Maté: A Tiny Virtual Machine for TinyOS

Philip Levis and Neil Patel
pal@cs.berkeley.edu

Version 1.1
August 21, 2003

Contents

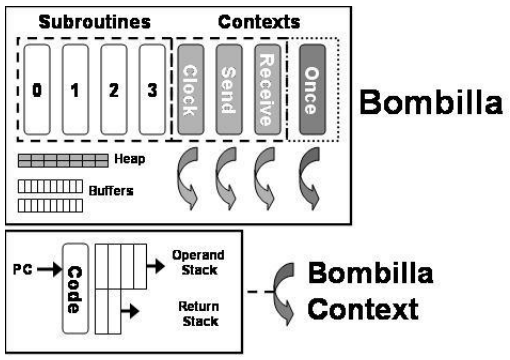

Figure 1: Bombilla Architecture and Execution Model: Capsules, Contexts, and Stacks

1 Introduction

Maté is a tiny bytecode interpreter that runs on top of TinyOS. Because it presents a high-level communication-centric interface, Maté's programs are very short; combined with a safe execution environment, this makes mote reprogramming rapid and error-free. Programs can self-propagate through a network; this makes reprogramming mostly autonomous, as once a self-propagating capsule is introduced, it will eventually install itself over the entire network.

Maté has multiple execution contexts; each runs in response to an event, and they can interleave at instruction granularity. Maté prevents race conditions by implicitly locking all shared state that is used; as every resource is statically named and Maté programs are short, the set of required locks can be quickly determined with a full program analysis. Program writers can explicitly yield or release locks to improve parallelism.

One goal of Maté is to provide a programming interface to motes that is much simpler than TinyOS; a sense-and-send program can be as few as six instructions in Maté, and Maté's error detection mechanisms can help novice programmers find the bugs in their programs.

Maté is a architecture for constructing application specific virtual machines. Using the architecture, developers can easily change instruction sets, execution events, and VM subsystems. The first half of this document presents Bombilla, a specific instance of a Maté VM that is part of the standard TinyOS distribution, explaining it as a complete system. The second half explains the Maté architecture, shows how Bombilla is an instance of Maté, and gives examples on how Bombilla can be used as a template to build other virtual machines.

2 Bombilla

Bombilla is a set of TinyOS components that sit on top of several system components, including sensors, the network stack, and non-volatile storage (the "logger")[1]. Code is broken in **capsules** of 24 instructions, each of which is a single byte long; larger programs can be composed of multiple capsules. In addition to bytecodes, capsules contain identifying and version information. Each Bombilla context has two stacks: an operand stack and a return address stack. Most instructions operate solely on the operand stack, but a few instructions control program flow and several have embedded operands. There are three execution contexts that can run concurrently at instruction granularity. Bombilla provides both a built-in ad-hoc routing algorithm (the `send` instruction) as well as mechanisms for writing new ones (the `sendr` instruction).

2.1 Architecture, Instruction Set, and Data Types

Bombilla instructions hide the asynchrony (and oft-resulting race conditions) of TinyOS programming. For example, when the `send` instruction is issued, Bombilla calls a command in the ad-hoc routing component

[1]Support for the logger is currently not implemented, due to the pending addition of a new logger interface in TinyOS.

basic	`00iiiiii`	`i` = instruction
m-class	`010iixxx`	`i` = instruction, `x` = argument
v-class	`011ixxxx`	`i` = instruction, `x` = argument
j-class	`10ixxxxx`	`i` = instruction, `x` = argument
x-class	`11xxxxxx`	`i` = instruction, `x` = argument

Figure 2: Bombilla Instruction Formats

to send a packet. Bombilla suspends the context until a message send complete event is received, at which point it resumes execution. By doing this, Bombilla does not need to manage message buffers – the capsule will not resume until the network component is done with the buffer. Similarly, when the `sense` instruction is issued, Bombilla requests data from the sensor TinyOS component and suspends the context until the component returns data with an event. This synchronous model makes application level programming much simpler and far less prone to bugs than dealing with asynchronous event notifications. Additionally, Bombilla efficiently uses the resources provided by TinyOS; during a split-phase operation, Bombilla does nothing on behalf of the calling context, allowing TinyOS to put the CPU to sleep or use it freely.

