not only filled by expressions but also by other contexts (giving a notion of context composition). As a consequence we have the following specification fragment (where we also lifted subsitutions to lists of contexts):

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

eq  < ? := cx > ? = cx .
eq  < ? := cx > const = const .
eq  < ? := cx > var = var .
eq  < ? := cx > (cx' cxl'') =
    ((< ? := cx > cx') (< ? := cx > cxl'')) .
eq  < ? := cx > (cx' ; cx'') =
    ((< ? := cx > cx') ; (< ? := cx > cx'')) .
eq  < ? := cx > (If cx' Then cx'' Else cx''') =
    (If (< ? := cx > cx')
     Then (< ? := cx > cx'') Else (< ? := cx > cx''')) .
eq  < ? := cx > (Let [idl = cxl'] cx'') =
    (Let [idl = < ? := cx > cxl'] (< ? := cx > cx'')) .
eq  < ? := cx > (LetRec [idl = cxl'] cx'') =
    (LetRec [idl = < ? := cx > cxl'] (< ? := cx > cx'')) .
eq  < ? := cx > (Lam [idl : typel] cx') =
    (Lam [idl : typel] (< ? := cx > cx')) .
eq  < ? := cx > (Try cx' Handle [id] cx'') =
    (Try (< ? := cx > cx')  Handle [id] (< ? := cx > cx'')) .
eq  < ? := cx > ( cx' : type) =
    ( (< ? := cx > cx') : type ) .
eq  < ? := cx > empty-exl =  empty-exl .
ceq < ? := cx > (cx' , cxl'') =
    ((< ? := cx > cx') , (< ? := cx > cxl''))
    if cxl'' =/= empty-exl .
```

We found that is is rarely necessary to carry out metavariable substitution to keep track of the context. In fact, nearly all rules use the "lazy" versions of these operators which do not have any associated equations.

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

These "lazy" operators reflect the view of the reduction context as a stack, a view that is exploited by the rules for exception handling discussed below.

The only rule which uses the "nonlazy" version is the following control rule. Its purpose is to move the current focus toward the top (viewing the term as a tree) if the current focus is a value, i.e. cannot be further reduced.

```
rl  RedState(<< ? := cx >> cx', val)
    =>
    RedState(cx', < ? := val > cx)  .
```

As in the previous rule we make frequent use of a systematic classification of expressions into two disjoint sorts: values (sort `Val`) and nonvalues (sort `NonVal`). The main purpose is to organize the divison of labour between control and reduction rules (examples will be discussed below). Nonvalues are by definition those expressions which can be further reduced by reduction rules, or correspond to runtime exceptions. An expression may be a nonvalue because it contains a reducible subterm, but on the other hand there are values (e.g. λ-abstractions, chunks) which may contain nonvalues as subterms.

```
sort Val NonVal .

mb (Bool bool) : Val .
mb (Int int) : Val .
mb (Lam [idl : typel] ex) : Val .
mb (cstr vall) : Val .

mb (val (exl,nval,exl')) : NonVal .
mb (nval exl) : NonVal .
mb (ncstr exl) : NonVal .
mb ((Lam [idl : typel] ex) exl) : NonVal .
mb (ex ; ex) : NonVal .
mb (If ex Then ex' Else ex'') : NonVal .
mb (Let [idl = exl] ex') : NonVal .
mb (LetRec [idl = exl] ex') : NonVal .
mb (Try ex Handle [id] ex') : NonVal .
mb (ex : type) : NonVal .
```

In the following we use variables $nval$, $nval'$, $nval''$, etc. to range over non values (sort `NonVal`) and variables val, val', val'', etc. to range over values (sort `Val`).

We have already mentioned the some rules for function application as an example in the previous section. There are two control rules stating that if the focus is a function application ($ex\ exl'$) then we move the focus to the leftmost nonvalue.

```
rl  RedState(cx, (nval exl'))
    =>
    RedState(<< ? := (? exl') >> cx, nval) .

