Maté Manual

Philip Levis
pal@cs.berkeley.edu

Version 2.19a
November 30, 2004

Contents

1 Introduction

Maté is a framework for building accessible programming interfaces to TinyOS sensor networks. The core of Maté is a bytecode interpreter template. A user can customize the interpeter's instruction set and execution events to match the abstractions needed by a particular deployment, and programs a network with high-level scripts. Given the right set of abstractions, a user script can express complex behavior concisely and simply. Conciseness allows programs to compile to a small number of instructions, so code propagation can be rapid and inexpensive. Simplicity makes bugs less likely.

Once introduced, Maté programs self-propagate through a network using an epidemic broadcast protocol. Reprogramming only requires introducing a single copy of a new program: this copy will then install itself across the entire network.

This document describes the Maté architecture and how to use it. It outlines the major TinyOS components that comprise a Maté template, the interfaces they use to interact, describes the algorithms Maté uses for services such as code propagation and synchronization, and covers how the VMBuilder tool builds a virtual machine from user specifications. It assumes you have already read the Maté tutorials, and provides details beyond them, such as how to implement a new language.

2 Maté Interfaces

The nesC components that comprise Maté have a wide range of interfaces. This section contains a brief description of each interface. Detailed information on the individual commands and events can be found in the standard nesdoc documentation.

2.1 MateAnalysis

MateAnalysis is for invoking resource utilization analysis. When new a handler arrives, the Maté's viral propagation subsystem calls MateAnalysis to compute what shared resources the code uses. Maté uses this information to determine which handlers can safely run concurrently, and which cannot.

2.2 MateBuffer

MateBuffer is for accessing buffer data structres as an abstract data type. MateBuffer has commands for inserting, removing, sorting, and typechecking.

2.3 MateBytecodeLock

MateBytecodeLock is how Maté's code analysis determines what shared resource an instruction uses, so it can determine what handlers can safely run concurrently. If an instruction encapsulates a shared resource, then it must implement this interface.

2.4 MateBytecode

MateBytecode is the bytecode execution interface. When the interpreter executes a bytecode, it executes an instance of this interface. The interface also has a command that returns the byte width of the instruction, so the scheduler knows how much to increment a context's program counter by.

2.5 MateContextLocks

MateContextLocks has commands for acquiring and releasing locks on a context basis, and determinig whether a context can acquire all of the locks it needs. These commands are rarely called by external components; they are used by implementations of MateContextSynch to halt and resume contexts.

2.6 MateContextStatus

MateContextStatus has a single event, which fires when a context halts. Among other things, this allows a context that has queued execution requests to know when it can handle the next one.

2.7 MateContextSynch

MateContextSynch is how components interact with the Maté concurrency manager. Components can submit contexts to the Maté concurrency manager for execution. The concurrency manager decides which contexts can safely run concurrently and forwards them to the scheduler. MateContextSynch has commands for resuming, halting, and yielding contexts.

2.8 MateEngineControl

MateEngineControl has events for signalling the VM to reboot, halt, or resume. Telling the VM to reboot will make it signal its own reboot event to interested components through the MateEngineStatus interface.

2.9 MateEngineStatus

MateEngineStatus is how the VM engine notifies interested components when it reboots. For example, when the VM reboots, context components reset their contexts and split-phase instructions clear their queues.

2.10 MateError

MateError is for indicating an error has occured, that should halt execution. When invoked, this causes the VM to enter an error state, blinking the LEDs and broadcasting the cause of the error.

2.11 MateHandlerStore

MateHandlerStore is how components interact with the underlying code store and propagation subsystem. It presents code handlers as an abstract data type, with accessor commands and an event for notifying when code has changed.

2.12 MateLocks

MateLocks presents shared resources locks as an abstract data type. The Maté concurrency manager uses MateLocks to manage utilization of shared resources.

2.13 MateQueue

MateQueue is for manipulating context queues as an abstract data type. It supports enqueueing, dequeueing, removal, and initialization. Several Maté components use context queues, including the scheduler, concurrency manager, and blocking operations.

2.14 MateScheduler

MateScheduler is the interface the core VM interpreter provides for submitting contexts to the run queue. The Maté concurrency manager uses this interface to submit contexts it has determined to be safe to run.

2.15 MateStacks

MateStacks presents the operand stack of a Maté execution context as an abstract data type. It has commands for initializing a stack, pushing various types of operands, and popping operands.

2.16 MateTypes

MateTypes provides commands for operand typechecking. Generally, a command has two forms, query and check. Queries merely return whether an operand passes a type requirement; checks return whether the operand passes, and automatically trigger an error condition if the check fails.

2.17 MateVirus

MateVirus is the interface to Maté's viral code propagation subsystem. Generally, a component that provides MateHandlerStore sits on top of a component that provides MateVirus. MateHandlerStore signals arrival in terms of units of execution, while MateVirus signals arrivals in terms of code propagation (a single propagation unit, for example, may contain two handlers).

