P: Safe Asynchronous Event-Driven Programming

November 2012

Technical Report
MSR-TR-2012-116

We describe the design and implementation of P, a domain specific language to write asynchronous event driven code. P allows the programmer to specify the system as a collection of interacting state machines, which communicate with each other using events. P unifies modeling and programming into one activity for the programmer. Not only can a P program be compiled into executable code, but it can also be verified using model checking. P allows the programmer to specify the environment, used to "close" the system during model checking, as nondeterministic ghost machines. Ghost machines are *erased* during compilation to executable code; a type system ensures that the erasure is semantics preserving. The P language is carefully designed so that we can check if the systems being designed is responsive, i.e., it is able to handle every event in a timely manner. By default, a machine needs to handle every event that arrives in every state. The default safety checker looks for violations of this rule. Sometimes, handling every event at every state is impractical. The language provides a notion of deferred events where the programmer can annotate when she wants to delay processing an event. The language also provides default liveness checks that an event cannot be potentially deferred forever. Call transitions (which are like subroutines) are used to factor common event handling code, and allow programmers to write complicated state machines. P was used to implement and verify the core of the USB device driver stack that ships with Microsoft Windows 8. The resulting driver is more reliable and performs better than its prior incarnation (which did not use P), and we have more confidence in the robustness of its design due to the language abstractions and verification provided by P.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1. Introduction

Asynchronous systems code that is both performant and correct is hard to write. Engineers typically design asynchronous code using state machine notations, use modeling and verification tools to make sure that they have covered corner cases, and then implement the design in languages like C. They use a variety of performance tricks, as a result of which the structure of the state machines is lost in myriad of details. Clean state machine diagrams that were initially written down become out-of-date with the actual code as it evolves, and the resulting system becomes hard to understand and evolve. During the development of Windows 8, the USB team took a bold step and decided to unify modeling and programming. Various components of the USB driver stack were specified as state machines and asynchronous driver code was auto-generated from these state machines. We were able to use model checking techniques directly on the state machines to find and fix design bugs. Since the executable code was auto-generated from the source, we could make changes at the level of state machines and perform both verification and compilation from one description. This methodology was used to design, validate and generate code for the USB stack that ships with Windows 8. The resulting driver stack is not only more reliable, but also more performant.

In this paper, we formalize and present the salient aspects of this methodology as a domain specific language P. Though P has a visual programming interface, in this paper, we represent P as a textual language with a simple core calculus, so that we can give a full formal treatment of the language, compiler and verification algorithms. A P program is a collection of state machines. Each state machine has a set of states, accepts a set of events, and enumerates the transition that are allowed to happen on each (state,event) pair. For programming convenience, call-transitions are used to factor out common code that needs to be repeated in various situations (similar to nested modes in state charts [9]). Each state has an *entry function*, which contains code to read and update local variables, send events to other state machines, send private events to itself, or call external C functions. The external C functions are used to write parts of the code that have do with data transfer and has no relevance to the control logic.

OS components are required to be responsive. Consequently P programs are required to handle every message that can possibly arrive in every state. Our notion of responsiveness is weaker than synchronous languages like Esterel [3] (which require input events to be handled synchronously during every clock tick, and are hence too strong to be implemented in asynchronous software), but stronger than purely asynchronous languages like Rhapsody [10] (where asynchronous events can be queued arbitrarily long before being handled). Thus, our notion of responsiveness lies in an interesting design point between synchrony and asynchrony. In practice, handling every event at every state would lead to combinatorial explosion in the number of control states, and is hence impractical. The language provides a notion of deferred events to handle such situations and allow a programmer to explicitly specify that it is acceptable to delay processing of certain events in certain states. The language also provides default liveness checks, which ensure that events cannot be deferred for ever.

We also show how to efficiently apply model checking to a P program. The environment can be modeled as set of *ghost* machines, which are used only during modeling and verification and elided during compilation. The type system of P ensures that the ghost machines can be *erased* during compilation without changing the semantics. Using delay bounding scheduling [6] we can efficiently check for large programs that events are properly handled, there is no assertion failure, and that the program is memory safe. Our delay bounded scheduler prioritizes schedules that follow the causal sequence of events, parameterized by a delay bound d. A delay bound of 0 captures all executions that are feasible in a single-threaded runtime, and most bugs that occur in practice are found using low values of the delay bound.

In summary, our contributions are the following:

- We design a DSL P to program asynchronous interacting state machines at a higher level of abstraction, than detailed event handlers that lose the state machine structure.
- We present formal operational semantics,a compiler and runtime that enables P programs to run as KMDF (Kernel Mode Driver Framework) device drivers.
- We show how to efficiently model check P programs and provide a new delay bounding algorithm that explores schedules of the programs by following the causal order of events.
- We report on the use of P in a production environment. Our case study is the USB stack in Windows 8.

2. Overview

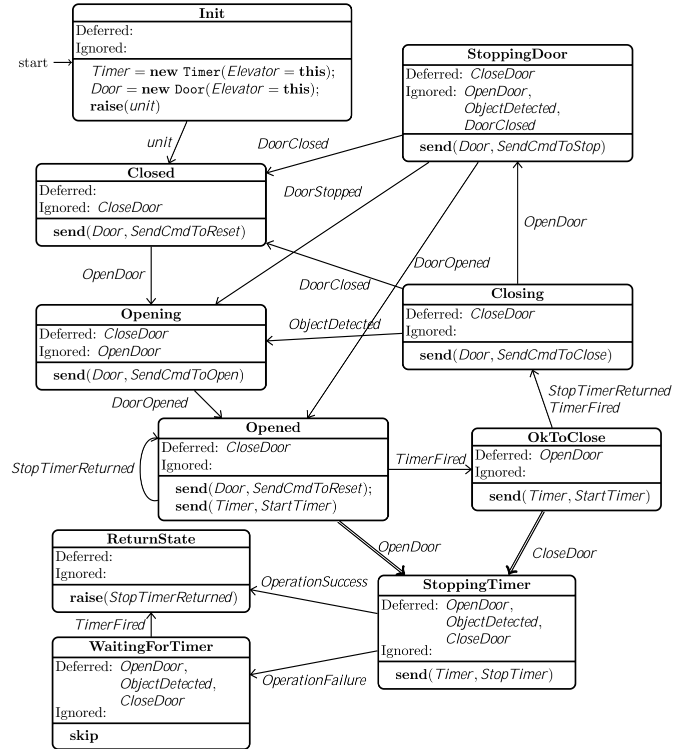
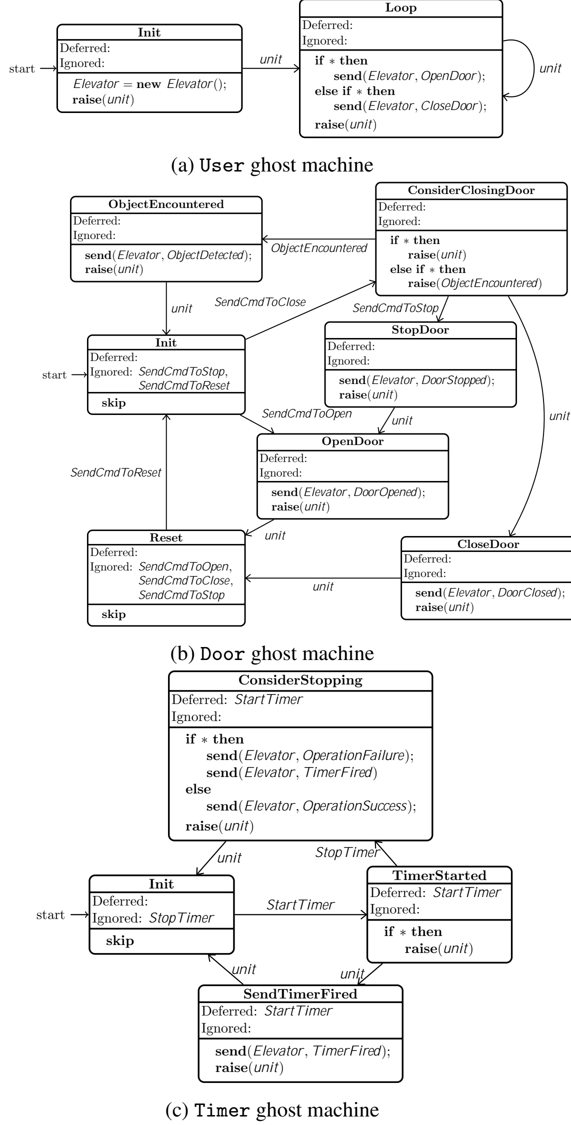


Figure 1: Elevator example

P is a domain-specific language for writing asynchronous event-driven programs. Protocols governing the interaction among concurrently executing components are essential for safe execution of such programs. The P language is designed to clearly explicate these control protocols; to process data and perform other functions irrelevant to control flow, P machines have the capability to call external functions written in C. We call those functions *foreign functions*.

A P program is a collection of *machines*. Machines communicate with each other asynchronously through *events*. Events are queued, but machines are required to handle them in a responsive manner (defined precisely later)—failure to handle events is detected by automatic verification.



(a) `User` ghost machine

(b) `Door` ghost machine

(c) `Timer` ghost machine

Figure 2: Environment for elevator

We illustrate the features of P using the example of an elevator, together with a model of its environment. The elevator machine is shown in Figure 1 and the environment machines in Figure 2. The environment is composed of *ghost machines* which are used only during verification, and elided during compilation and actual execution. Machines that are not ghost are called *real machines*. We use the term *machine* in situations where it is not necessary to distinguish between real and ghost machines. A text version of the example with explanation can be found in Appendix A.

Machines communicate with each other using events. Events are partitioned into two classes: public or private. Public events are sent between different machines and private events are raised within a machine. Each machine is composed of control states, transitions, local events and variables. The elevator machine has private events *unit* and *StopTimerReturned* (which are used for communication locally inside the elevator machine), and two *ghost variables* `Timer` and `Door`. Ghost variables are used only during verification and are used to hold references to ghost machines.

Each state description consists of a 4-tuple (n, d, i, s), where (1) n is a state name, (2) d is a set of events (called *deferred set*), (3) i is a set of events (called *ignored set*), and (4) s is a statement (called *entry statement*), which gets executed when the state is entered. For instance, the **Init** state in Figure 1 has an empty deferred set, empty ignored set, and an entry statement that creates an instance of the `Timer` and `Door` machines and raises the event *unit*. Note that `Timer` and `Door` are ghost machines and their references are stored in ghost variables. As another example, the **Opening** state has $\{CloseDoor\}$ as the deferred set, $\{OpenDoor\}$ as the ignored set, and $\mathbf{send}(Door, SendCmdToOpen)$ as the entry statement. If the state machine enters the **Opening** state, the following things happen: On entry to the state, the statement $\mathbf{send}(Door, SendCmdToOpen)$ is executed, which results in the event *SendCmdToOpen* being sent to the *Door* machine. On finishing the execution of the entry statement, the machine waits for events on the input buffer. The initial state of the `Elevator` machine is **Init**. Whenever an instance of the `Elevator` machine is created (using the $\mathbf{new}$ statement), the machine instance's state is initialized to **Init**.