Bombilla is a stack-based architecture. This allows a concise instruction set; most instructions do not have to specify operands, as they exist on the operand stack. There are five classes of Bombilla instructions: basic, m-class, j-class, v-class, and x-class. Figure 2 shows the instruction formats for each class. Basic instructions include such operations as arithmetic, halting, and activating the LEDs on a mote. m-class instructions access message headers; they can only be executed within the message send and receive contexts. v-class instructions access the 16 word Bombilla heap. j-class instructions are the two jump instructions, for loops and conditionals. The one x-class instructions is `pushc` (push constant). All instruction classes except basic have embedded operands..

Bombilla's three principal execution contexts, illustrated in Figure

1, correspond to three events: clock timers, message receptions and message send requests. Inheriting from languages such as FORTH, each context has two stacks, an operand stack and a return address stack. The former is used for all instructions handling data, while the latter is used for subroutine calls. The operand stack has a maximum depth of 16 while the call stack has a maximum depth of 8. We have found this more than adequate for programs we have written.

There is an additional context, the "once" context. Unlike other contexts, which run their capsules many times, this context only runs its capsule once, when it is installed; this allows a user to initialize state, adjust constants, or perform other operations that only need a single execution.

There are three operands types: values, sensor readings, and buffers. Some instructions can only operate on certain types. For example, the `putled` instruction expects a value on the top of the operand stack. However, many instructions are polymorphic. For example, the `add` instruction can be used to add any combination of the types, with different results. Adding buffers results in appending the data in the second operand onto the first operand. Adding a value to a message appends the value to the message data payload. Sensor readings can be turned into values with the `cast` instruction.

Sensor readings are typed, and cannot be modified. For example, in order to take an average over a set of readings, each reading must be first be converted to a value; these values can then be averaged. Many instructions (e.g. `inv`) automatically cast sensor readings to values. This ensures that a sensor reading variable has some meaning; otherwise, it could express some arbitrary quantity. Adding two sensor readings of the same type (e.g. magnetometer X-axis) produces a value, and adding two sensor readings of different values is an error.

Buffers are also typed, and can hold up to ten values. A buffer can only hold elements of one type, whether they be values or a certain sensor type. An empty buffer has no type; the first element added will set the type of the buffer. Buffers have several access instructions, including `bhead` (copy of the first element of the buffer onto the operand stack), `byank` (pull the nth element out of the buffer and push it into the operand stack), and `bsorta` (sort the elements in ascending order.

There is a 16 word heap shared among the context. It can be accessed with the `setvar` and `getvar` instructions, which have a 4-bit embedded operand.. This allows the separate contexts to communicate shared state (e.g. sensor readings).

```
getvar 0    # Get heap variable 0
pushc 1     # Push one onto operand stack
add         # Add the one to the stored counter
copy        # Copy the new counter value
setvar 0    # Set heap variable 0
pushc 7
and         # Take bottom three bits of counter
putled      # Set the LEDs to these three bits
halt
```

Figure 3: Bombilla`cnt_to_leds` – Shows the bottom 3 bits of a counter on mote LEDs

```
pushc 1     # Push one on the operand stack
sense       # Read sensor 1 (light)
copy        # Copy the sensor reading
getvar 0    # Get previous sent reading

inv         # Invert previous reading
add         # Current - previous sent value
copy        # 2 copies of difference on top of stack
pushc 32    #

gt          # Is current 32 greater than previous?
swap        # Swap result with copy
pushc 32    #
inv         #

lt          # Is current 32 less than previous?
or          # Either  32> or 32<
jumps 15    # Jump to send
halt

copy        # Copy new value
setvar 0    # Set current
bpush0      # Push buffer 0 onto stack
bclear      # Clear its contents

add         # Add current reading to buffer
send        # Send buffer
halt
```

Figure 4: Bombilla Program to Read Light Data and Send a Packet on Reading Change

2.2 Capsules and Execution

Bombilla programs are broken up into capsules of up to 24 instructions. This limit allows a capsule to fit into a single TinyOS packet. By making capsule reception atomic, Bombilla does not need to buffer partial capsules, which conserves RAM. Every code capsule includes type and version information. Bombilla defines two types of code capsules: context capsules, which are the root execution of a context, and subroutine capsules, which can be called from context capsules or other subroutine capsule. Subroutine capsules allow programs to be more complex than what fits in a single capsule. Applications invoke and return from subroutines using the `call` and `return` instructions. There are names for up to 2^{15} subroutines; to keep Bombilla's RAM requirements small, its current implementation has only four.