2.18 MateType

Language-independent functions cannot make assumptions about a language's data model. When data is internal to a mote, this is not a problem: a VM controls access to data structures, so their internal representation is separated from a program. When VMs communicate (over the radio, for example), however, they must agree on a data format, which can be different than the in-memory representation a VM uses. For example, a VM may represent a list as a linked list in memory, but needs to compact it to a vector to transmit it. The MateType interface is for packing and unpacking network data type representations, so functions can handle data types without knowing their internal structure.

3 Maté Template

A Maté VM's components fall into two classes: the components every VM includes (the basic template), and the components that define the particular Maté instance. The basic VM template includes scheduling, concurrency managment, and code storage/propagation. Adding an instruction set and execution contexts to the template makes a application-specific virtual machine. This section describes the three major template components, `MateEngine`, `MContextSynchProxy`, and `MHandlerStoreProxy`.

Many Maté subsystems have "Proxy" components. These proxy components separate the interface of the subsystem from its implementation. If every Maté component wires to the proxy, instead of the component itself, then a user can change what implementation the VM uses by only changing the proxy component. For example, to change the `MateLocks` implementation from `MLocks` to `MLockSafe` (the latter performs many checks the former does not), a user only has to change the `MLocksProxy` component to refer to `MLocksSafe`. In contrast, if a proxy were not used, then every file which wires to `MLocks` would have to be changed to `MLocksProxy`.

3.1 Scheduling: `MateEngine`

`MateEngine` is a configuration that wires `MateEngineM`, the core Maté scheduler, to all of its needed subsystems. It has the following signature:

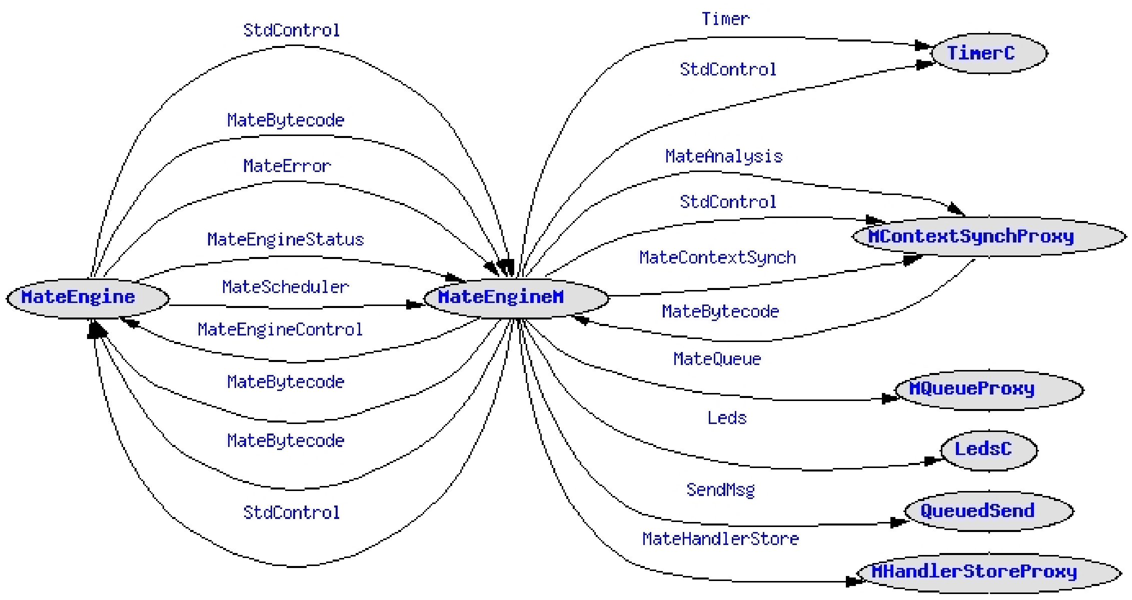

Figure 1: `MateEngine` wiring diagram.

```
configuration MateEngine {
  provides {
    interface StdControl;
    interface MateError as Error;
    interface MateEngineStatus as EngineStatus;
    interface MateScheduler as Scheduler;
    interface MateBytecode as Functions[uint8_t functionID];
  }
  uses {
    interface MateEngineControl as EngineControl;
    interface MateBytecode as Bytecode[uint8_t bytecode];
    interface MateBytecode as FunctionImpls[uint8_t fnID];
    interface StdControl as SubControl;
  }
}
```

Figure 1 shows how `MateEngine` wires `MateEngineM`. `TimerC`, `LedsC`, and `QueuedSend` are all for when an error condition occurs (triggered by `MateError`): `MateEngine` starts a periodic timer, blinking the LEDs and broadcasting the source of the error. `MQueueProxy` is for manipulating the run queue and `MHandlerStoreProxy` is for fetching opcodes from handlers.