Deferred and ignored events. Events sent to a machine are stored in a FIFO queue. However, it is possible to influence the order in which the events are delivered. In a given state, some events can be deferred or ignored. When trying to receive an event a machine scans its event queue, starting from the front. Assume that the first event in the queue is e. If e is ignored it is dequeued and the machine looks at the next event in the queue. If e is deferred the machine looks at the next event but e stays in the queue. If e is neither ignored nor deferred then the machine dequeues e and takes the appropriate transition. For instance, in the **Opening** state, the event *CloseDoor* is deferred and *OpenDoor* is ignored.

Step and call transitions. The edges in Figure 1 specify how the state of the `Elevator` machine transitions on events. There are two types of transitions: (1) *step* transitions, and (2) *call* transitions. Both these transition types have the form (n_1, e, n_2), where n_1 is the source state of the transition, e is an event name, and n_2 is the target state of the transition. Step transitions are show by simple edges and call transitions by double edges. For instance, when the machine is in the **Init** state, if an event *unit* event arrives the machine transitions to the **Closed** state. On the other hand, call transitions have the semantics of pushing the new state on the top of the call stack. Call transitions are used to provide a subroutine-like abstraction for machines. For instance, there is a call transition to the **StoppingTimer** state from the **Opened** state on the *OpenDoor* event, and a similar call transition to the **StoppingTimer** state from the **OkToClose** state on the *CloseDoor* event. One can think about the **StoppingTimer** state as the starting point of a subroutine that needs to be executed in both these contexts. This subroutine has 3 states: **StoppingTimer**, **WaitingForTimer** and **ReturnState**. The "return" from the call happens when **ReturnState** raises the *StopTimerReturned* event. This event gets handled by the callers of the subroutine **Opened** and **OkToClose** respectively.

Unhandled events. The P language has been designed to aid the implementation of responsive systems. Responsiveness is understood as follows. If an event e arrives in a state n, and there is no transition defined for e, then the verifier flags an "unhandled event" violation. There are certain circumstances under which the programmer may choose to delay handling of specific events or ignore the events by dropping them. These need to be specified explicitly by marking such events in the associated deferred set or ignored set, so that they are not flagged by the verifier as unhandled. The verifier also implements a liveness check that prevents deferring events indefinitely. This check avoids trivial ways to silence the verifier by making every event deferred in every state.

Environment modeling. Figure 2 shows the environment machines (which are ghost machines) and initialization statement for the elevator. There are 3 ghost machines: `User`, `Door` and `Timer`. These machines are used as environment models during verification, but no code generation is done for these machines. For the purpose of modeling, the entry statements in the states of these machines are allowed to include nondeterminism. For example, the entry statement of the **TimerStarted** state is specified as "$\textbf{if} \ * \ \textbf{then raise}(unit)$". The $*$ expression evaluates nondeterministically to true or false. Thus, when the `Timer` machine enters this state, it can nondeterministically raise the *unit* event. The verifier considers both possibilities and ensures absence of errors in both circumstances. In the real world, the choice between these possibilities depends on environmental factors (such as timing), which we choose to ignore during modeling.

In this example, the initial machine is the `User` machine, and this is the starting point for the model checker to perform verification. Note that the initial state of the `User` machine creates an instance of `Elevator`, and the `Elevator` instance in turn creates instances of `Timer` and `Door` (in Figure 1). During execution, it is the responsibility of some external code to create an instance of the `Elevator` machine.

More examples are found in Appendix B.

3. P Syntax and Semantics

Figure 3 shows the syntax of the core of P. Some of the features presented in the examples of Section 2 can be compiled using a preprocessor into this core language. In particular, state descriptions in the core language are triples of the from (n, d, s), where n is a state name, d is a set of deferred events, and s is an entry statement. Note that the ignored set has been dropped from the syntax shown in the `Elevator` example. The reason is that an ignored set can be implemented using call transitions and self loops as shown later.

A program in the core language consists of declaration of events, a nonempty list of machines, and one machine creation statement. Each event declaration also specifies a list of types, which are types of data arguments that are sent along with the event (can be thought of as "payload" of the event).

A machine declaration consists of (1) a machine name, (2) a list of events, (3) a list of variables, (4) a list of states, (5) a list of function signatures, and (6) a list of transitions. The events declared inside a machine are its private, or local, events (as opposed to the events declared outside all the machines, which are global events). Each variable has a declared type, which can be int, byte, bool, event or machine identifier type (denoted $\mathbf{Id}$). The function signatures are used to link P machines to foreign code. Transitions are either steps or calls.

A machine can optionally be declared as *ghost* by prefixing its declaration by the keyword $\mathbf{ghost}$. Variables can be also declared as $\mathbf{ghost}$. Events sent to ghost machines are (implicitly) ghost events. Ghost machines, events and ghost variables are used only during verification, and are elided during compilation and execution of the P program.

As mentioned in Section 2, a state declaration consists of a name n, a set of events (called deferred set), and a statement. Each state declaration must have a distinct name. Thus, we can use the name n to denote the state. The statement associated with a state n is executed whenever control enters n. Given a machine name m and a state n in m, let $\mathit{Deferred}(m, n)$ denote the associated set of deferred events, and let $\mathit{Entry}(m, n)$ denote the associated statement. The initial state of the machine m is the first state in the state list and is denoted by $\mathit{Init}(m)$.

Each transition declaration comprises a source state, an event, and a target state. The set of transitions of m must be deterministic, that is, if (n, e, n_1) and (n, e, n_2) are two transitions then $n_1 = n_2$. The list of transitions is partitioned into step transitions and call transitions. A call transition is similar to function calls in programming languages and is implemented using a stack (more details below).

program	::=	evdecl machine$^+$ m(init*)
machine	::=	optghost $\mathbf{machine}$ m evdecl vrdecl fctdecl stdecl trdecl
optghost	::=	ϵ | $\mathbf{ghost}$
vrdecl	::=	rvrdecl gvrdecl
trdecl	::=	spdecl cldecl
evdecl	::=	ϵ | $\mathbf{event}$ edecl$^+$
rvrdecl	::=	ϵ | $\mathbf{var}$ vdecl$^+$
gvrdecl	::=	ϵ | $\mathbf{ghost\ var}$ vdecl$^+$
fctdecl	::=	ϵ | $\mathbf{function}$ fdecl$^+$
stdecl	::=	ϵ | $\mathbf{state}$ sdecl$^+$
spdecl	::=	ϵ | $\mathbf{step}$ tdecl$^+$
cldecl	::=	ϵ | $\mathbf{call}$ tdecl$^+$
edecl	::=	e | e(type)
vdecl	::=	x : type
sdecl	::=	$(n, \{e_1, e_2, \ldots, e_k\},$ stmt$)$
tdecl	::=	(n, e, n)
fdecl	::=	ftype f(arg1)
arg1	::=	ϵ | type arg2
arg2	::=	ϵ | , type arg2
type	::=	$\mathbf{int}$ | $\mathbf{byte}$ | $\mathbf{bool}$ | $\mathbf{event}$ | $\mathbf{Id}$
ftype	::=	$\mathbf{void}$ | type
stmt	::=	$\mathbf{skip}$
	|	$x :=$ expr
	|	$x := \mathbf{new}\ m$(init*)
	|	$\mathbf{delete}$
	|	$\mathbf{send}$(expr, e, expr)
	|	$\mathbf{raise}$ (e, expr)
	|	$\mathbf{return}$
	|	$\mathbf{assert}$(expr)
	|	stmt; stmt
	|	$\mathbf{if}$ expr $\mathbf{then}$ stmt $\mathbf{else}$ stmt
	|	$\mathbf{while}$ expr stmt
init	::=	$x =$ expr
expr	::=	$\mathbf{this}$ | $\mathbf{msg}$ | $\mathbf{arg}$
		| c | $\bot$ | x
		| uop expr | expr bop expr
		| $\mathbf{choose}$(type)
		| f(explst1)
explst1	::=	ϵ | expr explst2
explst2	::=	ϵ | , expr explst2
$c \in \mathbf{int}$		$b \in \mathbf{bool}$
$\neg, - \in$ uop		$+, -, \wedge, \vee \in$ bop
$r \in$ expr		$e, m, l, x, f \in$ name

Figure 3: Syntax

The entry statement associated with a state is obtained by composing primitive statements using standard control flow constructs such as sequential composition, conditionals, and loops. Primitive statements are described below. The $\mathbf{skip}$ statement does nothing. The assignment $x := r$ evaluates an expression r and writes the result into x. The statement $x := \mathbf{new}\ m(\text{init}^*)$ creates a new machine and stores the identifier of the created machine into x. The initializers give the initial values of the variables in the created machine. The $\mathbf{delete}$ statement terminates the current machine (which is executing the statement) and release its resources. The statement $\mathbf{send}(r_1, e, r_2)$ sends event e to the target machine identified by evaluating the expression r_1, together with arguments

obtained by evaluating r_2. The event e must be a global event. When e does not have any argument `null` is expected. In the examples, we use $\mathbf{send}(r_1, e)$ as syntactic sugar for $\mathbf{send}(r_1, e, \texttt{null})$. The statement $\mathbf{raise}(e, r)$ terminates the evaluation of the statement raising an event e with arguments obtained by evaluating r. The event e must be a local event. The $\mathbf{return}$ statement terminates the evaluation of the statement and returns to the caller (see below for more details). The statement $\mathbf{assert}(r)$ moves the machine to an error state of the expression r evaluates to false, and behaves like $\mathbf{skip}$ otherwise.

Foreign functions. To interact with external code, also called foreign code, a P program has the ability to call functions written in the C language. These functions needs to be introduced in the scope of a machine by a declaration that give the function's name and type signature. The runtime semantics of a function call to a foreign functions is similar to a standard C method call. For verification purposes we allow the user to optionally give a body to a foreign function. The body has to be erasable, i.e. using only ghost variables. If a foreign function has no body, the function call is replaced by the appropriate nondeterministic $\mathbf{choose}(\mathsf{type})$ expression during verification. Appendix B.2 shows an example using this feature.

Expressions and evaluation. The expressions in the language, in addition to the declared variables, can also refer to three special variables—$\mathbf{this}$, $\mathbf{msg}$ and $\mathbf{arg}$. While $\mathbf{this}$ is a constant containing the identifier of the executing machine, $\mathbf{msg}$ contains the event that is last received from the input buffer of the machine, and $\mathbf{arg}$ contains the payload from the last event. Expressions also include constants c, the special constant $\perp$, variables, and compound expressions constructed from unary and binary operations on primitive expressions. Binary and unary operators evaluate to $\perp$ if any of the operand expressions evaluate to $\perp$. $\perp$ values arise either as a constant, or if an expression reads a variable whose value is uninitialized, and propagate through operators in an expression. The expression $\mathbf{choose}(\mathsf{type})$ nondeterministically evaluates to any of the values of the given type. The expression $*$ is used as syntactic sugar for $\mathbf{choose}(\mathbf{bool})$. Nondeterministic expressions are allowed only in ghost machines and are used to model abstractions of the environment conveniently.

Memory management. P programs manage memory manually by using the $\mathbf{new}$ and $\mathbf{delete}$ commands. The $\mathbf{new}$ command allocates a new instance of a machine and returns its reference, and the $\mathbf{delete}$ command terminates the machine which executes the command and frees its resources. It is the responsibility of the P programmer to perform cleanup and ensure absence of dangling references, or pending message exchanges before calling $\mathbf{delete}$. Manually managing the memory add some complexity in order to retain a precise control over the footprint of the program. Appendix B.1 shows a variant of the elevator example where the environment allocates several instances of the elevator machine and gracefully cleans up these machines in the presence of asynchrony.

