

The MACBETH Specification, Modeling and Programming Language*

Carlos Puchol †
Transmeta Corporation
cpg@transmeta.com

Abstract

We introduce MacBeth, a language for the behavioral specification and programming of real-time applications. The aim of the language is to provide a way to specify systems in which timing and state are first-class objects. It does so at a higher level of abstraction, in a general way, yet in a formal, concise and efficient fashion.

1 Introduction

One of the main purposes behind the introduction of a new programming language is to allow the user to specify entities in the “problem space” more directly and closer to the problems to be solved or implemented; in short, at a higher level of abstraction with respect to the domain. Although programs have representations in bits and machine code, it has been long recognized that abstraction is a fundamental feature for achieving practical use of a language. In the realm of reactive and real-time systems, devices and applications typically have been programmed in fairly low-level languages, much closer to the hardware and operating system than to the system being programmed.

We introduce MacBeth, a language for specification and programming real-time systems, that is, the domain of problems that MacBeth aims at is the behavioral specification and programming of real-time applications, in which timing and state are first-class objects. The aim of the language is to provide a way to do so at a higher level of abstraction, in a general way, yet in a formal, concise and efficient fashion.

MacBeth is designed to be amenable for analysis: the purpose of MacBeth is not just to produce better code but to produce *correct* code faster and in a structured, modular way. It is fundamentally a descendant of the family of synchronous programming languages [6, 1, 2, 4], but introduces several new concepts. On top of the concurrency, synchrony, hierarchy and broadcast communication that synchronous languages feature, MacBeth provides new or substantially refined features, such as:

- Asynchronous communication and execution among components,
- Full language support for the execution of periodic and event-driven tasks,

- Extended, flexible transition expressions,
- Simplification of basic common constructs and new elements for representing common features, such as exceptions, alarms, interrupt handlers, etc., and
- Encapsulation, modularization, more structuring and parameterization of modes.

MacBeth is a language based on a representation of the state space of the system that provides synchronous and asynchronous constructs both for communication and execution. It lets the user control the parts of the specification that are synchronous and the parts that are asynchronous in a flexible way. It is based on notions of *processes*, *channels*, *modes* and *transitions*. The *processes* represent asynchronously executing structural entities in the system, and *channels* represent synchronization and data transfer between processes. The *modes* represent partitions of the state space of processes, and *transitions* represent mode changes in the state space (i.e., execution control flow).

The approach is more functional in nature than traditional state-based formalisms and it is more structured, emphasizing modularity. The language encourages programmers to take advantage of the hierarchy, but users can still make full use of sophisticated state-transition behaviors.

From a practical point of view, the concerns are suitability of the language for specification, programming and implementation of real-world reactive and real-time systems and protocols. From an analytical point of view, the concerns are its formal semantics and the suitability for formal analysis.

The purpose of this paper is to overview MacBeth informally by touching on the major part of the language. Section 2 discusses the computation model assumed by MacBeth and where it fits in the spectrum of formalisms for protocols and real-time/reactive systems. Section 3 introduces MacBeth by a series of examples. Section 4 offers a summary of the language features and future avenues for work. The full report on MacBeth [9] contains more details, specifically, it contains more details concerning the formal underpinnings of the language.

2 MacBeth Model

Synchrony vs. Asynchrony

The spectrum of formalisms for protocols, reactive/real-time, parallel and concurrent finite state systems can be

*Work performed while at Bell Labs, Lucent Technologies.

†Address: 3940 Freedom Circle, Santa Clara, CA 95054, USA.

roughly classified in two groups, based on the main underlying computation model they use: the synchronous approach (Modechart [6], Esterel [1], Lustre [2] and S/R [7]) and the asynchronous approach (UNITY [3], SPIN [5] and VFSM [10]).

MacBeth provides an integrated approach for modeling *both* aspects of a system. These aspects differ in the execution model for parallel and concurrent processes. Processes are said to execute asynchronously when no assumption is made about their relative execution rates or speeds. In the synchronous approach, process execution takes place theoretically instantaneously, with respect to their environment—they execute like parallel FSMs (finite state machines), in lockstep.

Specification vs. Programming

We have referred to MacBeth as a *specification, modeling and programming* language. Some formalisms are modeling languages and provide no path to implementation. Others do not provide anything but a path to implementation (programming languages).

MacBeth provides a single language suitable to specify both the environment as well as the system implementation. Section 3 shows some examples that are perfectly implementable in a deterministic computer system (we call these programs) and others that are not implementable (we call these specifications). Typically, a complete specification will contain a part that is subject to implementation and the rest is supporting the assumptions made about the environment surrounding that part. Some other times, programmers may prefer to start with a very abstract version of the system that is not implementable and proceed via stepwise refinement until they accomplish their implementation goals.

MacBeth provides constructs to express non-deterministic choices in outgoing transitions, as well as constructs to denote static priorities among them. MacBeth also offers support for hard timing constraints as well as unbounded timing constraints allowing transitions to be taken based on, for example, *at least T* time units, or *at most T* time units.

3 MacBeth by Example

We now start introducing MacBeth by code fragments that illustrate some of the features of the language. The syntax of the language can be found in [9].