The set of components that are wired to `MateEngine`'s parameterized `Bytecode` implement a VM's instruction set. The main execution loop fetches the next bytecode from a handler (through the HandlerStore), then dispatches on this interface based on the opcode value. For example, if the bytecode `halt` has value `0x2a`, then the `OPhalt` component is wired to `MateEngine.Bytecode[0x2a]`. Generally, functions included in a VM (such as `send`) exist as bytecodes.

The `Functions` and `FunctionImpls` are a bit more complex. First order language (such as motlle) need to be able to refer to functions by values, which can be stored and passed. The language then needs a way to take this value and execute the function it refers to. If a VM supports a first-class language, then all of the functions must be wired to `FunctionImpls`: the parameters for this interface are distinct from those for the `Bytecode` interface. `Functions` is a simple pass-through to `FunctionImpls`. An instruction component can wire to `FunctionImpls` to dispatch to a function based on a value.

Any stand-alone component that has to provide `StdControl` should wire it to `MateEngine`'s `SubControl`. This will allow the VM to support power management in the future.

`MateEngineM` follows a round-robin FIFO policy. `MateEngineM` has two configuration constants for timeslicing, `MATE_CPU_QUANTUM` and `MATE_CPU_SLICE`. `MateEngineM` executes instructions in a task. `QUANTUM` is the maximum number of instructions it interprets in each task execution; `SLICE` is the number of quanta it gives to a context before switching to a new one. By default, `MATE_CPU_SLICE` is 5 and `MATE_CPU_QUANTUM` is 4. Unless a context halts or blocks, `MateEngine` timeslices them at the granularity of 20 instructions.

3.2 Data Model: `MStacks` and `MTypeManager`

Maté VMs follow a stack architecture. Each thread (execution context) has an operand stack. For example, to perform arithmetic addition, a program pushes two numbers onto the stack, then executes the add instruction. The add instruction pops the two elements off the stack, adds them and pushes the result onto the stack. The `MStacks` component presents the operand stack as an abstract data type.

Operands (and more generally, variables) have an associated type. Some types, such as integers, are simple. Variables can be more complex types, such as vectors, lists, or strings. When Maté motes communicate, they need to take the in-memory representation of a type and transform it into something that can be sent over a network. For example, a linked list needs to be compacted into a linear sequence (i.e., array); the receiver can then unpack the serialized form into its desired in-memory representation.

`MTypeManager` provides interfaces to the set of network-compatible types. Specifically, it provides a parameterized interface (the parameter is the type ID) of type `MateType` (which is distinct from `MateTypes`, which is type-checking). To transform a variable between in-memory and network represntations, components can invoke `MTypeManager`. `MTypeManager`'s interface is a pass-through to the implementing components: a component that supports a given type (such as `MBuffer`, which supports TinyScript's data buffers) wires to `MTypeManager` so calls will be forwarded properly. If a VM tries to transform a type for which there is no support, then `MTypeManager` indicates that the type is not supported: this will generally trigger an error condition in the VM.

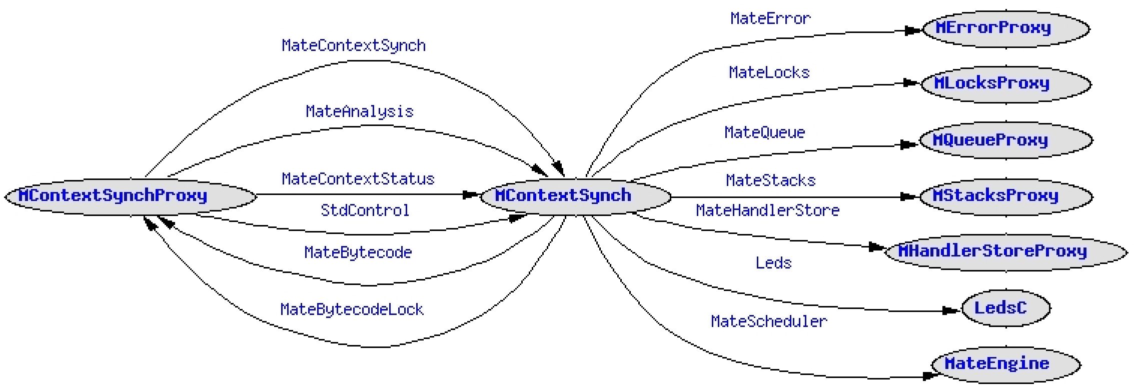

Figure 2: `MContextSynchProxy` wiring diagram.