Bombilla begins execution in response to an event – a timer going

off, a packet being received, or a packet being sent. Each of these events has a capsule and an execution context. Control jumps to the first instruction of the capsule and executes until it reaches the `halt` instruction. These three contexts can run concurrently. Each instruction is executed as a TinyOS task, which allows execution interleaving at an instruction granularity. Additionally, underlying TinyOS components can operate concurrently with Bombilla instruction processing. When a subroutine is called, the return address (capsule, instruction number) is pushed onto a return address stack and control jumps to the first instruction of the subroutine. When a subroutine returns, it pops an address off the return stack and jumps to the appropriate instruction.

The packet receive and clock capsules execute in response to external events; in contrast, the packet send capsule executes in response to the `sendr` instruction. As `sendr` will probably execute a number of Bombilla instructions in addition to sending a packet, it can be a lengthy operation. Therefore, when `sendr` is issued,

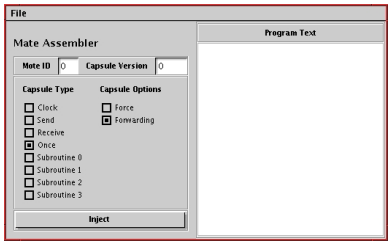

Figure 5: CapsuleInjector GUI

Bombilla copies the message buffer onto the send context operand stack and schedules the send context to run. Once the message has been copied, the calling context can resume execution. The send context executes concurrently to the calling context, preparing a packet and later sending it. This frees up the calling context to handle subsequent events – in the case of the receive context, this is very important.

The constrained addressing modes of Bombilla instructions ensure a context cannot access the state of a separate context. Every push and pop on the operand and return value stack has bound checks to prevent overrun and underrun. As there is only a single shared variable, heap addressing is not a problem. Unrecognized instructions result in simple no-ops. All bounds are always checked – the only way two contexts can share state is through `gets` and `sets`. Nefarious capsules can at worst clog a network with packets – even in this case, a newer capsule will inevitably be heard. By providing such a constrained execution environment and providing high-level abstractions to services such as the network layer, Bombilla ensures that it is resilient to buggy or malicious capsules.

2.3 Simple Programs

The Bombilla program in Figure 3 maintains a counter that increments on each clock tick. The bottom three bits of the counter are displayed on the three mote LEDs. The counter is kept as a value which persists at the top of the stack across invocations. This program could alternatively been implemented by using `gets` and `sets` to modify the shared variable. This code recreates one of the simplest TinyOS applications, `cnt_to_leds`, implemented in seven bytes.

The Bombilla program in Figure 4 reads the light sensor on every clock tick. If the sensor value differs from the last sent value by more than a given amount (32 in this example), the program sends the data using Bombilla's built-in ad-hoc routing system. This program is 24 bytes long, fitting in a single capsule.

2.4 Capsule Injector

A tool is included in the TinyOS release to aid in the writing of
Bombilla programs: `net.tinyos.vm_asm.CapsuleInjector`.

`CapsuleInjector` provides an interface for writing assembly programs and sending them to a mote connected to a PC; if the capsule is marked self-forwarding, it will then start propagating into the network.

One must set the destination mote ID of the capsule packet (important when using a GenericBase) and the version number of the capsule. Version numbers are explained in Section 2.6; Bombilla only installs a capsule if its version number is higher than the one it currently has.

If the program has an error (e.g. an unknown instruction), `CapsuleInjector` does not send out a packet.

2.5 Synchronization

Bombilla interleaves the execution of its contexts at instruction granularity. The presence of a 16-word shared heap means that if different contexts communicate or share variables (e.g. an aggregated sensor reading), race conditions can easily occur. As the program running on a mote is the combination of possibly forwarding capsules, applications can go through transient states of partial installation, making programmer efforts (e.g. a spin loop) ineffectual.

Bombilla therefore uses an implicit locking scheme, so that programmers are assured that there will be no race conditions in their programs. Experienced programmers can relax the locking requirements to improve parallelism.

2.5.1 Model

The Bombilla sychronization model is based on five abstractions: *handlers, invocations, resources, scheduling points* and *sequences*. We describe how we discover the resources used by an invocation, and how invocation communicate their resource requirements to the runtime system.

A *handler* is a function that is executed in response to some event. An *invocation* represents a particular execution of a handler in response to an event. At any time, invocations are in one of four states: *waiting* (for resources), *suspended* (waiting for an operation to complete), *ready* (can execute), *running* (executing). We say that an invocation that is ready or running is *active*.

A *resource* is a shared piece of state that a handler requires access to – examples are a variable, a disk arm, or a sensor. Resources can only be held by invocations.