3.1 Operational semantics

The role played by the environment is different during execution and verification of a P program. During execution, the environment is responsible for creating initial machines in the P program, sending some initial messages to it, and responding to events sent by the P machines. During verification, the environment is specified using ghost machines, and the program starts execution with a single machine instance of the machine specified by the initialization statement at the end of the program, and this machine begins execution in its initial state with an empty input queue. However, once the initial configuration is specified (which is different during execution and verification), the transition rules are the same for execution as well as verification. We formally specify the transition semantics using a single set of transition rules below.

Since our language allows dynamic creation of machines, a global configuration would contain, in general, a collection of machines. A machine identifier id represents a reference to a dynamically-created machine; we denote by $Name(id)$ the name of the machine with identifier id. A global configuration M is a map from a machine identifier to a tuple representing the machine configuration. A machine configuration is of the form (γ, σ, s, q) with components defined as follows:

- γ is a sequence of pairs (n, d), where n is a state name and d is a set of deferred events. This sequence functions as a call stack, to implement call and return.
- σ is a map from variables declared in the machine to their values; this map contains an entry for the local variables $\mathbf{this}$, $\mathbf{msg}$ and $\mathbf{arg}$.
- s is the statement remaining to be executed in machine id.
- q is a sequence of pairs of a event-argument pairs representing the input buffer of machine id.

We define $Trans(m, n, e)$ to be equal to $(\mathbf{step}, n')$ if there is a step transition labeled e between n and n' in machine m, and $(\mathbf{call}, n)$ if there is a call transition labeled e between n and n' in machine m; otherwise, $Trans(m, n, e) = \perp$, denoting that no transition has been defined for event e at state n in machine m.

Let S be constructed according to the following grammar:

$$S ::= \square \mid S; \mathsf{stmt}$$

The leftmost position in S is a *hole* denoted by $\square$; there is exactly one $\square$ in any derivation for S. We denote by $S[s]$ the substitution of statement $s \in \mathsf{stmt}$ for the unique hole in S. Finally, we have $|q| = \bigcup_{(e,v) \in q} \{e\}$.

The rules in Figure 4 give the operational semantics of our programming language. The program starts execution in a configuration M defined at a single id_0 such that $Name(id_0) = m$, where m is the machine name specified in the program's initialization statement (at the end of the program). and $M[id_0]$ = $((Init(m), \{\}), \lambda x.\ \perp, Entry(m, Init(m)), \epsilon)$. The semantics is defined as a collection of rules for determining transitions of the form $M \longrightarrow M'$. All existing state machines are running concurrently retrieving events from their input queue, performing local computation, and possibly sending events to other machines. Each rule picks an existing machine with identifier id and executes it for a step. To simplify the rule we use small steps ($\longrightarrow$) for statements and big steps ($\Downarrow$) for the expression. The rules for expressions are as expected, thus, omitted.

The rules from ASSIGN to WHILE-DONE execute small steps performed during the computation of the entry function of a state. During this computation, local variables could be modified and events could be sent to other state machines.

The rule SEND shows the semantics of the statement $\mathbf{send}(r_1, e, r_2)$. First, the target of the send $id' = \sigma(r_1)$, and the payload of the event $v = \sigma(r_2)$ are evaluated and the event (e, v) is appended to the queue of the target machine identified by id' using the special append operator $\odot$. The operator $\odot$ is defined as follows. If $(e, v) \notin q$, then $q \odot (e, v) = q \cdot (e, v)$. Otherwise, $q \odot (e, v) = q$. Thus, event-value pairs in event queues are unique, and if the same event-value pair is sent more than once to a machine, only one instance of it is added to the queue, avoiding flooding of the queue due to events generated by hardware, for instance. In some cases, the programmer may want multiple events to be queued, and they can enable this by differentiating the instances of these events using a counter value in the payload.

$$\frac{M[id] = (\gamma, \sigma, S[x := r], q) \quad \sigma(r) \downarrow v}{M \longrightarrow M[id := (\gamma, \sigma[x := v], S[\mathbf{skip}], q)]} \ (\textsc{Assign})$$

$$\frac{\begin{array}{c} M[id] = (\gamma, \sigma, S[x := \mathbf{new}\, m'(x_1 = r_1, x_2 = r_2, \ldots, x_n = r_n)], q) \\ id' = \mathit{fresh}(m') \\ n' = \mathit{Init}(m') \quad \sigma(r_1) \downarrow v_1 \quad \sigma(r_2) \downarrow v_2 \quad \cdots \quad \sigma(r_n) \downarrow v_n \\ \sigma' = \lambda x. \bot\, [\mathbf{this} := id'][x_1 := v_1][x_2 := v_2] \cdots [x_n := v_n] \end{array}}{\begin{array}{c} M \longrightarrow M[id := (\gamma, \sigma[x := id'], S[\mathbf{skip}], q)] \\ [id' := ((n', \{\}), \sigma', \mathit{Entry}(m', n'), \epsilon)] \end{array}} \ (\textsc{New})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{delete}], q)}{M \longrightarrow M[id := \bot]} \ (\textsc{Delete})$$

$$\frac{\begin{array}{c} M[id] = (\gamma, \sigma, S[\mathbf{send}(r_1, e, r_2)], q) \\ \sigma(r_1) \downarrow id' \quad \sigma(r_2) \downarrow v \quad M[id'] = (\gamma', \sigma', C', q') \end{array}}{M \longrightarrow M[id := (\gamma, \sigma, S[\mathbf{skip}], q)][id' := (\gamma', \sigma', C', q' \odot (e, v))]} \ (\textsc{Send})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{raise}\,(e, r)], q) \quad \sigma(r) \downarrow v}{M \longrightarrow M[id := (\gamma, \sigma, \mathbf{raise}\,(e, v), q)]} \ (\textsc{Raise})$$

$$\frac{M[id] = ((n, d) \cdot \gamma, \sigma, S[\mathbf{return}], q)}{M \longrightarrow M[id := (\gamma, \sigma, \mathbf{skip}, q)]} \ (\textsc{Return})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{assert}(r)], q) \quad \sigma(r) \downarrow \mathbf{true}}{M \longrightarrow M[id := (\gamma, \sigma, S[\mathbf{skip}], q)]} \ (\textsc{Assert-Pass})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{skip}; s], q)}{M \longrightarrow M[id := (\gamma, \sigma, S[s], q)]} \ (\textsc{Seq})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2], q) \quad \sigma(r) \downarrow \mathbf{true}}{M \longrightarrow M[id := (\gamma, \sigma, S[s_1], q)]} \ (\textsc{If-Then})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2], q) \quad \sigma(r) \downarrow \mathbf{false}}{M \longrightarrow M[id := (\gamma, \sigma, S[s_2], q)]} \ (\textsc{If-Else})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{while}\ r\ s], q) \quad \sigma(r) \downarrow \mathbf{true}}{M \longrightarrow M[id := (\gamma, \sigma, S[s; \mathbf{while}\ r\ s], q)]} \ (\textsc{While-Iterate})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{while}\ r\ s], q) \quad \sigma(r) \downarrow \mathbf{false}}{M \longrightarrow M[id := (\gamma, \sigma, S[\mathbf{skip}], q)]} \ (\textsc{While-Done})$$

$$\frac{\begin{array}{c} M[id] = ((n, d) \cdot \gamma, \sigma, \mathbf{skip}, q_1 \cdot (e, v) \cdot q_2) \quad m = \mathit{Name}(id) \\ d_1 = \mathit{Deferred}(m, n) \quad t = \{e \mid \mathit{Trans}(m, n, e) \neq \bot\} \\ d' = (d \cup d_1) - t \quad |q_1| \subseteq d' \quad e \notin d' \end{array}}{M \longrightarrow M[id := ((n, d) \cdot \gamma, \sigma, \mathbf{raise}\,(e, v), q_1 \cdot q_2)]} \ (\textsc{Dequeue})$$

$$\frac{\begin{array}{c} M[id] = ((n, d) \cdot \gamma, \sigma, \mathbf{raise}\,(e, v), q) \quad m = \mathit{Name}(id) \\ \mathit{Trans}(m, n, e) = (\mathbf{step}, n'), \quad \sigma' = \sigma[\mathbf{msg} := e][\mathbf{arg} := v] \end{array}}{M \longrightarrow M[id := ((n', d) \cdot \gamma, \sigma', \mathit{Entry}(m, n'), q)]} \ (\textsc{Step})$$

$$\frac{\begin{array}{c} M[id] = ((n, d) \cdot \gamma, \sigma, \mathbf{raise}\,(e, v), q) \\ m = \mathit{Name}(id) \quad \mathit{Trans}(m, n, e) = (\mathbf{call}, n') \\ d' = d \cup \mathit{Deferred}(m, n) \quad \sigma' = \sigma[\mathbf{msg} := e][\mathbf{arg} := v] \end{array}}{M \longrightarrow M[id := ((n', d') \cdot (n, d) \cdot \gamma, \sigma', \mathit{Entry}(m, n'), q)]} \ (\textsc{Call})$$

$$\frac{\begin{array}{c} M[id] = ((n, d) \cdot \gamma, \sigma, \mathbf{raise}\,(e, v), q) \\ m = \mathit{Name}(id) \quad \mathit{Trans}(m, n, e) = \bot \end{array}}{M \longrightarrow M[id := (\gamma, \sigma, \mathbf{raise}\,(e, v), q)]} \ (\textsc{Pop})$$

Figure 4: Operational semantics: correct transitions

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{assert}(r)], q) \quad \sigma(r) \downarrow \mathbf{false}}{M \longrightarrow \mathit{error}} \ (\textsc{Assert-Fail})$$

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{send}(r_1, e, r_2)], q) \quad \sigma(r_1) \downarrow \bot}{M \longrightarrow \mathit{error}} \ (\textsc{Send-Fail1})$$

$$\frac{\begin{array}{c} M[id] = (\gamma, \sigma, S[\mathbf{send}(r_1, e, r_2)], q) \\ \sigma(r_1) \downarrow id' \quad M[id'] = \bot \end{array}}{M \longrightarrow \mathit{error}} \ (\textsc{Send-Fail2})$$

$$\frac{M[id] = (\epsilon, \sigma, \mathbf{raise}\,(e, v), q)}{M \longrightarrow \mathit{error}} \ (\textsc{Pop-Fail})$$

Figure 5: Operational semantics: error transitions

The computation terminates either normally via completion of executing all statements in the entry statement, execution of a **return** statement (which results in popping from the call stack), or by raising an event e. In the first two cases, the machine attempts to remove an event from the input queue via the rule DEQUEUE prior to taking a step based on the retrieved event via the rule STEP. In the third case, the raised event e must be processed immediately via the rules STEP or CALL. If these two rules are not applicable due to the unavailability of a suitable transition, then the topmost state on the machine stack is popped via the rule POP to allow the next state to process the event.

Each state in a state machine can opt to defer a set of events received from the outside world. The logic for dequeuing an event from the input buffer is cognizant of the current set of deferred events and skips over all deferred events from the front of the queue. The deferred set of a stack of state is interpreted to be the union of the deferred set at the top of the call stack with the value resulting from evaluating the deferred set expression declared with that state. In case an event e is both in the deferred set and has a defined transition from a state, the defined transition overrides, and the event e is not deferred (see rule DEQUEUE).