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

The actual beta-reduction rule is

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

There are further rules for the application of built-in functions such as

```
rl  RedState(cx, (Plus ((Int int),(Int int'))))
    =>
    RedState(cx, (Int _+_(int,int'))) .

op  length : PlanList  -> PlanInt .
eq  length(Nil) = (Int 0) .
ceq length(Cons (val, pl) ) = (Int _+_(1,int) )
    if (Int int) := length(pl) .

rl  RedState(cx, (Length pl))
    =>
    RedState(cx, length(pl)) .
```

In most cases build-in functions are mirrored by corresponding functions in Maude which are either provided by the library (such as `_+_`) or defined equationally (such as `length`).

A notable exeption are the well-known functions `foldr`/`foldl` for list iteration which due to the possibility of side-effects in the iteration step do not have equationally defined counterparts.

```
rl  RedState(cx, (Foldr (val,Nil,val'')))
    =>
    RedState(cx, val'') .

crl RedState(cx, (Foldr (val,(Cons (val0',val1')),val'')))
    =>
    RedState(cx, (val (val0', (Foldr (val,val1',val'')))))
    if (Cons (val0',val1')) : PlanList .
```

```
rl  RedState(cx, (Foldl (val,val',Nil)))
    =>
    RedState(cx, val') .

crl RedState(cx, (Foldl (val,val',(Cons (val0'',val1'')))))
    =>
    RedState(cx, (Foldl (val,(val (val',val0'')),val1'')))
    if (Cons (val0'',val1'')) : PlanList .
```

In complete analogy to the rules for application of λ-abstractions given above we have the rules for the nonrecursive let construct. As before, the control rule ensures that the focus is moved to the leftmost nonvalue.

```
rl  RedState(cx,
             Let [idl = (vall', nval', exl')] ex'')
    =>
    RedState(<< ? := Let [idl = (vall', ?, exl')] ex'' >> cx,
             nval') .
```

And the reduction rule for let shows that a let with evaluated definitons exactly corresponds to a substitution.

```
rl  RedState(cx, Let [idl = vall'] ex'')
    =>
    RedState(cx, [idl := vall'] ex'') .
```

A more interesting rule is the following reduction rule for recursive let (the control rule is standard). On the left hand side it uses the `LetRec` operator lifted to lists of expressions in the obvious way.

```
rl  RedState(cx, LetRec [idl = vall'] ex'')
    =>
    RedState(cx, [idl := (LetRec [idl = vall'] vall')] ex'') .
```

Compared with the previous reduction rules for nonrecursive let the only difference is that the `LetRec` environment of *vall′* is preserved, which is important since vall' typically contains λ-abstractions with recursive references. The intuitive idea is that whenever we have to reduce a LetRec expression it is unfolded precisely once. In the PLAN frame language there is no upper bound on the number of such unfolding steps for a given `LetRec` construct, so that we can express general resursion giving rise to possibly nonterminating programs.

In PLAN programs, however, there are syntactic restrictions on recursive references which ensure that recursive calls always coincide with a decrease of resources. As a consequence, all PLAN programs are terminating (a more precise analysis of the termination property will be given in a separate document [?]).

The next construct of interest is `If_Then_Else_` which evaluates only the condition as we can see from the following control rule.

```
rl  RedState(cx, (If nval Then ex' Else ex''))
    =>
    RedState(<< ? := (If ? Then ex' Else ex'') >> cx, nval) .
```

If the condition evaluates to either true or false only one the branches is evaluated as expressed by the two reduction rules:

```
rl  RedState(cx, (If (Bool true) Then ex' Else ex''))
    =>
    RedState(cx, ex') .

rl  RedState(cx, (If (Bool false) Then ex' Else ex''))
    =>
    RedState(cx, ex'') .
```

In case of sequential composition the machine always evaluates the leftmost expression (control rule) and ignores the resulting value (reduction rule).

```
rl  RedState(cx, (nval ; ex'))
    =>
    RedState(<< ? := (? ; ex')>> cx, nval) .

rl  RedState(cx, (val ; ex'))
    =>
    RedState(cx, ex') .
```