A handler contains a number of *scheduling points* at which it can be suspended and gain or lose resources (and resources cannot be acquired anywhere but at scheduling points). Scheduling points are the handler's entry and exit points, and some subset of its operations which we call *scheduled operations*. An invocation goes through two states during execution of a scheduled operation: first, the invocation is suspended awaiting the completion of the operation; second, the invocation is waiting for the resources it wishes to gain to become available. Either of these two phases may be trivial: a `yield` operation completes immediately but may wait for some resources, a message send does not complete immediately but, if it is not waiting for any resources, will not wait. A *sequence* is any code path between two scheduling points which does not include another scheduling point.

The model for resource acquisition and release is as follows: before an invocation can start execution, it must acquire all resources it will need during its lifetime. At each subsequent scheduling point, the invocation can release a set R of resources before suspending execution, and acquire a set A of resources before resuming. To prevent deadlock, we require $A \subseteq R$ (we prove below that this condition is sufficient for building deadlock-free schedulers). Finally, when an invocation exits it must release all held resources. Note that we do not guarantee any atomicity between two invocations of the same handler.

To preserve safety, the static analysis of a handler's resource usage and the runtime system must guarantee that an invocation holds all resources at the time(s) at which they are accessed and that the intersection of the resources held by any two invocations is empty. We restrict our invocation model to considering a static number of resources, and require operations to explicitly name the resources they use so that we can easily analyse handlers at compile (or load) time. Resource discovery must be conservative to preserve correctness.

2.5.2 Bombilla Implementation

We implemented this syncronization model in Bombilla. Each Bombilla context is an invocation, and capsules are implicitly broken up into sequences. Bombilla maintains two queues of invocations: ready and waiting. Whenever Bombilla installs a new capsule, it performs a static full-program analysis to generate the acquire sets of its invocation start points. Without requiring any annotation from a programmer, Bombilla runs invocations atomically while allowing parallelism. Programmers can improve the degree of parallelism by yielding resources at scheduling points.

Bombilla invocations are broken into sequences by scheduling point instructions. When a context executes one of these instructions, the Bombilla runtime examines the current release set of the issuing context and releases the locks on the indicated resources. Bombilla then checks the waiting queue to see if any contexts have been made runnable by the release of these locks. When the scheduling point instruction completes, Bombilla checks the acquire set of the context to see if it can be made active; if so, the context acquires its locks and Bombilla places it on the ready queue. If the context cannot be made active, Bombilla places it on the waiting queue.

Bombilla defines its scheduling point instructions to be those that trigger split-phase operations in TinyOS. This includes acquiring sensor data (`sense`), sending packets (`send, sendr, uart`), and accessing non-volatile flash memory storage (`logr, logw, logwl`). Additionally, there is a `yield` instruction, which is effectively a split-phase operation that immediately completes. Locks are added to a context's release set

```
pushc 3  #                          |  pushc 3
unlock   # Add lock 3 to R,A        |  unlockb   # Add locks 0,1 to R,A
pushc 2  #                          |  pushc 12
punlock  # Add lock 2 to R          |  punlockb  # Add locks 2,3 to R
sense    # Yield                    |  sense     # Yield
```

Figure 6: Sample Bombilla Unlocking

with the `unlock` instruction – by default, the set is empty. `unlock` also adds the lock to the context's acquire set. For a lock to be released, but not re-acquired, a context much use the `punlock` (unlock permanent) instruction. The `unlock` and `punlock` instructions affect individual locks, enumerated by an operand; the `unlockb` and `punlockb` instructions use the operand as a bitmask for locks to be released. Locks are not released until a scheduling-point instruction is executed. Figure 6 contains two sample Bombilla instruction sequences that demonstrate resource unlocking.

Release and acquire sets are atomically handled by Bombilla. A context does not acquire any locks in its acquire set unless it can acquire all of them, and acquires them atomically. Similarly, release sets are released atomically. This, combined with monotonically decreasing lock sets, ensures the system is deadlock free.

2.6 Viral Programming

Bombilla code capsules can be marked "forwarding." Capsules marked forwarding are automatically forwarded by the viral propagation subsystem (BVirus) of the VM.

The first implementation of the VM had an explicit caspule forwarding system (the `forw` instruction); experimental results showed this to be a terrible idea, as programs could very easily saturate the network unknowingly. We therefore adopted this simple probabilistic model. It is by no means perfect; for example, even if every mote in the network is running the same capsule, they will continue to forward it indefinitely.

Every capsule has a version number. When Bombilla hears a capsule broadcast, it checks if the capsule is newer than the one currently installed; if so, Bombilla halts execution of that context and installs the new capsule.