Figure 5 specifies error transitions. The error configuration, denoted by error, can be reached in one of 4 ways: (1) by failing an assertion (rule ASSERT-FAIL), (2) by executing a statement $\mathbf{send}(r_1, e, r_2)$ with r_1 evaluating to $\bot$ (rule SEND-FAIL1), (3) by executing a statement $\mathbf{send}(r_1, e, r_2)$ with r_1 evaluating to some id', but with $M[id'] = \bot$, thereby attempting to send to an uninitialized or deleted machine (rule SEND-FAIL1), and (4) attempting to handle an event from an empty stack (rule POP-FAIL). In Section 5, we show how to detect all these 4 types of errors automatically using model checking.

Responsiveness. Beyond providing constructs for building safe programs, the design of the P language also contains constructs to build responsive programs. Explicitly deferring messages instead of doing so implicitly is such a design choice. However, it is still possible to excessively defer events, thus not processing them. Therefore, we propose a stronger notion of progress. We identify two properties that every real machine m has to satisfy: (1) if m with a non-empty queue is scheduled infinitely often then m eventually dequeues an event; (2) if event e is sent to m and m dequeues infinitely often then m eventually process e. In Section 5 we show how to encode responsiveness as Büchi automata.

The first property checks that no machine can get into a cycle of private events. A machine entering such a cycle will loop forever which leads to a non-terminating program. The second property ensures the absence of excessive deferring. The goal is to prevent events from being always deferred, thus never processed. In some case, it may happen that this property is not respected. For instance, in a system with prioritized events, enough high priority events may delay the lower priority events. Thus, we provide the ability to annotate some events as exempt from (2).

Implementing ignored sets. We briefly describe how ignored sets (as used in the examples from Section 2) can be implemented using **call** transitions, thereby justifying that ignored sets can be omitted from the core language, and implemented using a pre-processor. For every state description of the from (n, d, i, s) where n is a state name, d is a deferred set, i is a non-empty ignored set and s is a statement, we perform the following transformation: (1) Add a new state with name n' with an empty deferred set, and a **skip** statement as entry statement. (2) Add a new event e_ignore. (3) Add a $\mathbf{raise}(e_ignore)$ statement at the end of S. (4) Add a call transition (n, e_ignore, n'). (5) For every event $e \in I$ add the self-loop step transition (n', e, n').

It is easy to argue that this transformation implements ignored sets correctly. When the state n is reached, after executing the entry statement s, the statement `raise e_ignore` forces a call transition to the state n'. Any events in i that are received while at n' are ignored, consistent with the semantics of ignored sets. If an event $e \notin i$ is received while at state n', the state n' is popped (according to rule POP) in the operational semantics, and the event is handled by the transitions from state n.

3.2 Type system and erasure

The type system of P is, on purpose, kept very simple. It mostly does simple checks to make sure the machines, transitions, and statements are well-formed. In particular, the following checks are performed: (1) check that identifiers for machines, state names, events, and variables are unique, (2) check that statements inside real machines are deterministic, (3) ensure that private events are raised and public events are sent, and (4) ensure that ghost machines, ghost variables and ghost events can be *erased* during compilation and execution.

The formal presentation of the type system with typing rules is present in Appendix C. The only non-trivial part of our type system is the rules that deal with the erasure property of ghost variables, and ghost machines. We identify "ghost terms" in statements of real machines, and check that they do not affect the runs of real machines (except for assertions). The separation is needed since ghost terms are kept only for verification purposes and are erased during the compilation. Therefore, only a limited flow of information is allowed between real and ghost terms. For machine identifiers we enforce complete separation, because we need to unambiguously identify the **send** operation that targets ghost machine, so that it can be preserved during verification and erased during compilation.

The error transitions specified in Figure 5 cannot be detected by our type checker. Instead, we use state-space exploration techniques with appropriate bounding techniques (described in Section 5) to check for these errors statically.

4. Execution

This section explains how we generate code from a state machine specified in P, so that the generated code can run as a Windows device driver. This needs a host driver framework and our current implementation uses Windows KMDF (Kernel Mode Driver Framework).

The complete driver, which runs inside Windows, has 4 components: (1) The *generated code* is a C file which is produced by the P compiler from a state machine description. For instance, `elevator.p` is compiled to `elevator.c`. (2) The *runtime* is a library `stateMachine.dll` that interacts with the generated code and provides utilities for synchronization and management of state, execution, and memory. (3) The *foreign code* is a skeletal KMDF driver which mediates between the OS and the generated code by creating instances of P machines and getting the execution started, and translating OS callbacks into P state machine events that are queued into the queues of the respective machines. (4) The *foreign functions* are provided as C source files or libraries. The function calls occurring in the machines are linked to those files.

Generated code. When P is compiled to C, the state of a machine is wrapped into an object of type `StateMachineContext` defined as in Figure 6. The pointer `locals` is downcast to an object of appropriate type in the code generated by the compiler. The `MemoryBlob` is downcast, interpreted and managed by the foreign code and functions. The `EventQueue` is a linked list of items of type `Event`. Each instance of `Event` has a `name` and `value`. The remaining components of the `StateMachineContext` are explained in the comments in Figure 6.

```
1  struct StateMachineContext
2  {
3    void * locals;     //local variables
4    void * MemoryBlob; //for use by foreign code
5    Queue EventQueue ; //event queue
6    Event PrivateEvent;//slot for internal event
7    Stack CallStack;   //state stack
8    Bool IsRunning;    //is state machine running
9    WdfSpinLock lock;  //For exclusive access to context
10 }
11 struct Event
12 {
13   String name;
14   void * value;
15 }
```

Figure 6: StateMachineContext and Event types

P statement	Runtime API
`raise(e,v)`	`SMAddPrivateEvent(smc,e,v)`
`return`	`SMReturn(smc)`
`send (m,e,v)`	`SMAddEvent(m,e,v)`
`new m(x,...)`	`SMCreateMachine(...)`
`delete`	`SMDeleteMachine(m)`

Figure 7: Mapping from DSL primitives to runtime API

The P compiler generates code for the following methods that that are called by the runtime. The method `GNExistsTransition(curState,e)` returns true if the state `curState` has a transition corresponding to event `e`, and false otherwise. The methods `GNGetTargetStateOfTransition(curState,e)` and `GNIsCallTransition(curState,e)` can be called only if `GNExistsTransition(curState,e)` has returned true. The method `GNGetTargetStateOfTransition(curState, e)` returns the new state obtained by executing the transition, and the method `GNIsCallTransition( curState, e)` returns true if the transition in question is a call transition and false otherwise. The method `GNExecuteEntryFunction( curState, locals)` executes the entry function of `curState` reading and possibly mutating the state of `locals`. The compiler generates code for the body of the entry function using the operational semantics given in Section 3. The table in Figure 7 shows the mapping between the primitives in the P code and the methods used to implement these primitives in the runtime.

Runtime. Windows drivers are parsimonious with threads. Worker threads are rarely created and drivers typically use calling threads to do all the work. Thus, when the OS calls the driver, either due to an application request or due to a hardware interrupt or DPC (deferred procedure call), the driver uses the calling thread to process the event and run to completion. Every driver has one or more device objects, and all the state of the driver is stored in the heap attached to the device object (since the driver does not have a thread, and hence no call stack on its own that is preserved across callback invocations from the OS).

The runtime supports the following API methods described below. The method `SMCreateStateMachine` creates a state machine and returns a unique identifier for it. The method `SMAddEvent` adds an external event into the state machine's queue, and also executes the state machine by calling the internal method `SMRunStateMachine` if the machine is blocked, waiting for an external event. The method `SMAddPrivateEvent` adds an internal event to the state machine. This method is called only from the generated code. The method `SMRunStateMachine` looks for

a suitable event in the event queue and executes the transition on the event, and if successful, executes the entry function associated with the new state. The process of running machines continues recursively until all the consequences of adding the event have been computed. When receiving an event priority is given to the private events stored in `smc->PrivateEvent` over events stored in `smc->EventQueue`. The runtime is also responsible for updating the call stack and the set of deferred events when executing a machine. More details about the implementation of the runtime API and the handling of transitions and events by the internal methods of the runtime can be found in Appendix D.

Foreign code. The foreign code is used to mediate between the OS and the P code. It is written as a skeletal KMDF driver, which handles callbacks from the Windows OS and translates them into events it adds to the queue of the P machine, using the runtime API.

In KMDF, the `EvtAddDevice` callback is used to create the state machine using the `SMCreateMachine` API. All events such as Plug and Play or Power management or other events are handled by the foreign code by queuing a corresponding event using the `SMAddEvent` API. The `EvtRemoveDevice` callback results in a special event `Delete` added to the P driver. Every P state machine is required to handle this event by cleaning up and executing the `delete` statement. Note that the P machine may have to do internal bookkeeping and keep track of other machines it has created, and the state of the interactions it has with other machines, cleanup the state of the interactions, and only then execute the `delete` statement.

Foreign functions. The foreign functions are provided by the programmer to complement the P machines. The foreign functions must have one additional argument on top of the ones declared in P. This argument, of type `void *`, points to a zone of memory that can be used by the programmer to persist some information as part of the state of calling machine, see Figure 6. The foreign functions are assumed to terminate and to limit any side effect to the provided memory. The foreign functions are usually coupled with the foreign code. Whereas the foreign code contains part which are generic enough so that they can be fully automated. the foreign functions give the ability to handle more specific interactions with the environment.

In order to make the P compiler work for other driver frameworks such as NDIS or even other systems than Windows OS, the runtime and foreign code needs to be reimplemented appropriately, but the generated code does not need to change.

Efficiency of generated code and runtime. In order to evaluate the efficiency of the code generated by P and the runtime, we performed the following experiment. We developed two drivers for a simple switch-and-led device, one using P, and one directly using KMDF. Both drivers use the same level of asynchrony. The P code is about 150 lines (see Appendix B.2). The driver machine has 15 states and 23 transitions, and there are 4 ghost machines each with 3–4 states and transitions. The foreign code is 1720 lines, written directly in C, interfacing between KMDF and the P code. In contrast, the full KMDF driver (written without using P) is about 6000 lines of C code.

We tested both drivers in an environment which sends 100 events per second, and both drivers are able process each event with an average processing time of 4ms, demonstrating that the P compiler and runtime do not introduce additional overhead. We present a more substantial case study in Section 6.

5. Verification

P is designed so that it is easy to verify P programs using model checking. Ghost machines, which are used to model the environment, generate a "closed" program. The nondeterministic statements present in the ghost machines model a range of behaviors of the environment. The model checker systematically explores every possible outcome of this nondeterminism, and checks for the possible errors (see Figure 5) namely, (1) assertion failures, (2) executing send commands with uninitialized target identifiers, (3) sending events to machine that has been already freed, and (4) unhandled events. On top of the safety properties described above, we also check that P programs are responsive.

Partial order reduction. When exploring the state-space of concurrent programs the number of interleavings that needs to be explored is a critical factor in the state-explosion problem. Since the machines communicate with each other only through events, the model checker explores only interleavings where context-switches occur during sends of events, and machine creation. If an error can occur in an execution with a more fine-grained context-switching, it can be shown to occur in another equivalent execution in which context-switches happen only at the points mentioned above. This reduction of the search space is a form of partial order reduction [5].

Delay bounding. Unfortunately, the coarse-grained context-switching described above is not sufficient to prevent the state space explosion problem. Assuming that there are k machines enabled at each interleaving point, the number of possible schedules for runs of with n context switched is k^n. In addition, since there are no restrictions on the length of the message buffers, it is possible to write P programs that have an infinite number of states. Consequently, we have designed a *delay bounded* scheduler to model check P programs.