The rules for the `Try_Handle[_]_` construct show that the construct does not have any effect in the nonerror case. The control rule and the reduction rule for the nonerror case are as follows.

```
rl  RedState(cx, (Try nval Handle [id] ex'))
    =>
    RedState(<< ? :=  (Try ? Handle [id] ex') >> cx, nval) .

rl  RedState(cx, (Try val Handle [id] ex'))
    =>
    RedState(cx, val) .
```

To deal with the error case, a special form of reduction rules, namely *exception handling rules*, come into play. If an exception is raised the exception is propagated untill the next encompassing `Try_Handle[_]_` construct, which finally catches the exception, is reached. In this process, nonrelevant constructs are skipped by the *exception propagation rules*:

```
rl  RedState(cx, ((Raise eval) exl))
    =>
    RedState(cx, (Raise eval)) .

rl  RedState(cx, (val (vall,(Raise eval),exl)))
    =>
    RedState(cx, (Raise eval)) .

rl  RedState(cx, (Raise eval) ; ex)
    =>
    RedState(cx, (Raise eval)) .

rl  RedState(cx, (If (Raise eval) Then ex Else ex'))
    =>
    RedState(cx, (Raise eval)) .

rl  RedState(cx, ((Raise eval) : type))
    =>
    RedState(cx, (Raise eval)) .
```

After a possible propagation of the exception, the evaluation proceeds with the next surrounding exeption handler `[id] ex'`, which is treated similar to a λ-abstraction, the formal parameter being substituted by the exception raised. This is done by the *exception catching rule*:

```
rl  RedState(cx, (Try (Raise eval) Handle [id] ex'))
    =>
    RedState(cx, [id := eval] ex') .
```

We conclude this sections with some sample runs to give an idea how the reduction machine operates with concrete programs and how Maude can be used to execute the specification. More interesting examples of distributed PLAN programs operating in a network will be given in the next section.

As long as they don't contain any service calls programs can be executed by the machine without interacting with the remaining part of the configuration. For instance, a sample run of a program defining the factorial function looks as follows:

```
rew RedState(?,
      (LetRec ["fact" = Lam ["x" : TInt]
                 (If (Equal ("x"{0},(Int 0)))
                  Then (Int 1)
                  Else (Mul ("x"{0},
                          ("fact"{0} (Minus ("x"{0}, (Int 1)))))))]
       ("fact"{0} (Int 3)))) .
```

```
rewrites: 1469 in 10ms cpu (20ms real) (146900 rewrites/second)
result RedState: RedState(?, Int 6)
```

This program is more general than the programs admitted by the PLAN language, since the recursive reference to `fact` does not occur inside a chunk. However, we could instead use the following similar program:

```
rew Process(loc("l1"), addr("i1"), addr("i1"), 1, 100,
      RedState(?,
       (LetRec ["fact" = Lam ["x" : TInt]
                  (If (Equal ("x"{0},(Int 0)))
                   Then (Int 1)
                   Else (Mul ("x"{0},
                     (Eval (Chunk ("fact"{0},
                                  (Minus ("x"{0}, (Int 1)))))))))]
        ("fact"{0} (Int 3))))) .

rewrites: 1805 in 10ms cpu (20ms real) (180500 rewrites/second)
result Process: Process(loc("l1"), addr("i1"), addr("i1"), 1, 97,
                  RedState(?, Int 6))
```

Notice that the surrounding process is needed for the execution, since `Eval` consumes resources which are part of a process.

6 Examples

In the following we present two examples taken from [22] representing different solutions to the problem of finding paths in a network. To give running examples we first introduce a concrete network, in which these programs can be injected.

6.1 Example of a Network Topology

In Fig. 1 we have depicted an example topology consisting of six nodes `l0`,...,`l5` and six subnets `na`,...,`ne`,`ni`, where `ni` will be used to inject programs. Since each node has at most one network device leading to a given network, each host address is just a combination of the name of the subnet and the node.