Our first implementation of the VM had an explicit caspule forwarding system (the `forw` instruction); experimental results showed this to be a terrible idea, as programs could very easily saturate the network unknowingly. We therefore adopted this simple probabilistic model. It is by no means perfect; for example, even if every mote in the network is running the same capsule, they will continue to forward it indefinitely.

Currently, Bombilla uses a density-adjusting algorithm; every mote maintains a time interval of length τ, and picks a random point t in tau in which to transmit a summary of capsule versions. If the mote hears an identical version summary before t, it suppresses its transmission. This mechanism controls the number of version summaries sent within a cell during any time period tau, keeping it to a small constant. The algorithm also scales tau to achieve low overhead when the network is stable and high reprogramming rates when there is new code. When a node hears a version summary with older capsules than it has, it broadcasts the needed capsules.

2.7 Error State

Bombilla has an error state, which can help users debug their programs. If a program triggers an error (for example, by trying to add incompatible sensor readings, or by overflowing the operand stack), Bombilla halts execution on all contexts. Then, every second, it blinks all of the LEDs and sends a packet containing debugging information over the UART. The packet contains the offending context identifier, the capsule it was executing, the instruction that caused the error, and the error code. Error codes can be found in `Bombilla.h`. They are:

```
typedef enum {
  BOMB_ERROR_TRIGGERED               =  0,
  BOMB_ERROR_INVALID_RUNNABLE        =  1,
  BOMB_ERROR_STACK_OVERFLOW          =  2,
  BOMB_ERROR_STACK_UNDERFLOW         =  3,
  BOMB_ERROR_BUFFER_OVERFLOW         =  4,
  BOMB_ERROR_BUFFER_UNDERFLOW        =  5,
```

```
  BOMB_ERROR_INDEX_OUT_OF_BOUNDS      =  6,
  BOMB_ERROR_INSTRUCTION_RUNOFF       =  7,
  BOMB_ERROR_LOCK_INVALID             =  8,
  BOMB_ERROR_LOCK_STEAL               =  9,
  BOMB_ERROR_UNLOCK_INVALID           = 10,
  BOMB_ERROR_QUEUE_ENQUEUE            = 11,
  BOMB_ERROR_QUEUE_DEQUEUE            = 12,
  BOMB_ERROR_QUEUE_REMOVE             = 13,
  BOMB_ERROR_QUEUE_INVALID            = 14,
  BOMB_ERROR_RSTACK_OVERFLOW          = 15,
  BOMB_ERROR_RSTACK_UNDERFLOW         = 16,
  BOMB_ERROR_INVALID_ACCESS           = 17,
  BOMB_ERROR_TYPE_CHECK               = 18,
  BOMB_ERROR_INVALID_TYPE             = 19,
  BOMB_ERROR_INVALID_LOCK             = 20,
  BOMB_ERROR_INVALID_INSTRUCTION      = 21
} BombillaErrorCode;
```

The Bombilla error packet has the following payload:

```
typedef struct BombillaErrorMsg {
  uint8_t context;
  uint8_t reason;
  uint8_t capsule;
  uint8_t instruction;
} BombillaErrorMsg;
```

3 Maté

Bombilla is an instance of the Maté virtual machine. Maté's architecture has many similarities to TinyOS; the core of the VM is a simple component that schedules contexts to execute. nesC wiring allows developers to easily customize and change the VM contexts, instructions, and subsystems.

3.1 Component Architecture

The core Maté component, `BombillaEngine`, receives requests from contexts to be scheduled and executes them at a granularity of up to a single Maté instruction. The contexts themselves, the instruction set, and the VM services are all separate from the VM scheduler.

A TinyOS component provides every Maté instruction. While most instruction components implement only one instruction, some implement several related instructions. For example, the shared memory access instructions, `getvar` and `setvar`, are both provided by a single component; it's unlikely that only one of the two would ever be needed. Instruction components provide the `BombillaBytecode` interface, and `BombillaEngine` uses that interface with an 8-bit parameter: each instruction is a single byte. The set of instruction components wired to `BombillaEngine` specifies the instruction set.