Intuitively, the delay bounded scheduler explores schedules that follow the causal sequence of events. Diverging from that sequence is done by delaying some machine. Given a bound d, the scheduler may introduce at most d delays. Suppose machine m_1 sends an event e to machine m_2's input buffer. Then, at a later point m_2 removes e from its input buffer, and processes this message, thereby resulting in an event e' sent to machine m_3's input buffer. The delay bounded scheduler follows the causal sequence of steps which consists of machine m_1 sending the event e to m_2, machine m_2 processing the event e and sending the event e' to m_3, and machine m_3 handling the event e'. A delay that the scheduler may choose to introduce is, for instance, at the second context-switch delaying m_3 and executing m_2.

More formally, the delay bounded scheduler maintains a stack S of P machine identifiers, and an integer delay score. Initially, the stack S contains a single machine identifier corresponding to the instance created by the initialization statement of the P program, and the delay score is set to 0. For example, in the `Elevator` example from Section 2, the stack initially contains the id of the `User` ghost machine.

Let m be the machine id at the top of S. m executes until it reaches a scheduling point (which is a send or the creation of a machine). At the scheduling point the scheduler decides to either follow the event-causal schedule or delay m. In either case, it updates S and the delay score according to the following rules:

- If m is scheduled and m sends an event to machine m', and $m' \notin S$, it pushes m' on S. Otherwise, if $m' \in S$ then S is left unchanged. The delay score is left unchanged.
- If m is scheduled and m creates a new machine m', then it pushes m' on S. The delay score is left unchanged.
- If m is delayed, moves m from top of S to the bottom of S, and increments the delay score by 1.

Given a delay bound d, a delay bounded scheduler explores only those schedules which have a delay score lesser than or equal to d. The scheduler is encoded as a part of the model checker's input.

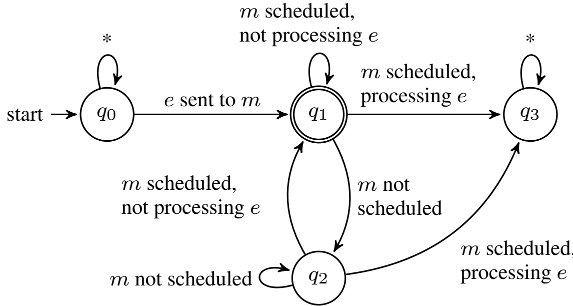


Figure 8: Büchi automaton for responsiveness condition (2)

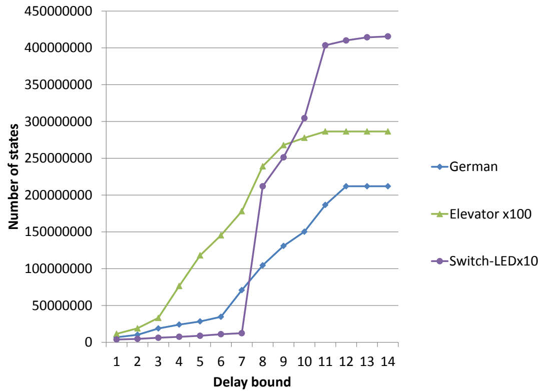


Figure 9: States explored with increasing delay bound.

We can show that for $d = 0$, the real part of schedules explored by the delay bounded scheduler are exactly the same as the one executed by the P runtime in Section 4, assuming no multithreading (that is, at most one thread calls into the P runtime from the kernel). Differences between the runtime and the delay bounded scheduler occur only in the interaction with the ghost machines/environment. Increasing the delay bound d let the scheduler explores more schedules and captures more interactions with the environment. We can also show that as d approaches infinity, in the limit, the delay bounded scheduler explores all possible schedules of a P program, and in particular includes all cases where the P runtime is invoked by an arbitrary number of parallel threads from the kernel. However, even for low values of d, the delay bounded scheduler is very useful in error detection, as shown below.

Responsiveness. We also check the responsiveness properties using model checking. Responsiveness as defined in Section 3 is a liveness property that can be captured by Büchi automata. Figure 8 presents such an automaton for the second responsiveness condition. This automaton has two parameters: a machine m and an event e. It detects cycles where m is not making progress w.r.t. e, i.e. an infinite trace where m takes steps but never processes e.

Whereas the delay bounded scheduler can be used to check safety properties without modification of the model checker, checking liveness properties such as responsiveness is not as simple. The paths described by liveness properties are infinite paths. Embedding theses paths in a finite state-space creates lassos, composed of a finite prefix (the stem) and a cycle. The model checker needs to carefully handle the delays in order to preserve the cycles.

Empirical results. In order to evaluate the efficacy of delay bounding, we conducted the following experiments. For three benchmark P programs (elevator example from Section 2, driver for Switch-and-LED device, and for a software implementation of German's cache coherence protocol), we studied the behavior of delay bounding for varying values of the delay bound parameter d. Figure 9 shows how the number of explored states varies as we increase the value of the delay bound parameter. We scaled the number of states in the Switch-LED by a factor of 10 and Elevator by a factor of 100 to make the graphs legible. We also experimented with buggy versions of these designs and determined that bugs are found within a delay bound of 2. The data can be summarized as follows: bugs are found for low values of delay bound (note the low value of number of states explored for delay bound of 2 in Figure 9 within which bugs were found), and as we increase the delay bound we eventually explore all states within a delay bound of around 12.

6. Case Study

USB Hub Driver: Context and challenges. The state machine methodology described in this paper, together with code generation as well as verification, was used in the development of core components of the USB 3.0 stack that was released as part of Microsoft Windows 8. In particular, the USB hub driver ("USBHUB3.sys") was developed using our methodology. The USB hub driver is responsible for managing USB hubs, the ports in these hubs, enumerating the devices and other hubs connected to their downstream ports. It receives a large number of un-coordinated events sent from different sources such as OS, hardware and other drivers, in tricky situations when the system is suspending or powering down. It can receive unexpected events from disabled or stopped devices, non-compliant hardware and buggy drivers. The hub driver can fail requests from incorrect hardware or buggy function drivers. However, it is important that the USB hub itself handles all events and does not crash or hang itself.

State machine	P states	P transitions	Explored states (millions)	Time (hh:mm)	Memory MB
HSM	196	361	5.9	2:30	1712
PSM 3.0	295	752	1.5	3:30	1341
PSM 2.0	457	1386	2.2	5:30	872
DSM	1919	4238	1.2	5:30	1127

Figure 10: State machine sizes and exploration time

Experience using P in USB Hub. We designed the USB Hub in P as a collection of state machines. The hub, each of the ports, and each of the devices are designed as P machines. Using P helped us serialize the large number of uncoordinated events coming in from hardware, operating system, function drivers and other driver components. In order to make hub scalable, the processing in the hub should be as asynchronous as possible. We captured all the complexity of asynchronous processing using P state machines, and fine-grained and explicit states for each step in the processing. We used sub-state machines to factor common event handling code, and control the explosion in the number of states in the P code, and deferred events to delay processing of low-priority events. We made sure that any code in the driver that lives outside the P state machine (as foreign functions) is relatively simple and primarily does only data processing, and no control logic. This helped us ensure that the most complex pieces of our driver are verified using the state exploration tools of P.

We had to carefully constrain the environment machines in several cases to help direct the verification tools. Even with such

constraints, the actual state spaces explored by the verifier were on the order of several millions, and the verifier runs took several hours to finish (even after using multicores to scale the state exploration), once our designs became mature and the shallow bugs were found and fixed. Figure 10 shows the size and scale of state spaces for the various state machines. The second and third columns show the number of states and transitions at the level of P. The fourth column shows the number of states explored by the explored (taking into account values of variables, state of queues, and state of ghost machines modeling the environment). The fifth and sixth column give the time and space needed to complete the exploration.

The systematic verification effort enabled by P helped us greatly flesh out corner cases in our design, forced us to handle every event (or explicitly defer it) in every state, and greatly contributed to the robustness of the shipped product. Overall, state exploration tools helped us identify and fix over 300 bugs, and justified their continued use throughout the development cycle. A majority of the bugs were due to unhandled events that we did not anticipate arriving. Other bugs were due to unexpected interactions between machines, or with the environment, which manifested in either unhandled messages or assertion violations.

Comparison with existing USB stack. The old USB driver in Windows has existed for several years and predates P. We compare the old USB driver and new driver in terms of functionality, performance, and reliability.

1. Functionality. The new USB hub driver has to deal with new USB 3.0 hardware in addition to all the requirements for the old driver. Therefore the new USB hub driver implements functionality that is a super set of the functionality of the old hub driver.
2. Reliability. The old USB hub driver had significantly more synchronization issues in PnP, power and error recovery paths even till date. The number of such issues has dropped dramatically in the new USB hub driver. The number of crashes in the new USB hub driver due to invalid memory accesses and race conditions is insignificant.
3. Performance. The new USB hub driver performs much better than the old USB hub driver —average enumeration time for a USB device is 30% faster. We have not seen any instances of worker item starvation that we used to see with the old hub driver.

This gain in performance is mainly due to the highly asynchronous nature of the new hub driver. In comparison, the old hub driver blocks processing of worker items in several situations, leading to performance degradation. It is theoretically possible to develop a driver that is not based on explicit state machines such as in P, but is equally (or more) performant. However, in practice, when we have tried to build such asynchronous drivers directly, we have run into myriad of synchronization issues and unacceptable degradation in reliability. The support for asynchrony in P in terms of explicitly documented states and transitions, and the verification tools in P that systematically identified corner cases due to asynchrony were the key reasons why we were able to design the new USB hub driver with both high performance and high reliability.

We note that the new USB hub driver has only been released to the public for about a month at the time of this writing. Once we get more empirical data from usage of USB by Windows 8 users, we can make a more thorough comparison on actual number of crashes and hangs with the old driver.

7. Related work

Synchronous languages. Synchronous languages such as Esterel [3], Lustre [8] and Signal [2] have been used to model, and generate code for real-time and embedded systems for several decades. All these languages follow the synchrony hypothesis, where time advances in steps, and concurrency is deterministic —that is, given a state and an input at the current time step, there is a unique possible state at the next time step. Lustre and Signal follow a declarative dataflow model. Every variable or expression in Lustre represents a *flow* which is a sequence of values. For a flow x, Lustre uses $pre(x)$ do denote a flow with values postponed by one time step. A Lustre program [8] is a set of definitions of flows, where each flow is defined using some constant flows or other flows. Even though flows can be recursively defined, each recursive cycle should be broken using the pre operator. In contrast, Esterel is an imperative language [3] where a program consists of a collection of nested concurrently running threads, and each step is triggered by an external event, and threads are scheduled until all internally generated events are consumed. The Esterel compiler ensures a property called *constructive causality*, which guarantees absence of cyclical dependencies in propagating events, and ensures that each step terminates. Harel's StateCharts [9] is a visual language, with hierarchical states, broadcast communication of events and a synchronous fixpoint semantics which involves executing a series of micro-steps within each time step until all internally generated events are consumed.

The synchronous model has the advantage that every event sent to machine is handled in the next clock tick, and is widely used in hardware and embedded systems. However, in an OS or a distributed system, it is impossible to have all the components of the system clocked using a global clock, and hence asynchronous models are used for these systems. In such models events are queued, and hence can be delayed arbitrarily before being handled. However, arbitrary delays are unacceptable in OS components such as device drivers, which require responsiveness in event handling. The main focus of our work is an asynchronous model where responsiveness is enforced using verification, with the ability do code generation.