The topology is expressed by the `Node` constructs in the following configuration which also contains information about the next available fresh key and an empty data dictionary associated with each node. Each node also contains its local routing table, but this information is not used in the two examples given below, because packets are send directly to specific neighbors using `OnNeighbor` rather than routed through the network (which could be done using `OnRemote`).

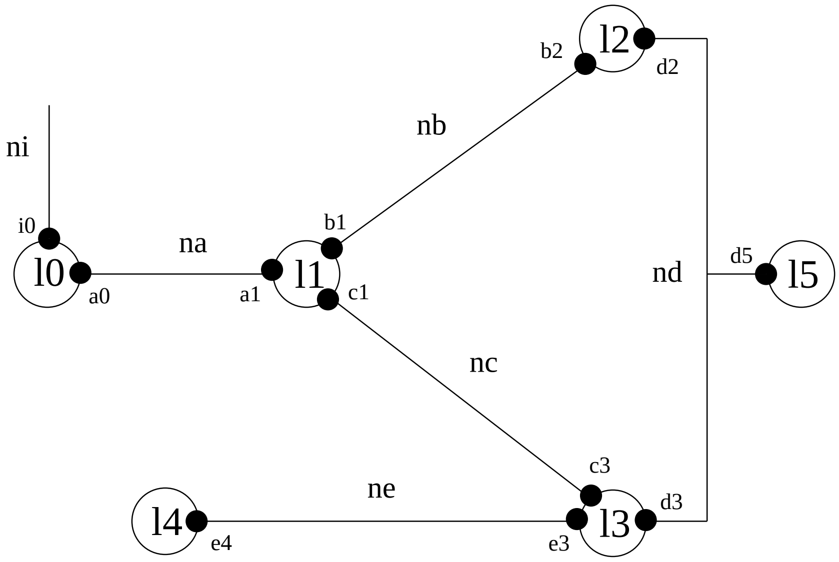

Figure 1: Example Topology

```
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"))),
                       (addr("c1") via (addr("a0") >> addr("a1"))),
                       (addr("b2") via (addr("a0") >> addr("a1"))),
                       (addr("d2") via (addr("a0") >> addr("a1"))),
                       (addr("c3") via (addr("a0") >> addr("a1"))),
                       (addr("e3") via (addr("a0") >> addr("a1"))),
                       (addr("d3") via (addr("a0") >> addr("a1"))),
                       (addr("e4") via (addr("a0") >> addr("a1"))),
                       (addr("d5") 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"))),
                       (addr("d2") via (addr("b1") >> addr("b2"))),
                       (addr("c3") via (addr("c1") >> addr("c3"))),
                       (addr("e3") via (addr("c1") >> addr("c3"))),
                       (addr("d3") via (addr("c1") >> addr("c3"))),
                       (addr("e4") via (addr("c1") >> addr("c3"))),
```

```
                           (addr("d5") via (addr("b1") >> addr("b2")))))
     Data(loc("l1"), empty-dil)
     Node(loc("l2"), (addr("b2"),addr("d2")),
                          ((addr("b2") >> addr("b1")),
                           (addr("d2") >> addr("d3")),
                           (addr("d2") >> addr("d5"))),
                          ((addr("a0") via (addr("b2") >> addr("b1"))),
                           (addr("a1") via (addr("b2") >> addr("b1"))),
                           (addr("b1") via (addr("b2") >> addr("b1"))),
                           (addr("c1") via (addr("b2") >> addr("b1"))),
                           (addr("c3") via (addr("d2") >> addr("d3"))),
                           (addr("e3") via (addr("d2") >> addr("d3"))),
                           (addr("d3") via (addr("d2") >> addr("d3"))),
                           (addr("e4") via (addr("d2") >> addr("d3"))),
                           (addr("d5") via (addr("d2") >> addr("d5")))))
     Data(loc("l2"), empty-dil)
     Node(loc("l3"), (addr("c3"),addr("e3"),addr("d3")),
                          ((addr("c3") >> addr("c1")),
                           (addr("e3") >> addr("e4")),
                           (addr("d3") >> addr("d2")),
                           (addr("d3") >> addr("d5"))),
                          ((addr("a0") via (addr("c3") >> addr("c1"))),
                           (addr("a1") via (addr("c3") >> addr("c1"))),
                           (addr("b1") via (addr("c3") >> addr("c1"))),
                           (addr("c1") via (addr("c3") >> addr("c1"))),
                           (addr("b2") via (addr("d3") >> addr("d2"))),
                           (addr("d2") via (addr("d3") >> addr("d2"))),
                           (addr("e4") via (addr("e3") >> addr("e4"))),
                           (addr("d5") via (addr("d3") >> addr("d5")))))
     ...
```