To be specific, this is the interface signature of `BombillaEngine`:

```
configuration BombillaEngine {
  provides {
    interface StdControl;
    command result_t computeInstruction(BombillaContext* context);
    command result_t executeContext(BombillaContext* context);

    interface BombillaContextComm;
  }
  uses interface BombillaBytecode as Bytecode[uint8_t bytecode];
}
```

and this is a snippet of Bombilla's configuration:

```
configuration AbstractMate {}
implementation {
  components BombillaEngine as VM;
  components OPhalt, OPputled, OPcopy, OPadd, OPland, OPlor, OPlnot;

  ...

  VM.Bytecode[OPhalt] -> OPhalt;
  VM.Bytecode[OPputled] -> OPputled;
  VM.Bytecode[OPadd] -> OPadd;
  VM.Bytecode[OPcopy] -> OPcopy;
  VM.Bytecode[OPland] -> OPland;
  VM.Bytecode[OPlor] -> OPlor;
  VM.Bytecode[OPlnot] -> OPlnot;

  ...
}
```

The bytecode wirings use the name of a bytecode twice, as a component and as a constant from an enum. For example, `VM.Bytecode[OPadd] -> OPadd;` means "wire the OPadd **component** to the instance of Bytecode specified by the enum **constant** OPadd." The constant is specified in `Bombilla.h`.

something similar is done for contexts

By encapsulating instructions into components, TinyOS allocates storage only for used systems. For example, the `sense` instruction keeps some state on the sensor being sampled and maintains a wait queue of contexts. Similarly, subroutine capsules are part of the `call` instruction; if a VM can't call the subroutines, there's no need to waste RAM on them.

3.2 Service Proxies

Instructions often need to use several VM subsystems; for example, almost every instruction manipulates the operand stack, requiring a component that provides the `BombillaStacks` interface. This raises the issue of where these wirings are made. On one hand, wiring at the top-level configuration is laborious: up to 256 instructions, with 2-3 used interfaces each, leads to a very large file. On the other, wiring at the instruction component level makes consistency difficult: it's possible that two instructions wire in two different implementations of a VM service.

To solve this problem, every instruction is a configuration, but instead of wiring used interfaces to a specific component, they're wired to proxy configurations, such as `BStacksProxy`. Every instruction has a module, e.g. `OPhaltM`, and a configuration, e.g. `OPhalt`. The configuration wires all of the module's used interfaces to these proxies. For example, every instruction modules that uses the `BombillaStacks` interface has a configuration that has something like:

```
  Instr.BombillaStacks -> BStacksProxy;
```

These proxy configurations are simple "pass-through" configurations. For example:

```
configuration BStacksProxy {
  provides {
    interface BombillaStacks;
    interface BombillaTypes;
  }
}
implementation {}
```

They do not actually provide a service; instead, they represent a single, common wiring point for all users of that service. Then, the top-level VM configuration can pick an implementation of that service and wire it to the proxy:

```
configuration AbstractMate{}
implementation {
  components BStacks, BStacksProxy;
```

```
  BStacksProxy.BombillaStacks -> BStacks;
  BStacksProxy.BombillaTypes -> BStacks;
}
```

This abstraction is necessary for when there are multiple versions of a service, and a single implementation needs to be chosen for the entire VM. For example, there are two providers of `BombillaLocks`: `BLocks` and `BLocksSafe`. The former is a fast and efficient implementation, while the latter performs additional checks in case callers have errors in their logic (e.g., unlocking locks they do not hold). The implementation can be changed by modifying a single line in the VM configuration.

3.3 Instruction Examples

Every instruction component must provide the `BombillaBytecode` interface, which the `BombillaEngine` calls when it reaches the associated opcode in a program. Perhaps the simplest Bombilla instruction is `pop`, which pops an operand off the operand stack. This is the complete code for the `OPpopM` module:

```
includes Bombilla;
includes BombillaMsgs;

module OPpopM {
  provides interface BombillaBytecode;
  uses interface BombillaStacks as Stacks;
}

implementation {

  command result_t BombillaBytecode.execute(uint8_t instr,
                                            BombillaContext* context,
                                            BombillaState* state) {
    dbg(DBG_USR1, "VM (%i): Popping top operand off of stack. \n", context->which);
    call Stacks.popOperand(context);
    return SUCCESS;
  }
}
```

Its configuration (`OPpop`) is:

```
includes Bombilla;
includes BombillaMsgs;

configuration OPpop {
  provides interface BombillaBytecode;
}

implementation {
  components OPpopM, BStacksProxy;

  BombillaBytecode = OPpopM;

  OPpopM.Stacks -> BStacksProxy;
}
```