Asynchronous languages. Asynchronous languages are used to model and program software systems. The adaptation of StateCharts in the Rhapsody tool has produced a variant, which is suitable for modeling asynchronous software systems. This variant (see [10]) allows steps that take non-zero time and resembles a collection of communicating and interacting state machines, where each machine has a named input queue, and each transition of a machine consumes a message from its input queue and possibly sends messages to the output queues of one or more machines. Other asynchronous models include the actor model [11] and process calculi, such as CSP [12] and CCS [13], and Join Calculus [7], which have asynchronous processes communicating with each other via messages. While these models are useful in modeling and reasoning about asynchronous systems, our goal is to unify modeling and verification with programming, and generate code that can run in an OS kernel.

Domain-specific languages. The Teapot [4] programming language shares similar goals to our work in that they attempt to unify modeling, programming and verification, although in a different application domain —cache coherence protocols. Teapot's continuation passing design is related to the design of P's call transitions. The notion of deferred sets, ghost machines and erasure property, default safety and liveness checks, delay bounding, and the ability of the P compiler and runtime to generate code that runs in an OS kernel are all unique to P.

Automatic stack management. There have been attempts to provide automatic stack management for event-driven programming to allow the possibility of blocking constructs inside procedure calls (e.g., [1]). In P, entry functions of states are written in non-blocking style and call transitions are provided to conveniently factor common event handling code. Thus, stack management is particularly simple in our current design.

8. Conclusion

We presented P, a domain specific language for writing asynchronous event-driven programs. We have given a full formal treatment of various aspects of P, including operational semantics, type system, and verifier. We also presented experience using P to program the USB stack that ships with Windows 8. The main technical contribution of our work is an asynchronous model which forces each event in the queue to be handled as soon as the machine associated with the queue is scheduled, and has a chance to dequeue the event. Our verifier systematically explores the state space of machines and ensures that there are no unhandled events. In certain circumstances, such as processing a high priority event, or processing a sequence of event exchanges during a transaction, some other lower priority events may have to be queued temporarily. P has features such as deferred events for a programmer to explicitly specify such deferrals. Thus, our main contribution is the design of an asynchronous language, which promotes a discipline of programming where deferrals need to be declared explicitly, and consequently leads to responsive systems.

References

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In C. S. Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 289–302. USENIX, 2002.

[2] A. Benveniste, P. L. Guernic, and C. Jacquemot. Synchronous programming with events and relations: the Signal language and its semantics. *Sci. Comput. Program.*, 16(2):103–149, 1991.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Sci. Comput. Program.*, 19(2):87–152, Nov. 1992.

[4] S. Chandra, B. Richards, and J. R. Larus. Teapot: A domain-specific language for writing cache coherence protocols. *IEEE Trans. Software Eng.*, 25(3):317–333, 1999.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.

[6] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In T. Ball and M. Sagiv, editors, *POPL*, pages 411–422. ACM, 2011.

[7] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In H.-J. Boehm and G. L. S. Jr., editors, *POPL*, pages 372–385. ACM Press, 1996.

[8] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[9] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[10] D. Harel and H. Kugler. The Rhapsody semantics of Statecharts (or, on the executable core of the UML) - preliminary version. In H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors, *SoftSpez Final Report*, volume 3147 of *Lecture Notes in Computer Science*, pages 325–354. Springer, 2004.

[11] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[12] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[13] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

```
1  //global event declarations
2  event OpenDoor, CloseDoor, DoorOpened, DoorClosed,
3        ObjectDetected, DoorStopped, TimerFired,
4        OperationSuccess, OperationFailure, SendCmdToOpen,
5        SendCmdToClose, SendCmdToStop, SendCmdToReset,
6        StartTimer, StopTimer
7
8  machine Elevator
9   event unit, StopTimerReturned //private event
10  ghost var Timer:Id, Door:Id // ghost machine references
11  state (Init, {}, {},  Timer = new Timer(Elevator=this);
12                        Door  = new Door(Elevator=this);
13                        raise(unit))
14        (Closed,  {}, {CloseDoor},
15                  send(Door, SendCmdToReset))
16        (Opening, {CloseDoor}, {OpenDoor},
17                  send(Door, SendCmdToOpen))
18        (Opened,  {CloseDoor}, {},
19                  send(Door, SendCmdToReset);
20                  send(Timer, StartTimer))
21        (OkToClose, {OpenDoor}, {}
22                  send(Timer, StartTimer))
23        (Closing, {CloseDoor}, {}
24                  send(Door, SendCmdToClose))
25        (StoppingDoor, {CloseDoor},
26                  {OpenDoor, ObjectDetected, DoorClosed}
27                  send(Door, SendCmdToStop))
28        (StoppingTimer,{OpenDoor, CloseDoor, ObjectDetected},
29                  {}, send(Timer, StopTimer))
30        (WaitingForTimer, {OpenDoor, CloseDoor, ObjectDetected},
31                  {}, skip))
32        (ReturnState, {}, {}, raise(StopTimerReturned))
33
34  step  (Init,           unit,      Closed)
35        (Closed,         OpenDoor,   Opening)
36        (Opening,        DoorOpened, Opened)
37        (Opened,         TimerFired, OkToClose)
38        (Opened,         StopTimerReturned, Opened)
39        (OkToClose,      StopTimerReturned, Closing)
40        (OkToClose,      TimerFired, Closing)
41        (Closing,        OpenDoor,   StoppingDoor)
42        (Closing,        DoorClosed, Closed)
43        (Closing,        ObjectDetected, Opening)
44        (StoppingDoor,   DoorOpened, Opened)
45        (StoppingDoor,   DoorClosed, Closed)
46        (StoppingDoor,   DoorStopped, Opening)
47        (StoppingTimer,  OperationSuccess, ReturnState)
48        (StoppingTimer,  OperationFailure, WaitingForTimer)
49        (WaitingForTimer, TimerFired, ReturnState)
50
51  call  (Opened,         OpenDoor,   StoppingTimer)
52        (OkToClose,      CloseDoor,  StoppingTimer)
```

Figure 11: Elevator example

```
1  ghost machine User
2   var Elevator : Id
3   event unit
4   state
5       (Init, {}, {}, Elevator = new Elevator(); raise(unit) )
6       (Loop, {}, {}, if(*) send(Elevator, OpenDoor)
7                  else if (*) send(Elevator, CloseDoor);
8                  raise(unit) )
9   step
10      (Init, unit, Loop)
11      (Loop, unit, Loop)
12
13 ghost machine Door
14   var Elevator : Id
15   event unit, ObjectEncountered
16   state
17    (Init, {}, {SendCmdToStop, SendCmdToReset}, skip)
18    (OpenDoor, {}, {}, send(Elevator, DoorOpened); raise(unit) )
19    (ConsiderClosingDoor, {}, {},
20                 if(*) raise(unit)
21                 else if(*) raise(ObjectEncountered) )
22    (ObjectEncountered, {}, {},
23                 send(Elevator, ObjectDetected); raise(unit) )
24    (CloseDoor, {}, {}, send(Elevator, DoorClosed); raise(unit) )
25    (StopDoor,  {}, {}, send(Elevator, DoorStopped); raise(unit) )
26    (Reset, {}, {SendCmdToOpen, SendCmdToClose,
27                 SendCmdToStop}, skip)
28   step
29    (Init,       SendCmdToOpen,      OpenDoor)
30    (OpenDoor,   unit,               Reset)
31    (Init,       SendCmdToClose,     ConsiderClosingDoor)
32    (ConsiderClosingDoor, unit,      CloseDoor)
33    (ConsiderClosingDoor, ObjectEncountered, ObjectEncountered)
34    (ConsiderClosingDoor, SendCmdToStop, StopDoor)
35    (CloseDoor,  unit,               Reset)
36    (StopDoor,   unit,               OpenDoor)
37    (ObjectEncountered, unit,        Init)
38    (Reset,      SendCmdToReset,     Init)
39
40 ghost machine Timer
41   var Elevator : Id
42   state
43    (Init, {}, {StopTimer}, skip)
44    (TimerStarted, {StartTimer}, {}, if(*) raise(unit) )
45    (SendTimerFired, {StartTimer}, {},
46                     send(Elevator, TimerFired); raise(unit) )
47    (ConsiderStopping,{StartTimer}, {},
48                     if(*) {send(Elevator, OperationFailure);
49                            send(Elevator, TimerFired)}
50                     else  send(Elevator, OperationSuccess);
51                     raise(unit) )
52   step
53    (Init,            StartTimer,   TimerStarted)
54    (TimerStarted,    unit,         SendTimerFired)
55    (SendTimerFired,  unit,         Init)
56    (TimerStarted,    StopTimer,    ConsiderStopping)
57    (ConsiderStopping, unit,        Init)
58
59 //Initialization statement.
60 User()
```

Figure 12: Environment for elevator

A. Text version of the Elevator example

We illustrate the features of P using the example of an elevator, together with a model of its environment. The elevator machine is shown in Figure 1 and the environment machines in Figure 2. The environment is composed of *Ghost machines* which are used only during verification, and elided during compilation and actual execution. Machines that are not ghost are called *real machines*. We use the term *machine* in situations where it is not necessary to distinguish between real and ghost machines.

Machines communicate with each other using events. Lines 1–6 in Figure 1 are used to declare events with global scope, which can be used across all machines.

Lines 8–52 contain the description of the elevator machine and contains declarations of local events and variables, states, and transitions. The elevator machine has private events `unit` and `StopTimerReturned` declared in line 9 (which are used for communication locally inside the elevator machine), and two *ghost variables* `Timer` and `Door` declared in line 10. Ghost variables are

used only during verification and are used to hold references to ghost machines.

Lines 11–32 describe the states of the elevator machine. Each state description consists of a triple (n, d, i, s), where (1) n is a state name, (2) d is a set of events (called *deferred set*), (3) i is a set of events (called *ignored set*), and (4) s is a statement (called *entry statement*), which gets executed when the state is entered. For instance, lines 11–13 describe the state `Init` with an empty deferred set, empty ignored set, and the entry statement which creates an instance of the `Timer` and `Door` machines and raises the event `unit`. Note that `Timer` and `Door` are ghost machines and their references are stored in ghost variables. As another example, lines 16–17 describe the state `Opening`, with $\{$`CloseDoor`$\}$ as the deferred set, $\{$`OpenDoor`$\}$ as the ignored set, and `send(Timer, StartTimer)` as the entry statement. If the state machine enters the `Opening` state, the following things happen: On entry to the state, the statement `send(Timer, StartTimer)` is executed, which results in the event `StartTimer` being sent to the `Timer` machine. On finishing the execution of the entry statement, the machine waits for events on the input buffer. In this example the `send` method takes two arguments since the event `StartTimer` does not have any payload.

The initial state of the `Elevator` machine is `Init` which is the first state specified in the list of states. Whenever an instance of the `Elevator` machine is created (using the `new` statement), the machine instance's state is initialized to `Init`.

Deferred and ignored events. Events sent to a machine are stored in a FIFO queue. However, it is possible to influence the order in which the events are delivered. In a given state, some events can be deferred or ignored. When trying to receive an event a machine scans its event queue, starting from the front. Assume that the first event in the queue is e. If e is ignored it is dequeued and the machine looks at the next event in the queue. If e is deferred the machine looks at the next event but e stays in the queue. If e is neither ignored nor deferred then the machine dequeue e and takes the appropriate transition. For instance, in the `Opening` state (line 16), the event `CloseDoor` is deferred and occurrences of `OpenDoor` are ignored.