6.2 Finding a Path (Independent Process Solution)

The first solution floods the network with independent processes which are propagated by each node to neighbors which have not been visited yet (according to the local knowledge the current process). If a destination is reached the `goback` function is used to go back to the node where the program was originally injected into the network. Both the propagation of processes and the return to the original location are performed recursivelly by functions `find` and `goback`, but recursion is not used in the ordinary sense but in the sense of recursive calls at remote locations using the `OnNeighbor` service.

```
ops find-prog-1 : Addr -> Ex .

eq find-prog-1(find-dest) =
   (LetRec ["goback" = Lam [ ("remaining","route")
         : ((TList (TPair TAddr TAddr)),(TList (TPair TAddr TAddr))) ]
     (If (Equal ("remaining"{0}, Nil))
```

```
        Then (Print "route"{0})
        Else (Let ["nexthop" = (Snd (Hd "remaining"{0}))]
               (OnNeighbor
                ((Chunk("goback"{0},((Tl "remaining"{0}),"route"{0}))),
                 "nexthop"{0}, (GetRB empty-exl),
                 (GetDevToHost "nexthop"{0})))))]
   (LetRec ["find" = Lam [ ("dest","route","visited") :
         (TAddr,(TList (TPair TAddr TAddr)),(TList TAddr)) ]
     (Let ["myaddrs" =
       (ThisHost empty-exl)]
     (Let ["neighbors" =
       (GetNeighbors empty-exl)]
     (Let ["addifnew" = Lam [ ("neigh","neighl") :
             ((TPair TAddr TAddr), (TList (TPair TAddr TAddr))) ]
       (If (Not (Member ((Fst "neigh"{0}),"visited"{0})))
        Then (Cons ("neigh"{0},"neighl"{0}))
        Else "neighl"{0})]
     (Let ["newneighbors" =
       (Foldr ("addifnew"{0},"neighbors"{0},Nil))]
     (Let ["childrb" =
       (If (Equal ((Length "newneighbors"{0}),(Int 0)))
        Then (GetRB empty-exl)
        Else (Div ((GetRB empty-exl),(Length "newneighbors"{0}))))]
     (Let ["sendchild" = Lam [ ("neigh","dum")
                               : ((TPair TAddr TAddr),TUnit) ]
       (Let ["route" = (Cons ((Pair ((GetSrcDev empty-exl),
                                     (Snd "neigh"{0}))),
                              "route"{0}))]
         (OnNeighbor
          ((Chunk("find"{0},("dest"{0},"route"{0},
                              (Append ("myaddrs"{0},"visited"{0}))))),
           (Fst "neigh"{0}), "childrb"{0}, (Snd "neigh"{0}))))]
     (If (ThisHostIs "dest"{0})
      Then ("goback"{0} ("route"{0},(Reverse "route"{0})))
      Else (Foldr ("sendchild"{0},"newneighbors"{0},Dummy))))))))]
   ("find"{0} ((Addr find-dest),Nil,Nil)))) .
```

A sample run can be started by simulating the injection of a program into the network. This is done by combing the passive topology `example-topology` with an active process which contains a call of `find`. In this example we use `e4` as a destination address.

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

A possible final configuration computed by Maude is the following. We have only shown processes of the configuration, since the other parts remain unchanged.