3.3.1 Error Checking

Most of the Maté service components automatically perform checks; for example, if `BStacks` is told to pop off an empty stack, it triggers an error condition (`BOMB_STACK_UNDERFLOW`). Error conditions are triggered through the `BErrorProxy` component. Sometimes, however, instructions have additional checks; for example, the `bfull` instruction takes a buffer as an operand. If the top operand is not a buffer, it triggers an error condition.

```
includes Bombilla;
includes BombillaMsgs;

module OPbfullM {
  provides interface BombillaBytecode;
  uses interface BombillaStacks as Stacks;
  uses interface BombillaTypes as Types;
}

implementation {

  command result_t BombillaBytecode.execute(uint8_t instr,
                                             BombillaContext* context,
                                             BombillaState* state) {
    BombillaStackVariable* arg = call Stacks.popOperand(context);
    dbg(DBG_USR1, "VM (%i): Checking if buffer full.\n", (int)context->which);

    if (!call Types.checkTypes(context, arg, BOMB_VAR_B)) {return FAIL;}
    call Stacks.pushOperand(context, arg);
    call Stacks.pushValue(context, (arg->buffer.var->size == BOMB_BUF_LEN)? 1: 0
);
    return SUCCESS;
  }
}
```

If `Types.checkTypes()` fails, it triggers an error condition in the calling context. The final parameter is a bitmask of valid types: `BOMB_VAR_B` for buffer, `BOMB_VAR_S` for sensor, and `BOMB_VAR_V` for value. These can be combined; for example, `(BOMB_VAR_B | BOMB_VAR_V)` will succeed for buffers or values.

Error conditions can also be triggered explicitly. For example, the comparison instructions `gte, gt, lt, lte, eq` all require two operands of the same type. The code for `gte` is as follows:

```
module OPgteM {
  provides interface BombillaBytecode;
  uses interface BombillaStacks as Stacks;
  uses interface BombillaError;
}

implementation {

  command result_t BombillaBytecode.execute(uint8_t instr,
                                             BombillaContext* context,
                                             BombillaState* state) {
    BombillaStackVariable* arg1 = call Stacks.popOperand(context);
    BombillaStackVariable* arg2 = call Stacks.popOperand(context);

    if ((arg1->type == BOMB_VAR_V) &&
        (arg2->type == BOMB_VAR_V)) {
      call Stacks.pushValue(context, arg2->value.var > arg1->value.var);
    }
    else if ((arg1->type == BOMB_VAR_S) &&
             (arg2->type == BOMB_VAR_S) &&
             (arg1->sense.type == arg2->sense.type)) {
      call Stacks.pushValue(context, arg2->sense.var > arg1->sense.var);
    }
    else {
      call BombillaError.error(context, BOMB_ERROR_INVALID_TYPE);
      return FAIL;
    }
    return SUCCESS;
  }
}
```

where `BombillaError.error` is explicitly called at the end.

3.3.2 Split Phase Operations

Generally, instructions that merely manipulate operands are fairly simple, as the examples above show. Instructions that encapsulate split-phase operations (such as `sense` and `send`) are a bit more complex, as these instructions cause contexts to block. As split-phase operations are scheduling points, they can also yield locks.

The basic pseudocode for a scheduling point instruction is this:

```
if another context is using the resource
    put caller on wait queue
else if underlying resource is busy
    put caller on wait queue
else
    start split-phase operation
    set context state to appropriate value
    set active context as one executing operation
    call Synch.releaseLocks
    call Synch.yieldContext
```

`Synch.releaseLocks()` releases all of the locks a context has set to yield with, for example, the `unlock` or `punlock` instructions. `Synch.yieldContext()` then takes the context off the run queue, and checks if there are any contexts made runnable by yielded locks. Putting a caller on a wait queue also requires restoring it to its state before it executed this instruction, so it can retry later.

The corresponding event of the split-phase operation is then responsible for resuming the blocked context, and pulling a waiting context off the wait queue.

```
if no active context
    return
put active context in run state
call Synch.resume
make any necessary operand stack operations
clear active context
if wait queue is not empty
    dequeue context from wait queue
    call Synch.resume
```

The second `Synch.resume` will cause the waiting context to execute its next instruction, which will be the one that executes the split-phase operation.

For examples of this logic, look at `OPsense` and `OPsend`.