Step and call transitions. Lines 34–52 specify how the state of the `Elevator` machine transitions on events. There are two types of transitions: (1) *step* transitions, and (2) *call* transitions. Both these transition types have the form (n_1, e, n_2), where n_1 is the source state of the transition, e is an event name, and n_2 is the target state of the transition. For instance, line 34 states that when the machine is in the the `Init` state, if an event `unit` event arrives the machine transitions to the `Closed` state.

Call transitions have similar syntax, but more complicated semantics —one of pushing the new state on the top of the call stack. Call transitions are used to provide a subroutine-like abstraction for machines.

For instance, line 51 specifies a call transition to the `StoppingTimer` state from the `Opened` state on the `OpenDoor` event, and line 52 specifies a call transition to the `StoppingTimer` state from the `OkToClose` state on the `CloseDoor` event. One can think about the `StoppingTimer` state as the starting point of a subroutine that needs to be executed in both these contexts. This subroutine has 3 states: `StoppingTimer`, `WaitingForTimer` and `ReturnState`. The states are described in lines 28–29, and transitions between the states are described in lines 47–49. The "return" from the call happens the `ReturnState` raises the `StopTimerReturned` event (see line 32). This event gets handled by the callers of the subroutine in lines 38 and 39 respectively.

Unhandled events. The P language has been designed to help program responsive systems, which handle events in a *responsive* manner. Responsiveness is understood as follows. If an event e arrives in a state n, and there is no transition defined for e, then the verifier flags an "unhandled event" violation. There are certain circumstances under which the programmer may choose to delay handling of specific events or ignore the events by simply dropping them. These need to be specified explicitly by marking such events in the deferred set or ignored set associated with the state, so that they are not flagged by the verifier as unhandled. The verifier also implements a liveness check that prevents deferring events indefinitely. This check avoids trivial ways to silence the verifier by making every event deferred in every state.

Environment modeling. Figure 2 shows the environment machines (which are ghost machines) and initialization statement for the elevator. There are 3 ghost machines: `User`, `Door` and `Timer`. These machines are used as environment models during verification, but no code generation is done for these machines. For the purpose of modeling, the entry statements in the states of these machines are allowed to include nondeterminism. For example, in line 44, the entry statement of the `TimerStarted` state is specified as `if(*) raise(unit)`. The $*$ expression evaluates nondeterministically to true or false. Thus, when the `Timer` machine enters this state, it can nondeterministically raise the unit event. The verifier considers both possibilities and ensures absence of errors in both circumstances. In the real world, the choice between these possibilities depends on environmental factors (such as timing), which we choose to ignore during modeling.

The initialization statement at line 60 creates one instance of the `User` machine, and this is the starting point for the model checker to perform verification. Note that the initial state of the `User` machine creates an instance of `Elevator` (in line 5), and the `Elevator` instance in turn creates instances of `Timer` and `Door` (in lines 11 and 12 in Figure 1. During execution, it is the responsibility of some external code to create an instance of the `Elevator` machine.

B. Examples

B.1 Memory Management for the Elevator

In our Elevator example from Figures 1 and 2, we created only one instance of the `Elevator` machine using the **new** command in line 5 of Figure 2, and never deleted that instance.

Figure 13 shows the `Elevator` machine in a more dynamic setting where the environment repeatedly creates and deletes instances of the `Elevator` machine. The `Elevator` machine now handles two new events `Inc` and `Dec`, which it uses to increment and decrement a reference count. When the reference count reaches 0 on handling a `Dec` event, the `Elevator` machine invokes the **delete** statement (see lines 17–20). The `User` machine initially allocates an instance of `Elevator` (in line 36 of Figure 13, and sends an `Inc` event to that instance in after creating it (in line 37). Nondeterministically, it deletes the `Elevator` instance by sending a `Dec` event (see line 42), and goes back to the `Init` state and create a fresh instance of `Elevator`.

In the `Elevator` machine, the `Inc` and `Dec` events are handled using call transitions in lines 29–30, with the corresponding target states described in lines 17–19. The wild card "_" in the source state of these transitions is a macro directive, which expands to a transition for every possible value of the source state. The `Inc` transition results in the variable `refcount` being incremented, and the `Dec` transition results in the variable `refcount` being decremented, and **delete** is executed if the `refcount` reaches 0 after the decrement.

When the elevator interacts with other machines (such as `Door` or `Timer`), the Elevator sends itself an `Inc` event so that it cannot be freed while it is waiting for a a response to come back from the `Door` or `Timer`. Otherwise, a runtime exception can be generated when `Door` or `Timer` sends its response to a freed instance of

```
1   ...
2   //include all the events from Figure 1, and in
3   //addition the two events below
4   event Inc, Dec
5
6   machine Elevator
7   ...
8   var refcount = 0
9   ...
10  state
11  ...
12       (AcquireRef, {}, {}, send(Elevator, Inc);
13                              raise(unit) )
14       (ReleaseRef, {}, {}, send(Elevator, Dec)
15                              raise(unit) )
16  ...
17       (Increment, {}, {}, refcount = refcount + 1)
18       (Decrement, {}, {}, refcount = refcount - 1;
19                            if(refcount =0) delete)
20
21  step
22  ...
23       (Closed,       OpenDoor,   AcquireRef)
24       (AcquireRef,   unit,       Opening)
25       (DoorOpening,  DoorOpened, ReleaseRef)
26       (ReleaseRef,   unit,       Opened)
27  ...
28  ...
29  call  (_, Inc, Increment)
30        (_, Dec, Increment)
31
32  ghost machine User
33   var Elevator : Elevator
34   event unit, restart
35   state
36    (Init, {}, {},  Elevator = new Elevator();
37                    send(Elevator, Inc);
38                    raise(unit) )
39    (Loop, {}, {},  if(*) {send(Elevator, OpenDoor); raise(unit) }
40                    else if (*) {send(Elevator, CloseDoor);
41                                raise(unit) };
42                    else {send(Elevator, Dec); raise(restart) )
43   step
44       (Init, unit, Loop)
45       (Loop, unit, Loop)
46       (Loop, restart, Init);
```

Figure 13: Elevator with memory management

`Elevator`. To prevent such runtime exceptions, an increment of the reference count is done by the `Elevator` during the time it is expecting a response from any other machine. For instance, when the `OpenDoor` event is received, the `Elevator` increments its reference count and keeps it incremented until the `DoorOpening` event is received from the `Door`, upon which the reference count is decremented back. This is shown using the transitions in line 23–26 and the state descriptions in lines 12–15. Similar increment and decrement operations need to be done for every state in the `Elevator` where it waits for responses from `Timer` or `Door`.

B.2 Switch-LED Driver

The P code for the Switch-LED driver can be found in Figure 14 and the ghost machines for the environment can be found in Figure 15.

C. Type System and Erasure

C.1 Type System

The type system of P, shown in Figure 16, 17, and 18, is on purpose kept very simple. It mostly does simple checks to make sure the machines, transitions, and statements are well-formed. In particular, the following checks are performed: (1) check that identifiers for machines, state names, events, and variables are unique, (2) check that statements inside real machines are deterministic, (3) ensure that private events are raised and public events are sent, and (4) ensure that ghost machines, ghost variables and ghost events can be *erased* during compilation and execution.

Notations. A machine is a 6-tuple (N, E, V, F, S, T, C) where N is the machine name, E is the private events, V is the variables, F is the foreign functions declaration, S is the states, T is the transitions, C is the calls.

A state is the triple (n, d, s) where n is the state name, d is the list of deferred events, s is the entry function (statement).

The symbols are split into two environment. The global one, denoted Γ, which contains the global events and machines.
The local one, denoted Λ, contains information about the local machine, private events, variables, states, and transitions.

We use the _ symbol as wild card for state names and as "don't care" value for boolean expressions. There are a few auxiliary functions. $isGhost$ holds when the argument is a ghost value. $isInit$ is true when the given state name correspond to the initial state of the machine. $current$ extracts the name of the machine being typed from the local environment. Furthermore, We omit rules for the existence of base types.

Explanation of the rules.

First, there are three global rules that check the global structure of the program with the machines and events. The top-level PROGRAM rules checks that the event and machines are well-defined. EVENT makes sure that the payload of is well-typed and that its name is unique. MACHINE checks that the machine name is unique and that each component is well-typed.

Then, we have the rules for the inner components of a machine. INTERNAL EVENT performs the same checks as EVENT with the additional local scope of the machine. VARIABLE DECL. verifies that the type exists and that there is no other variable with the same name. TRANSITION and CALL checks that the source and target states exist and that there is no ambiguity when receiving the event. i.e. there is no other edge with the same source and the same event. STATE first verifies the uniqueness of the state name and that the deferred set contains only known events. Then is tries to find the type of the payload. The payload is accessible only if all the incoming edges carries event with a similar payload.

The rules for statements contains the first checks for ghost values. Ghost values are allowed in assertions, assignment to ghost variables and send operations. ASSIGN makes sure that the types of both sides of the assignment match and that the separation of ghost and real values is respected. This separation is strict for the machines ids and more permissive for the other types. ASSERT is as expected. Assertions that contains ghost values are only checked during the verification phase. They are removed at compile time. SEND checks the payload and make sure that only public event are sent. RAISE is similar as SEND except that only private events can be raised. IF-THEN-ELSE and WHILE corresponds to the expected rules with the addition that ghost values cannot be used as part of the condition. The remaining rules are as expected.

Finally, the rules for the expressions are straightforward. They contains an additional flag to tell whether a ghost values have to be used.

C.2 Erasure

Given a set of ghost and real machines our compiler removes the ghost ones and modifies the real machines to erase the ghost variables and expressions. Inside real machines, we can distinguish two different usages for ghost values: (1) reference to ghost machines that models the environment, and (2) non-$\mathbf{Id}$ ghost variables that

are used in assertions. Both cases are handled in a sightly different way.

The simplest case is (2). These variables and the expression in which they occurs are simply removed since the type system guarantees that ghost terms do not affect the runs of real machines if the assertion are valid. The only change in the semantics occurs for unsafe programs, i.e. the programs which contains invalid ghost assertions.

In case (1), the compiler replace the **send** to a ghost machine by the appropriate call to foreign code, see Section 4 for the details. This explains why for machine identifiers we enforce complete separation of ghost and real **Id**. We need to unambiguously identify the **send** operation that targets ghost machine.

D. Details of the Runtime

Figure 19 shows a simplified implementation of the entry methods of the runtime API. Figure 20 and 21 present a simplified version of some internal methods of the runtime. Figure 20 shows the methods that run a machine and Figure 21 shows the methods that interacts with the event queue.

The internal method `SMGetNextEvent` gives priority to an internal event (if one exists, stored in `smc->PrivateEvent`) over external events (stored in `smc->EventQueue`). The method `SMReturn` pops the state on top of the call stack. It is called from the generated code. Pushes to the call stack are done while executing call transitions (see line 37, in Figure 20). The method `UpdateDeferredEventSet` is used to update the deferred sets in lines 27 and 36, during push and pop in accordance with the rules CALL and POP from Figure 4.