Observe that the program has computed three different paths, and all of them are printed at node l0.

```
rewrites: 774851 in 11640ms cpu (11690ms real) (66567 rewrites/second)
result Configuration:
    ...
    Process(loc("l0"), addr("i0"), addr("a0"), 1, 6,
      RedState(?,
        Print (Cons (Pair (Addr addr("i0"),Addr addr("a0")),
               Cons (Pair (Addr addr("a1"),Addr addr("b1")),
               Cons (Pair (Addr addr("b2"),Addr addr("d2")),
               Cons (Pair (Addr addr("d3"),Addr addr("e3")),Nil)))))))
    Process(loc("l0"), addr("i0"), addr("a0"), 1, 12,
      RedState(?,
       Print (Cons (Pair (Addr addr("i0"),Addr addr("a0")),
              Cons (Pair (Addr addr("a1"),Addr addr("c1")),
              Cons (Pair (Addr addr("c3"),Addr addr("e3")),Nil))))))
    Process(loc("l0"), addr("i0"), addr("a0"), 1, 16,
      RedState(?,
       Print (Cons (Pair (Addr addr("i0"),Addr addr("a0")),
              Cons (Pair (Addr addr("a1"),Addr addr("b1")),
              Cons (Pair (Addr addr("b2"),Addr addr("d2")),
              Cons (Pair (Addr addr("d5"),Addr addr("d5")),
              Cons (Pair (Addr addr("d3"),Addr addr("e3")),Nil))))))))
    Process(loc("l0"), addr("i0"), addr("i0"), 1, 0,
      RedState(?, Dummy))
    Process(loc("l1"), addr("i0"), addr("a1"), 1, 1,
      RedState(?, Dummy))
    Process(loc("l1"), addr("i0"), addr("b1"), 1, 0,
      RedState(?,Dummy))
    ...
```

6.3 Finding a Path (Competing Process Solution)

The second solution to the find path problem does not aim to return all paths but at least one. Again the network is flooded with processes which propagate themselves to nodes that are known to be not yet visited. In this solution, howver, the knowledege about the visted nodes is not local to each process, but stored in the data dictionary associated with each node so that that all processes on this node can share the information. In fact, the data stored is more than just the information if the node has been visited; it is actually a pointer to a previously visted node which leads (possibly via a chain of several nodes) back to the origin, i.e. the node where the program was injected.

```
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" =
             (If (Equal ((Length "neighbors"{0}),(Int 0)))
              Then (GetRB empty-exl)
              Else (Div ((GetRB empty-exl),
                         (Length "neighbors"{0}))))]
           (Let ["sendchild" = 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))) )) .
```

Similar to the previous example we can initiate a sample run by:

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

In the computed final configuration we can see that the data dictionary has been used and that a single path has been printed at node l0. The format of paths is somewhat different from the one used in the first solution.

```
rewrites: 328040 in 4740ms cpu (4790ms real) (69206 rewrites/second)
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))))))
Process(loc("l0"), addr("i0"), addr("a0"), 1, 32,
  RedState(?, Dummy))
Process(loc("l0"), addr("i0"), addr("i0"), 1, 0,
  RedState(?, Dummy))
Process(loc("l1"), addr("i0"), addr("a1"), 1, 0,
  RedState(?,Dummy))
Process(loc("l1"), addr("i0"), addr("b1"), 1, 9,
  RedState(?, Dummy))
...
```

The executable specification of PLAN in Maude together with these and further examples can be found on the web [**?**]. In addition, an executable specification of a type checker for PLAN in Maude, which operates on the same representation, has been developped by F. Belloni and is made available on the same site.

7 Conclusion

The formal specification of the PLAN semantics clarifies a number of issues that remain vague or unsatisfactory in the original mathematical specification [20] 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 treament of side-effects (e.g. in the built-in functions `foldr`, `foldl`), the mechanism of exception handing, the concurrent and distributed nature of the active network configuration.