3.3 Concurrency Management: `MContextSynchProxy`

`MContextSynchProxy` is encapsulates `MContextSynch` and wires it to needed services. `MContextSynch` is responsible for analyzing code to determine when handlers can safely run concurrently. ome instructions represent shared resources. For example, `bpush3` and `getvar4` access shared variables. For race free program, the VM execution engine must be aware of this and control scheduling appropriately.

When a component wants a context to run, it submits the context to `MContextSynch`; based on its analyses, `MContextSynch` either forwards the context on to `MateEngine` for execution, or puts it on a wait queue. If a context holding shared resources halts, `MContextSynch` has it release the resources and checks if that allows any waiting contexts to run.

`MContextSynch` keeps track of shared resources through the MateBytecodeLock interface. If an instruction manages a shared resource, then it must provide this interface. Additionally, in its ODF, it must have the optional element "locks" set to true.

For example, let's look at OPbpush3. This instruction pushes the eight shared buffers, buffer0-7, onto the operand stack. It is not a library function; instead, it is an element of the instruction set that TinyScript compiles to. In `tscript.ldf` (Section 5 describes language files in greater depth), it lists

```
<OPCODE opcode="bpush3" locks=true>
```

Then, in OPbpush3M:

```
module OPbpush3M {
  provides interface MateBytecode;
  provides interface MateBytecodeLock;
}
```

`MateBytecodeLock` has a single command:

```
interface MateBytecodeLock {
  command int16_t lockNum(uint8_t instr);
}
```

This takes a instruction opcode and returns a unique lock number. The idea is that certain opcodes have a lock associated with them. `bpush3`, for example, has three bits of embedded operand, for the eight buffers. If `bpush3 0` is passed to `OPbpush3M.nc`, then it returns the lock number for buffer zero, while `bpush3 1` will return the lock number for buffer one.

The full `OPbpush1M.nc` logic:

```
module OPbpush3M {
  ...
  provides interface MateBytecodeLock;
  ...
}


implementation {
  typedef enum {
    BOMB_BUF_LOCK_3_0 = unique("MateLock"),
    BOMB_BUF_LOCK_3_1 = unique("MateLock"),
    ...
    BOMB_BUF_LOCK_3_7 = unique("MateLock"),
  } BufLockNames;
  ...
  command int16_t MateBytecodeLock.lockNum(uint8_t instr) {
    uint8_t which = instr - OPbpush3;
    switch (which) {
    case 0:
      return BOMB_BUF_LOCK_3_0;
    case 1:
      return BOMB_BUF_LOCK_3_1;
    ...
    case 7:
      return BOMB_BUF_LOCK_3_7;
    default:
      return 255;
    }
  }
  ...
}
```

It declares eight unique lock numbers with the nesC unique function. When lockNum() is called, it returns the lock number associated with the corresponding buffer. Every context has

```
  uint8_t heldSet[(BOMB_LOCK_COUNT + 7) / 8];
  uint8_t releaseSet[(BOMB_LOCK_COUNT + 7) / 8];
  uint8_t acquireSet[(BOMB_LOCK_COUNT + 7) / 8];
```

where BOMB_LOCK_COUNT is defined to be uniqueCount(”MateLock”);

3.3.1 Synchronization Algorithms

Maté's concurrency manager is responsible for ensuring that handlers execute atomically. Although it may allow them to run concurrently, atomicity requires that doing so should be indistinguishable from their running serially. The concurrency manager maintains a bitmask of the shared resoures each handler uses (the *uses set*), and each context has three bitmasks: the set of resources it holds (the *held set*), the set of resources it can release (the *release set*), and the set of resources it needs to acquire (the *acquire set*). Initializing a context through `MateContextSynch.initializeContext` sets a context's acquire set to its handler's uses set. Every time MContextSynch acquires a lock for a context, it adds that resource to the context's held set.

When new code for a handler arrives, the concurrency manager runs a context and flow insensitive program analysis to determine the set of shared resources that handler uses. It iterates over each instruction in the handler, using the `MateBytecodeLock` interface to determine whether an instruction requires a shared resource, updating the uses set of the handler.

The concurrency manager maintains a lock for each shared resource; only one context may access a shared resource. When a component submits a context to run through the `MateContextSynch.resumeContext` command, MContextSynch checks the acquire set of the context. If every resource in the acquire set is available, MContextSynch atomically acquires all of the locks for the context and submits it to the scheduler to run. If one or more of the resources in the context's acquire set are already held, then MContextSynch puts the context on a wait queue and sets its state to `MATE_STATE_WAITING`.

While they execute, contexts may add resources to their release set. This does not release those resources. When a context yields, through the `MateContextSynch.yield` command, MContextSynch has it unlock each of the resources in its release set, then clears the set. MContextSynch then tests each context on the wait queue to see if it is now runnable (the release of locks means all locks in its acquire set are free). If a context is runnable, MContextSynch resumes it.