3.1 Modes

Modes are the basic building blocks for designing MacBeth programs and specifications. They represent partitions of the state space of the system. The mode structure of a MacBeth program describes a hierarchical arrangement of the state space of the system under specification. Transitions from modes to modes allow the specification of changes in the control flow and have labels denoting expressions that may trigger the transition. The simplest mode represents states in the system. Several modes can be combined to form other modes enclosing them in their *parent* mode, which determines

their *execution behavior*, acting as either a *parallel* or a *serial* operator: serial modes operate in exactly one of the child modes at any time (one of their children must be designated as the *initial* mode), and *parallel*, where all sub-modes are active simultaneously.

The simplest MacBeth program is a single (anonymous) state with no sub-states and no transitions out of it. This code fragment just halts: `{ }`. Consider now the example of a motor controller program consisting of a serial mode called `controller` with an `active` mode and an `inactive` mode. The `controller` mode is a serial mode (by default if not otherwise specified):

```
mode controller {
  mode inactive { ... }
  mode active { ... }
}
```

The `inactive` mode is entered when `controller` is entered, since it is the initial mode of `controller`, having been specified first inside it. We now expand the code segment above to show the inputs, `ReqOn` and `ReqOff`, and one output, called `Motor`.

```
mode controller {
  input ReqOn, ReqOff;
  output Motor = ^active;

  mode inactive { when ReqOn -> active; }
  mode active { when ReqOff -> inactive; }
}
```

Inputs and outputs in MacBeth are similar to signals in Esterel and wires in hardware description languages like S/R. The `Motor` output is bound to the expression `^active`, which denotes the entrance of mode `active`. Thus `Motor` will be present at exactly the time instants when the `active` mode is entered. The controller starts `inactive`. When `ReqOn` becomes present, `controller` transitions to `active`. It remains in `active` until the `ReqOff` input arrives, at which time the controller transitions to `inactive`. The `when` statements specify transitions among modes. Without going into detail, by default, transitions taken become effective one time instant after they are taken. Consider now this code, which extends the `controller` code:

```
mode controller {
  input ReqFwd, ReqBack, ReqOff;
  output Motor = ^active;

  mode inactive { when (ReqFwd | ReqBack) => active; }
  mode active {
    mode stopped {
      when ReqFwd -> fwd
      | ReqBack -> back;
    }
    mode fwd { when ReqStop -> stopped; }
    mode back { when ReqStop -> stopped; }
    when ReqOff -> inactive;
  }
}
```

The modes `stopped`, `up` and `down` are all serial children of `active`, with `stopped` being initial. We have

eliminated the `ReqOn` input and changed the transition expression from the inactive mode to active to account both the new `ReqFwd` or the `ReqBack` input signals. Notice also the new separator `=>` for the when expression, specifying that this transition is to be taken immediately (i.e., within the same time instant).

Besides being serial or parallel, modes can have other attributes, which are introduced to simplify the syntax of complex MacBeth programs as well as enhance the semantics of regular modes in ways that are commonly found in practice.

Terminal Modes

Terminal modes are regular modes that embody a notion of termination: in essence they are sink states, i.e., with no outgoing transitions but possibly with children. Transitions in other parts of the program can check whether the parent mode to a terminal mode has terminated. A terminal mode without children terminates immediately. A terminal mode with children terminates when one of its children terminates. Consider the situation of the controller specified above. Suppose that the controller via some events, can decide to go into an “out of service” mode that is never left, i.e., the controller must be restarted from the outside in order to become functional again:

```

mode controller {
  ...
  mode active {
    when ReqOff -> inactive;
    | ... -> out_of_service;
  }
  terminal out_of_service { ... }
}

```

The purpose of terminal modes is to explicitly denote termination, so that other code outside of the parent mode use it in transition expressions. Modes outside the controller may have transitions based on this state. Typically, this is used in polling situations, where some part of the specification is periodically checking on the status of some other part for termination.

Exceptions and Handler Modes

Another type of mode that helps in complex system specification and programming is the *exception* mode. Exception modes are modes with no children and no outgoing transitions. While terminals are suitable for polling scenarios, exceptions are suitable for on-demand (or interrupt-based) situations, such as error conditions, alerts or interrupt requests:

```

mode controller {
  ...
  mode active {
    when ... -> raise error;
  }
}

```

Exceptions, unlike terminals, must always show in the interface of a mode, unless they are “handled” inside. Exceptions in MacBeth are inspired by exceptions in Standard ML.

Exceptions in MacBeth must always be handled, and that is the purpose of *handler* modes. Handler modes must always be external to where the exceptions are raised.

```

mode system {
  ...
  mode controller { ... }
  handle controller.error { // handler code }
}

```

For all purposes it is as if a direct transition from the exception mode to the handler mode exist. The handler thus must be serial to the mode that it is handling. This syntax allows modes external to where the exception is raised to handle it without knowing the actual internal structure of the mode raising the exception. Similarly, exception raised do not need to know how they are handled. In effect, this helps in abstraction, reduces complexity and helps in code reuse.