Two essential differences to the mathematical specification are worth to emphazise. First of all, the use of a richer frame language, which is representationally very close to PLAN but leads to a theoretically cleaner and less add-hoc specifications. The use of a quite general frame language furthermore makes our specification more general than necessary for PLAN and reflects the general idea of a programming language for mobile programs based on an imperative λ-calculus. Second, the use of a high-level reduction machine, in contrast to the traditional rather low-level SECD machine, is an improvement which allows us to execute and reson about programs exactlyon the level of the frame language without being concerned with artifical implementation details.

In contrast to the mathematical specification our specification is rigorously formal. Although one may expect that readability would decrease, we have carefully structured the formal specification and we feel that it is actually more concise and easier to understand and comes close to the goal of a specification which can help people with very different intentions and backgrounds ranging from users and language implementors to formal method engineers.

Last but not least, 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 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 model for formal theorem proving.

References

[1] The DARPA Active Network Program. DARPA Information Technology Office, `http://www.darpa.mil/ito/research/anets/index.html`.

[2] The ONR Coordinated Research in Adaptive Network Centric Computing (CRANCC). `http://www.cis.upenn.edu/sdrl/crancc/`.

[3] IEEE, January 2000.

[4] K. J. Berkling. A symmetric complement to the lambda-calulus. Interner Bericht ISF-76-7, GMD, St. Augustin, Germany, 1976.

[5] K. J. Berkling and E. Fehr. A consistent extension of the lambda-calculus as a base for functional programming languages. *Information and Control*, 55:89–101, 1982.

[6] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[7] 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`.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on maude. `http://maude.csl.sri.com`, March 2000.

[9] N. G. de Bruijn. Lambda calculus with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Proceedings Kninkl. Nederl. Akademie van Wetenschappen*, volume 75(5), pages 381–392, 1972.

[10] 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.

[11] J G. Denker, 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.

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

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

[14] 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`.

[15] 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`.

[16] 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`.

[17] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society INFOCOM Conference*, pages 1124–1133. IEEE, 1999. `http://www.cis.upenn.edu/~switchware/papers/planet.ps`.

[18] 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.

[19] 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.

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

[21] 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`.

[22] 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`.

[23] I. A. Mason and C. L. Talcott. Feferman-Landin logic. In *Reflections – A symposium hounoring Solomon Feferman on his 70th birthday*, Lecture Notes in Logic. Association of Symbolic Logic, 2001.

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

[25] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18 – 61. Springer-Verlag, 1998.

[26] 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`.

[27] 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`. Extended version at `http://www.csl.sri.com/~stehr`.

[28] M.-O. Stehr. Concurrent object-oriented specification and analysis in Maude. In *FMOODS'2000, IFIP TC6/WG6.1 Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems, Stanford University, Stanford, CA, USA, September 6–8, 2000, Poster Abstracts*, 2000. `http://www.ics.uci.edu/~fmds2000/`.

[29] C. L. Talcott. Specifying the PLAN language in maude (joint work with j. meseguer, m.-o. stehr, c. gunter and p. kakkar), 2000. Talk given at University of Tokyo. Slides available at `http://www-formal.stanford.edu/clt/Talks/00sep-utokyo-talk.ps.gz`.

[30] C. L. Talcott. PLAN in Maude: A specification with multiple purposes (joint work with j. meseguer, m.-o. stehr, c. gunter and p. kakkar). In *Abstracts of the Workshop on Proofs for Mobility (PFM'2001), Genova, April 7, 2001*, 2001. http://www.pfm2001.ens.fr.

[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] B.-Y. Wang, J. Meseguer, and C. A. Gunter. Specification and formal verification of a PLAN algorithm in Maude. In T. Lai, editor, *Proceedings of the 2000 ICDCS workshop on Internet 2000, Distributed Real-Time Systems, Group Computation and Communications, Wireless Networks and Mobile Computing, Distributed System Validation and Verification, Knowledge Discovery and Data Mining in the World Wide Web*, pages E:49–E:56. IEEE, April 2000.