When a context adds a resource to its release set, it may also add it to its acquire set. This means that the context will release those resources at the next yield point, but will not continue execution until it can reacquire them: it can temporarily release the resources.

Currently, TinyScript does not manipulate context release sets, although it has instructions for doing so. This is an area for future work.

Because, once it starts running, the acquire set of a context is always a subset of its release set, the set of resources a context holds while running is montonically decreasing. A context can never hold a resource that it did not hold when it started running. This ensures that contexts can run deadlock-free, while the program analysis (barring incorrect lock releases at runtime) ensures race-free execution.

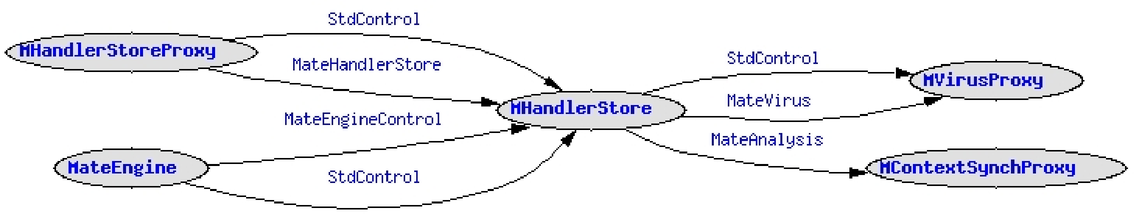

Figure 3: `MHanderStoreProxy` wiring diagram.

3.4 Code Propagation: `MHandlerStoreProxy`

The MHandlerStoreProxy component encapsulates Maté's code storage and propagation. Figure 3 contains its wiring diagram, and the component has the following signature:

```
configuration MHandlerStoreProxy {
  provides {
    interface StdControl;
    interface MateHandlerStore as HandlerStore[uint8_t id];
  }
}
```

Every component that has a handler should wire to HandlerStore, with a unique ID (the handler ID). VMBuilder automatically generates handler IDs for contexts, with `unique("MateHandlerID")`; if other components need to register handlers, they should use the same key for `unique`.

`MateHandlerStore` has the following signature:

```
interface MateHandlerStore {
  command result_t initializeHandler();
  command MateHandlerOptions getOptions();
  command MateHandlerLength getCodeLength();
  command MateOpcode getOpcode(uint16_t which);
  event void handlerChanged();
}
```

It provides accessor functions for handlers, and notifies its user when the code for the handler has changed. Currently, MHandlerStore allocates a static amount of memory for each handler's code (the default is 128 bytes), but by controlling access through an interface, this can easily be changed.

MHandlerStore provides access to code at handler granularity, but Maté propagates code in terms of capsules, which contain one or more handlers. The default implementation has a one-to-one handler-capsule mapping, but languages or compilers may require other implementations. For example, motlle, due to its semantics, requires that all handlers propagate together in a single capsule.

MHandlerStore is responsible for triggering program analysis via MContextSynchProxy when a capsule arrives. MHandlerStore sits on top of MVirus, which handles capsule propa-

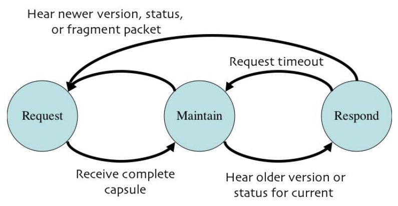
Hear newer version, status, or fragment packet
Request timeout
Request
Maintain
Respond
Receive complete capsule
Hear older version or status for current

Figure 4: State diagram for Maté capsule propagation.

gation. The idea is that a particular MHandlerStore implementation determines the format of the data regions of capsules and can parse them into handlers.

MVirus uses an epidemic-like approach to propagate capsules: a node that has newer code will broadcast it to local neighbors. The Trickle algorithm is used to broadcast three types of data: version packets, which contain the 32-bit version numbers of all installed capsules, capsule status packets which describes fragments of a capsule that a mote needs (essentially, a bitmask), and capsule fragments that are short segments of a capsule. Figure 4 contains the state diagram used by motes for code propagation. A mote can be in one of three states: maintain (exchanging version packets), request (sending capsule status packets), or respond (sending fragments). Nodes start in the maintain state. They enter the request state if they hear something that indicates someone has a newer capsule, whether it be a version, capsule status, or fragment packet. A requesting node returns to the maintain state once it receives the entire capsule. A node enters the respond state if it is in the maintain state and hears that someone has an older capsule (through a version packet), or needs part of its current capsule (through a capsule status packet). These state transitions mean that nodes prefer requesting over responding; a node will defer forwarding capsules until it thinks it is completely up to date.