3.3.3 Embedded Operands

Some instructions, such as `pushc`, have embedded operands. Instruction components that expect embedded operands have a number at the end of their name, specifying the bit width of the expected value. For example, `OPcall2` says that the `call` instruction expects two bits of operand, while `OPpushc6` says that `pushc` expects six. These instructions take up more than one slot in a VM's instruction set. For example,

```
VM.Bytecode[OPcall0] -> OPcall2;
VM.Bytecode[OPcall0+1] -> OPcall2;
VM.Bytecode[OPcall0+2] -> OPcall2;
VM.Bytecode[OPcall0+3] -> OPcall2;
```

The code of `call` then reads:

```
command result_t BombillaBytecode.execute(uint8_t instr,
                                          BombillaContext* context,
                                          BombillaState* state) {
  dbg(DBG_USR1, "VM (%i): Calling subroutine %hhu\n", (int)context->which, (ui
nt8_t)(instr & 0x3));
  call Stacks.pushReturnAddr(context);
  context->capsule = &(state->capsules[instr & 0x3]);
  context->pc = 0;
  return SUCCESS;
}
```

3.4 Contexts

Maté contexts are essentially what is described in the Bombilla section; they have two stacks, etc. Making each instruction a component allows a user to customize the VM instruction set. Contexts follow a similar model: each context is a separate component, which is wired to `BombillaEngine`. The component handles the event that triggers the context, and sets it runnable. The context component is responsible for both the context state and the context capsule; as with instructions, this means the VM only needs memory for contexts that are used. Bombilla has four contexts: clock, send, receive, and once. These implementations can be used as templates for new contexts.

The viral propagation subsystem, `BVirusProxy`, requires that components register capsules with it when the VM boots. This allows it to easily generate version vectors and install new code. When new code is installed, all running contexts must halt and reset.

4 Customizing Maté

The easiest way to customize Maté is to start from a working VM and modify it. For example, let's say your application needs to take square roots; implementing this in an instruction is much more efficient than trying to code it in Maté bytecodes.

The first step is to write the `OPsqrt` component. `sqrt` pops a single value operand off the stack, takes its square root, and pushes the result back onto the stack. This component provides one interface, `BombillaBytecode`, uses two: `BombillaStacks` to push and pop, and `BombillaTypes` to check that the top of the stack is a value.

The resulting signature for the module is:

```
includes BombillaMsgs;

module OPsqrtM {
  provides interface BombillaBytecode;
  uses {
    interface BombillaStacks as Stacks;
    interface BombillaTypes as Types;
  }
}
```

The component only implements one function, `execute`:

```
implementation {

  command result_t BombillaBytecode.execute(uint8_t instr,
                                            BombillaContext* context,
                                            BombillaState* state) {
    BombillaStackVariable* arg = call Stacks.popOperand(context);
    dbg(DBG_USR1, "VM (%i): Taking squart root of top of stack.\n", (int)context->which);
    if (!call Types.checkTypes(context, arg, BOMB_VAR_V)) {return FAIL;}
    arg->value.var = (int16_t)sqrt(arg->value.var)
    call Stacks.pushOperand(context, arg);
    return SUCCESS;
  }
}
```

The module needs a configuration, which wires it to the appropriate proxies:

```
includes Bombilla;
includes BombillaMsgs;

configuration OPsqrt {
  provides interface BombillaBytecode;
}
implementation {
  components OPsqrtM, BStacksProxy;

  BombillaBytecode = OPinvM;
  OPsqrtM.Stacks -> BStacksProxy;
  OPsqrtM.Types -> BStacksProxy;
}
```

The `OPsqrt` component is now a functioning instruction. The final step is to include it in the VM. There are two steps to this: the first is to define the opcode value of the instruction (probably replacing another instruction), and the second is to wire it to the VM.

The first is accomplished by modifying the application's `BombillaOpcodes.h`. In this case, we'll remove the `cpull` instruction. Do this by changing the line `OPcpull = 0x11` to `OPsqrt = 0x11`. Then, in the top-level VM configuration, remove the component `OPcpull`, adding `OPsqrt`, and the line `VM.Bytecode[OPcpull] -> OPcpull;`, replacing it with `VM.Bytecode[OPcpull] -> OPsqrt;`

Now, when `BombillaEngine` encounters the opcode `0x11`, it will execute the square root instruction.

4.1 Customizing CapsuleInjector

By default, `CapsuleInjector` uses the Bombilla configuration for its opcodes and contexts. Changing contexts and capsule options requires changing `CapsuleInjector`'s code, but the instruction set can be changed with much less work.

`CapsuleInjector` uses the java class `BombillaConstants` to assemble instructions to binary opcodes. `BombillaConstants` is automatically generated using `ncg`; if you change the `.h` file that `BombillaConstants` is generated from, then `CapsuleInjector` will recognize a different set of instructions.

`CapsuleInjector` recognizes constants whose name begins with `OP` as opcodes. For example, when it reads the instruction `jumpc` from a program, it looks for a constant named `OPjumpc` and translates it to the corresponding value.