3.2 Transitions

The control flow of the system takes place, as we have seen above, via *transitions*. The smallest mode completely containing the source and destination modes of a transition must be a serial mode. Transitions have *transition expressions* associated to them. When the transition expression becomes *triggered*, it may be *taken* and control moves from the source mode to the destination mode of the transition.

Deterministic and Non-Deterministic Transition Expressions

This segment is a deterministic version of the previous code for `stopped`:

```

mode stopped {
  when [ReqFwd; wait START_DELAY] -> fwd;
  when [ReqBack; wait START_DELAY] -> back;
}

```

Before, if `ReqFwd` and `ReqBack` were to be present simultaneously, the destination of the transition would be a non-deterministic choice between `fwd` and `back`, while here, `fwd` takes precedence over `back`.

Timing Transition Expressions

Timing transition expressions in MacBeth boil down to an ordered pair (lb, ub) , where lb is the lower bound or minimum *delay* of the transition and ub is the upper bound or *deadline*. The delay is a non-negative integer and the deadline is an integer equal or greater than the delay or `inf`, which denotes an unbounded deadline. Unless the delay and the deadline are the same, these expressions may introduce non-deterministic behaviors in specifications. Several constructs are build on top of this notation to introduce convenience, expressiveness and *implementation* constraints/hints.

3.3 Processes and Channels

In real systems, behavior is implemented in programs and programs run enclosed in processes, which are units of structuring, scheduling, naming, and containment.

Large applications are partitioned in collections of interacting processes, each assigned to one CPU, scheduled according to its needs. Processes typically provide a namespace within a program and offer a view of memory that contains the actions of the program.

MacBeth has a similar notion of processes. The compilation process maps into the traditional notion of process in real-time operating systems. MacBeth *processes* enclose other processes, which ultimately contain modes as behaviors. Behaviors, as we have seen above, have global, broadcast namespace, where events in one part of the behavioral specification is seen synchronously by all the specification and may trigger other events. Process execution is asynchronous, i.e., processes are assumed to execute independent of one another (modulo inter-process communication/synchronization). The inter-process communication is typically classified in *blocking* and *non-blocking* communication.

Channels and Bindings

MacBeth provides *channels* as the communication element between processes. A channel connects an input *port* to an output port. Processes (as and ultimately modes) have ports in them to perform communication with other processes (and their modes). Channels can be specified to be blocking, non-blocking, lossy, buffered, unidirectional, bidirectional, etc.

```

process table {
  chan { bidirectional } c0, c1, c2, c3;
  process p0 = phil; // four instances of phil
  process p1 = phil; // running asynchronously
  process p2 = phil;
  process p3 = phil;
  // channel bindings in a ring
  bind p0.left -> c3 -> p3.right;
  bind p3.right -> c3 -> p0.left;
  ...
  bind p2.right -> c2 -> p3.left;
}

```

Consider the classic distributed dining philosophers: four philosophers in a ring, each with three states: hungry, eating and thinking. The interface, includes only two ports, `left` and `right`, which are used to send and receive the fork tokens from the neighboring philosophers. We set up some `phil` processes in a cycle, enclosed by a process called `table`. The `table` process does not have behavior of its own, it is only a container of the other processes and the channels that communicate them. We arrange the (static) connections between the ports via the channels (which have been specified to be bidirectional) using `bind` operations.

Tasks and Parallel modes

In MacBeth it is possible to specify that tasks execute with timing constraints, such as calling arbitrary C code with periodically while a mode is active. In the philosopher example, inside hungry mode, instead of getting the right fork first and the the other one second, we could have specified the two modes to be in parallel to pick up the forks independently and then waiting for them after we have both forks.

4 Summary and Future Work

We outlined MacBeth, a language for specification and programming the behavioral specification and programming of protocols and reactive/real-time applications. The aim of the language is to provide a way to do so at a higher level of abstraction, in a general way, yet in a formal, concise and efficient fashion.

MacBeth introduces several new concepts. On top of the concurrency, synchrony, hierarchy and broadcast communication present in existing synchronous languages, MacBeth provides new features, such as asynchronous communication among components, language support for the execution of periodic and event-driven tasks with timing constraints, extended and flexible transition expressions, simplification of basic common constructs specific to real-time systems, encapsulation, modularization, as well as static typing, structuring and parameterization of state diagrams.

References

- [1] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.
- [2] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. Lustre, a declarative language for programming synchronous systems. In *Proceedings of the 14th Symposium on Principles of Programming Languages*, January 1987.
- [3] K. M. Chandy and J. Misra. *Parallel Program Design*. Addison Wesley, 1988.
- [4] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [5] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.
- [6] F. Jahanian and A. Mok. Modechart: a specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20(12):933–947, December 1994.
- [7] J. Katzenelson and R. P. Kurshan. S/R: A language for specifying protocols and other coordinating processes. In *Proc. IEEE Conf. Comput. Comm.*, pages 286–292, 1986.
- [8] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
- [9] C. Puchol. The MacBeth Specification, Modeling and Programming Language. Technical report, Transmeta Corporation, 2000. Available at <http://carlos.puchol.com/pub/>.
- [10] F. Wagner. VFSM executable specification. In *CompEuro92*, 1992.