Trickle's suppression operates on each type of packet (version, capsule status, and capsule fragment) individually. That is, a capsule fragment transmission will suppress all other fragment transmissions, but will not suppress version packets. This allows meta-data exchanges during propagation: sending a fragment will not cause someone to suppress a message saying what fragments it needs. Trickling fragments means that code propagates in a slow and controlled fashion, instead of as quickly as possible. This is unlikely to significantly disrupt any existing traffic, and prevents network overload.

3.4.1 Trickle: The Code Propagation Algorithm

The Trickle algoithm uses broadcast-based suppressions to quickly propagate new data but minimize transmissions when nodes share data. The algorithm operates on an time interval t, which has an upper length of t_h and a lower length of t_l.When an interval completes, Trickle doubles the size of an interval, up to t_h. When it learns of new code (e.g., by overhearing a capsule fragment, version vector, or capsule status packet with a higher version number), it shrinks the interval to t_l.

Essentially, when there's nothing new to say, Maté VMs gossip infrequently: τ is set to τ_h. However, as soon as a mote hears something new, it gossips more frequently, so those who haven't heard it yet find out. The chatter then dies down, as τ grows from τ_l to τ_h.

Trickle maintains a redundancy constant k and a counter c. Whenever it hears a packet that would suppress its own transmission (e.g., a capsule fragment if in the respond state), it increments c. At the beginning of an interval, the algorithm resets c to zero, and picks a random time in the range of $[\frac{t}{2}, t]$. When it reaches that time, it transmits data if and only if $c < k$.

4 Maté Operations

MateEngine executes programs in terms of operations. Every operation has one or more one-byte opcodes, which map to a component that implements the `MateBytecode` interface. An operation can have embedded operands, which cause it to exist as more than one opcode. For example, the `bpush3` operation (which the TinyScript language uses) has three bits of embedded operand, corresponding to the eight data buffers TinyScript makes available.

The Maté tutorials describe how to write new functions, which are a particular kind of operation: they never have embedded operands, and are generally expected to provide language-independent functionality. In addition to functions, operations can also be primitives, which are the operations that compose a language. This is why function components have the prefix "OP," such as `OPid`.

4.1 Operation Component Naming Convention

Operation components have the following naming convention:

```
OP<width><name><operand>.nc
```

Width and operand are both numbers. Width specifies how many bytes wide the instruction is. If no width is specified, the default is one. Operand specifies how many bytes of embedded operand there are. If no operand is specified, the default is zero. Note that, after considering width and operand, the instruction must have an opcode in its first byte. That is, the instruction cannot be two bytes wide and have no embedded operand; as the Maté scheduler dispatches on the first byte of the opcode, it would not be able to distinguish this instruction from other ones.

Here are a few examples:

Component	Width	Name	Embedded	Description
OPrand	1	rand	0	Generates a random number
OPpushc6	1	pushc	6	Push a constant onto the stack
OP2jumps10	2	jumps	10	Jump to a 10-bit address

Functions are always one byte wide and never have embedded operands. In the above example, neither `pushc` nor `jumps` are available as library functions; they are actually primitives that compose part of what TinyScript compiles to.

4.2 Primitives

Primitives are operation components that a language compiles to. They cannot be included in a VM in the same way functions can: they have no ODF which VMBuilder can use to include them. The only way to include them is to use a language that compiles to them. Operations such as branches, memory access, and arithmetic are examples of Maté primitives.

Primitives provide language storage abstractions, such as variables. For example, TinyScript has the notion of sixteen variables that are shared between all handlers, and eight variables which are private to each handler. The operations `getvar4` and `setvar4` implement the former, while the operations `getlocal3` and `setlocal3` implement the latter.

The implementations of the two abstractions are very similar. A single component implements `getvar4` and `setvar4`, providing two `MateBytecode` interfaces. Sixteen opcodes map to each instance of `MateBytecode`, for the four bits of embedded operand. The component allocates sixteen shared variables in its frame. To determine which variable a program is accessing, the implementation subtracts the executed opcode from the base opcode, Additionally, the implementation checks that the program holds the lock to the shared variable:

```
  MateStackVariable heap[16];

  command result_t Get.execute(uint8_t instr,
                               MateContext* context) {
    uint8_t arg = instr - OPgetvar4;
    uint8_t lock = varToLock(arg);
    if ((lock == 255) || !call Locks.isHeldBy(lock, context)) {
      call Error.error(context, MATE_ERROR_INVALID_ACCESS);
      return FAIL;
    }
    dbg(DBG_USR1, "VM (%i): Executing getvar (%i).\n", (int)context->which, (int
)arg);
    call Stacks.pushOperand(context, &heap[arg]);
    return SUCCESS;
  }
```

`getlocal3` and `setlocal3` are similar, except there are only eight variables, and there's no need to check locks. However, the component that implements the two primitives has to allocate state for each context:

```
  MateStackVariable vars[MATE_CONTEXT_NUM][NUM_VARS];

  command result_t Get.execute(uint8_t instr,
                               MateContext* context) {
    uint8_t arg = instr - OPgetlocal3;
    dbg(DBG_USR1, "VM (%i): OPgetlocal3 (%i).\n", (int)context->which, (int)arg)
;
    call Stacks.pushOperand(context, &vars[context->which][arg]);
    return SUCCESS;
  }
```

Some primitives are more than one byte wide. For example, `OP2jumps10`, the basic branch instruction is two bytes wide. The `byteLength()` command of the `MateBytecode` interface must return the byte width of an instruction, so the scheduler knows how much to increment the program counter by. `It increments the program counter before executing the instruction.` The component implementing the primitive is responsible for getting the extra bytes. For example, this is part of `OP2jumps10M`:

```
  command uint8_t MateBytecode.byteLength() {return 2;}

  command result_t MateBytecode.execute(uint8_t instr,
                                        MateContext* context) {
    uint16_t addr = (instr - OP2jumps10) << 8;
    MateStackVariable* cond = call Stacks.popOperand(context);

    addr |= call Store.getOpcode[context->currentHandler](context->pc-1);
```

It generates the 10-bit jump address by taking the bottom two bits of the opcode and incorporating the next byte. Since the VM has already incremented the program counter, the next byte is at `context->pc-1`.

The Maté tutorials briefly mention a requirement when using the Maté operand stack. When a component pops operands off the stack with `popOperand`, the call returns a pointer to a stack variable. This is a pointer into the stack data structure. If a component then pushes something onto the stack, that push can modify the region of the stack the pointer refers to. There are situations when it is safe to access popped operands after pushes, however.

Elements on the operand stack have a fixed size. For example, a buffer (which stores a pointer) is the same size as a value. An operand stack is an array of operands and a stack pointer, which indicates the next free element. Pushing something onto the stack fills the next free element, and increments the stack pointer. Popping something off the stack decrements the stack pointer, and returns a pointer to that element. The following three snippets of pseudocode are examples of the resulting behavior:

```
  op1 = pop();
  op2 = pop();
  push(4);   // op2 is now invalidated
  push(op2); // BUG, unless you want to copy the top of the operand stack
  op1 = pop();
  op2 = pop();
  push(op2);  // SAFE; returns op2 to operand stack
  push(op1);  // same as above

  op1 = pop();
  op2 = pop();
  push(op1); // invalidates op2, which is now the same as op1
  push(4);   // invalidates op1
```

5 Languages

A language is defined by the set of primitives it compiles to. A LANGUAGE element in a VM specification file causes VMBuilder to search for a language description file (`.ldf`). An LDF must have a LANGUAGE element with the NAME and DESC tags. Additionally, it should have a series of PRIMITIVE elements. PRIMITIVE elements have one required tag, OPCODE. They also have the optional field LOCKS, which specifies if the primitive encapsulates a shared resources which the concurrency manager must arbitrate access to. For example, this is a snippet of `tscript.ldf`:

```
<LANGUAGE name="TinyScript" desc="A simple, BASIC-like language.">
<PRIMITIVE opcode="halt">
<PRIMITIVE opcode="2pushc10">
<PRIMITIVE opcode="2jumps10">
<PRIMITIVE opcode="getlocal3">
<PRIMITIVE opcode="setlocal3">
<PRIMITIVE opcode="bpush3" locks=true>
<PRIMITIVE opcode="getvar4" locks=true>
<PRIMITIVE opcode="setvar4" locks=true>
<PRIMITIVE opcode="or">
<PRIMITIVE opcode="and">
<PRIMITIVE opcode="not">
<PRIMITIVE opcode="eq">
<PRIMITIVE opcode="gte">
<PRIMITIVE opcode="gt">
```

VMBuilder interprets PRIMITIVE elements in a language file; it does not load any additional files in response to them. It uses the OPCODE tag to refer to the primitive's component when wiring the instruction set. If the LOCKS tag exists (the value is ignored), the VMBuilder also has MateTopLevel wire the primitive component to the concurrency manager.

The LANGUAGE element has a single optional tag, FIRSTORDERFUNCTIONS. If the LANGAUGE element has this tag (whose value is ignored), the VMBuilder includes support for first order functions. It does so by wiring functions to the `FunctionImpls` interface of `MateEngine` and generating a set of function identifiers of the form fn_name, where *name* is the name of the function. These indentifiers are in an enum in `MateConstants.h`, and can therefore can be accessed through the VM Java constants file.

6 VM Options

Tutorial 4 presents how to specify the language, events, and functions that a Maté VM supports. More advanced users can also modify various VM options. These options can be set with one or more `OPTION` elements. The supported options are:

Name	Type	Description
OPDEPTH	integer	Sets the maximum depth of a context operand stack. The default value is 8. Changing this value will change the maximum length of TinyScript statements. Larger values increase RAM utilization.
BUF_LEN	integer	Sets the maximum size of a data buffer. The default value is 10. Larger values will allow you to manage larger buffers. However, this will increase the RAM allocated for each buffer. If made much larger, it increase packet size, **greatly** increasing RAM utilization. Decreasing this value will not reduce the RAM utilization of message buffers.
CAPSULE_SIZE	integer	Sets the maximum size of a code capsule. The default value is 128. Changing this value will affect how large a program you can write. Larger values increase RAM utilization.
DELUGE	string	Support Deluge binary reprogramming.
		The string specified is ignored. Including this option will incorporate Deluge binary reprogramming. Full information on Deluge can be found in the Deluge manual, which is part of the a standard TinyOS release. Deluge uses roughly 10kB of code memory and 250B of RAM.

For example, to change some options, you could add either a single element

```
<OPTION OPDEPTH=6 CAPSULE_SIZE=64>
```

or multiple elements

```
<OPTION OPDEPTH=6>
<OPTION CAPSULE_SIZE=64>
```

7 Java Toolchain

When VMBuilder generates a Makefile for a Maté VM, it includes rules for building a few Java classes that the Maté toolchain uses. The foremost of these is a constants class, which it generates with the `ncg` tool. This class contains all of the mappings between handler names and IDs, context names and IDs, instructions and their bytecodes, error codes, and data types. It also generates a set of message classes, for interacting with a VM.

7.1 Constants Class

Every VM has a constants class. The name of the class is VM-specific, to prevent users from accidentally loading the wrong constants file. For example, the Bombilla constants class name is BombillaConstants. The class contains all of the constants contained in MateConstants.h as public variables. The Maté toolchain has a class, named `net.tinyos.script.ConstantMapper`, for easily accessing these variables through the Java reflection API.

The constructor to ConstantMapper takes two arguments, a class name and a prefix. The prefix acts as a filter on the constants it considers. For example, if you instantiate a ConstantMapper like so:

```
ConstantMapper map = new ConstantMapper("BombillaConstants", "OP");
```

then all of its accessor functions will operate on the public fields of a class named `BombillaConstants` whose name begins with `OP`, in this case the instructions of the Bombilla VM.

ConstantMapper provides three basic methods for fetching constants: `codeToName()` `nameToCode()`, and `names()`. For example, the TinyScript compiler takes a TinyScript program and produces assembly code for it. The ScriptAssembler then takes each instruction and determines its bytecode with the `nameToCode()` method. In contrast, when the Scripter displays a VM error, it reads the binary values of the error condition, context, and handler, and produces human-readable names for them with the `codeToName()` method. Finally, the ScripterGUI determines the set of valid handlers one can write scripts for by using `names()`, then translates those names to IDs with `nameToCode()`.

Obtaining the name of a VM's constants file requires reading in the VM description file that VMBuilder produces. Every VM application directory has a file named `vm.vmdf`, which describes the VM. The Java class `net.tinyos.script.Configuration` automatically loads and parses VM descroption files, extracting the important fields. It takes the path to the file as an argument in its constructor. This is why you must run Scripter from the VM application directory: it instantiates a Configuration with the name `vm.vmdf`. This will also ensure that the Java loader will find the right constants file. Configuration has accessor functions to get at the important instances of ConstantMapper, such as capsule names, opcodes, and error codes. You can also get the constant class name, if you need to build other ConstantMappers, with the `constantClassName()` method.

7.2 Message Classes

By default, the Makefile VMBuilder generates builds several Java message classes. These classes are all built in the `vm_specific` subdirectory. They allow Scripters to generate the proper packets for transmitting code to a mote, as well as read mote data output.

The first class, code transmission, has three kinds of messages: `CapsuleMsg`, `CapsuleStatusMsg`, and `CapsuleChunkMsg`. CapsuleMsg is for generating a full Maté capsule that matches the on-mote memory layout. CapsuleStatusMsg is so tools can monitor download status as it occurs. CapsuleChunkMsg is so the Scripters can send the chunks that make up a capsule along the serial port to a mote.

The second class, data transmission, has three kinds of messages: `BufferBCastMsg`, `BufferUARTMsg`, and `MultiHopMsg`. The first two are the format the `bcast` and `uart` functions send: a Maté data buffer in the payload of a packet. MultiHopMsg is for packets sent through a multi-hop routing layer: it includes multihop header fields, as well as the Maté buffer. The tool `net.tinyos.script.VMBufferReader` listens for all three of these kinds of packets and outputs information on them. It is a good place to start if you want to read data from Maté into a Java application.

8 Conclusion

Maté is under active development. Bugs, questions, contexts, and functions can be sent to Phil Levis (`pal@cs.berkeley.edu`).