E. Details of the Verification

Algorithm for the delay bounded scheduler. The delay bounded scheduler's rules presented in Section 5 can easily be turned into an nondeterministic algorithm as shown by Algorithm 1. The scheduler does not need much information about the machine that are running. The RUNTOSENDORCREATE method let the machine run until the next scheduling point which is either a send or the creation of a machine. In both cases, the method returns (m') the identifier of the recipient or the new machine. In case the machine does not reach a context-switch point the method returns $\bot$ and m is popped from the stack. This may happen if a machine blocks when trying to receive a message or if it calls **delete**.

Algorithm 1 Pseudocode for delay bounded scheduler

```
Require: d ≥ 0 a bound on the delay, i the initial machine
  currD ← 0                                    ▷ initialization
  S ← new DoubleEndedQueue()
  PUSH_FRONT(S, i)
  while ¬ EMPTY(S) do
      m ← POP(S)
      if ∗ then                                 ▷ delaying
          if currD < d then
              currD ← currD + 1
              PUSH_BACK(S, m)
      else                                      ▷ no delay
          m′ ← RUNTOSENDORCREATE(m)
          if m′ ≠ ⊥ then
              PUSH_FRONT(S, m)
              if m′ ∉ S then
                  PUSH_FRONT(S, m′)
```

Responsiveness as Büchi automata. In Figure 8, we presented an automata for the condition (2) of responsiveness. Figure 22 shows a similar automaton for the condition (1). The structure of the automaton is similar. The difference lies in the edges label. Furthermore, the automaton has only one parameter m.

```
event D0Entry, D0Exit, TimerFired, SwitchStatusChange,
      TransferSuccess, TransferFailure,
      StopTimer, StartDebounceTimer,
      UpdateBargraphStateUsingControlTransfer,
      SetLedStateToUnstableUsingControlTransfer,
      SetLedStateToStableUsingControlTransfer

machine Driver
 event OperationSuccess, OperationFailure, TimerStopped,
       Yes, No, unit
   E2: set of Event = {D0Entry, D0Exit, SwitchStatusChange,
                       TimerFired, TimerStopped}
   E3: set of Event = {SwitchStatusChange,
                       TransferSuccess, TransferFailure,
                       D0Entry, D0Exit}
 ghost var Timer:Timer, LED: LED, Switch : Switch
 function
   void StoreSwitchAndEnableSwitchStatusChange()
   bool CheckIfSwitchStatusChanged()
   void CompleteDStateTransition()
   void UpdateBarGraphStateUsingControlTransfer(){
       send(LED, UpdateBarGraphStateUsingControlTransfer) }
   void SetLedStateToStableUsingControlTransfer(){
       send(LED, SetLedStateToStableUsingControlTransfer) }
   void SetLedStateToUnstableUsingControlTransfer(){
       send(LED, SetLedStateToUnstableUsingControlTransfer) }
   void StartDebounceTimer(){ send(Timer, StartDebounceTimer) }
 state
   (Init, {SwitchStatusChange}, {}, Timer = new Timer(Driver=this);
              LED  = new LED(Driver=this);
              Switch = new Switch(Driver=this); raise(unit))
   (Dx, {SwitchStatusChange}, {D0Exit}, skip)
   (CompleteD0Entry, {SwitchStatusChange}, {}, raise(OperationSuccess) )
   (WaitingForSwitchStatusChange, {}, {D0Entry}, skip)
   (CompletingD0Exit, {}, {}, raise(OperationSuccess) )
   (StoringSwitchAndCheckingIfStateChanged, {}, {D0Entry},
              StoreSwitchAndEnableSwitchStatusChange();
              if ( CheckIfSwitchStatusChanged() ) raise(Yes)
              else raise(No) )
   (UpdatingBarGraphState, E2, {D0Entry},
              UpdateBarGraphStateUsingControlTransfer())
   (UpdatingLedStateToUnstable, E2, {D0Entry},
              SetLedStateToUnstableUsingControlTransfer())
   (WaitingForTimer, {}, {D0Entry}, StartDebounceTimer())
   (UpdatingLedStateToStable, E2, {D0Entry},
              SetLedStateToStableUsingControlTransfer())
   (StoppingTimerOnStatusChange, E3, {D0Entry}, raise(unit))
   (StoppingTimerOnD0Exit, E3, {D0Entry}, raise(unit))
   //substate machine
   (StoppingTimer, {}, {D0Entry}, send(Timer, StopTimer))
   (WaitingForTimerToFlush, E3, {D0Entry}, skip)
   (ReturningTimerStopped, {}, {D0Entry}, raise(TimerStopped) )
 step
   (Init, unit, Dx)
   (Dx, D0Entry, CompleteD0Entry)
   (CompleteD0Entry, OperationSuccess, WaitingForSwitchStatusChange)
   (WaitingForSwitchStatusChange, D0Exit, CompletingD0Exit)
   (CompletingD0Exit, OperationSuccess, Dx)
   (WaitingForSwitchStatusChange, SwitchStatusChange,
                            StoringSwitchAndCheckingIfStateChanged)
   (StoringSwitchAndCheckingIfStateChanged, Yes, UpdagingBarGraphState)
   (StoringSwitchAndCheckingIfStateChanged, No, WaitingForTimer)
   (UpdatingBarGraphState, TransferSuccess, UpdatingLedStateToUnstable)
   (UpdatingBarGraphState, TransferFailure, UpdatingLedStateToUnstable)
   (UpdatingLedStateToUnstable, TransferSuccess, WaitingForTimer)
   (WaitingForTimer, TimerFired, UpdatingLedStateToStable)
   (UpdatingLedStateToStable, TransferSuccess,
                                 WaitingForSwitchStatusChange)
   (WaitingForTimer, SwitchStatusChange, StoppingTimerOnStatusChange)
   (StoppingTimerOnStatusChange, TimerStopped,
                            StoringSwitchAndCheckingIfStateChanged)
   (WaitingForTimer, D0Exit, StoppingTimerOnD0Exit)
   (StoppingTimerOnD0Exit, TimerStopped,  CompletingD0Exit)
   //submachine steps
   (StoppingTimer, StoppingSuccess, ReturningTimerStopped)
   (StoppingTimer, StoppingFailure, WiatingForTimerToFlush)
   (StoppingTimer, TimerFired, ReturningTimerStopped)
   (WaitingForTimerToFlush, TimerFired, ReturningTimerStopped)
 call
   (StoppingTimerOnStatusChange, unit, StoppingTimer)
   (StoppingTimerOnD0Exit, unit, StoppingTimer)
```

Figure 14: Switch-LED Device Driver

```
ghost machine User
 event unit
 var Driver:Driver
 state
     (Init, {},{}, Driver = new Driver(); raise(unit))
     (S0, {},{}, send(Driver, D0Entry); raise(unit))
     (S1, {},{}, send(Driver, D0Exit); raise(unit))
 step
     (Init, unit,  S0)
     (S0, unit, S1)
     (S1, unit, S0)


ghost machine Switch
 event unit
  var Driver:Driver
  state
      (Init, {},{}, raise(unit))
      (ChangeSwitchState, {},{},
           send(Driver, SwitchStatusChange);
           raise(unit) )
  step
      (Init, unit, ChangeSwitchState)
      (ChangeSwitchState, unit, ChangeSwitchState)

ghost machine LED
 event unit
 var Driver:Driver
   state
       (Init, {},{}, skip)
       (ProcessUpdate, {},{}, if(*) send(Driver, TransferFailure)
                              else  send(Driver, TransferSuccess);
                              raise(unit) )
       (Unstable, {},{},      send(Driver, TransferSuccess))
      (Stable, {},{},        send(Driver, TransferSuccess))
   step
       (Init, UpdateBarGraphStateUsingControlTransfer, ProcessUpdate)
       (ProcessUpdate, unit, Init)
       (Init, SedLedStateToUnstableUsingControlTransfer, Unstable)
       (Init, SedLedStateToStableUsingControlTransfer, Stable)
       (Unstable, SetLedStateToStableUsingControlTransfer, Init)
   (Stable, unit, Init)

//Timer fires when started
ghost machine Timer
 event unit
 var Driver:Driver
   state
       (Init, {}, {StopTimer} skip)
       (TimerStarted,{StartDebounceTimer}, {}, if(*) raise(unit))
       (SendTimerFired,{StartDebounceTimer}, {},
                       send(Driver, TimerFired,null); raise(unit))

       (ConsiderStopping,{StartDebounceTimer},{},
                            if(*) {send(Driver, StoppingFailure);
                                   send(Driver, TimerFired);}
                            else  send(Driver, StoppingSuccess);
                            raise(unit))
   step
       (Init, StartDebounceTimer, TimerStarted)
       (TimerStarted, unit, SendTimerFired)
       (sendTimerFired, unit, Init)

       (TimerStarted, StopTimer,ConsiderStopping)
       (ConsiderStopping, unit, Init)

//initial machine. life starts here!!!
User()
```

Figure 15: Environment for Switch-LED driver

$$\frac{\begin{array}{c} i = |E| \quad \forall k \in [1;i].\ \cup_{l \neq k} E_l \vdash E_k \\ j = |M| \quad \forall k \in [1;j].\ E, \cup_{l \neq k} M_l \vdash M_k \\ E, M \vdash I \end{array}}{\vdash E\ M\ I}\ (\textsc{Program})$$

$$\frac{e \notin \Gamma \quad \vdash t}{\Gamma \vdash e(t)}\ (\textsc{Event})$$

$$\frac{\begin{array}{c} N \notin \Gamma \\ i = |E| \quad \forall x \in [1;i].\ \Gamma, M\ \cup_{y \neq x} E_y \vdash E_x \\ j = |V| \quad \forall x \in [1;j].\ \Gamma, M\ \cup_{y \neq x} V_y \vdash V_x \\ k = |F| \quad \forall x \in [1;k].\ \Gamma, M\ \cup_{y \neq x} F_y \vdash F_x \\ l = |T| \quad \forall x \in [1;l].\ \Gamma, M\ E\ \cup_{y \neq x} T_y \vdash T_x \\ n = |C| \quad \forall x \in [1;n].\ \Gamma, M\ E\ T\ \cup_{y \neq x} C_y \vdash C_x \\ m = |S| \quad \forall x \in [1;m].\ \Gamma, M\ E\ V\ F\ \cup_{y \neq x} S_y\ t \vdash S_x \end{array}}{\Gamma \vdash (N, E, V, F, S, T, C)}\ (\textsc{Machine})$$

$$\frac{e \notin \Gamma \cup \Lambda \quad \vdash t}{\Gamma, \Lambda \vdash e(t)}\ (\textsc{Internal Event}) \qquad \frac{\vdash t \quad v \notin \Lambda}{\Gamma, \Lambda \vdash v : t}\ (\textsc{Variable Decl.})$$

$$\frac{(\vdash t) \vee t = \mathbf{void} \quad f \notin \Lambda \quad \forall i \in [1,k] \vdash t_k}{\Gamma, \Lambda \vdash f : t_1 \to \dots t_k \to t}\ (\textsc{Foreign Decl.})$$

$$\frac{\begin{array}{c} e \in \Gamma \cup \Lambda \quad l_1 \in \Lambda \quad l_2 \in \Lambda \\ \forall l.\ l \neq l_2 \implies (l_1, e, l) \notin \Lambda \end{array}}{\Gamma, \Lambda \vdash (l_1, e, l_2)}\ (\textsc{Transition})$$

$$\frac{\begin{array}{c} e \in \Gamma \cup \Lambda \quad l_1 \in \Lambda \quad l_2 \in \Lambda \\ \forall l.\ l \neq l_2 \implies (l_1, e, l) \notin \Lambda \end{array}}{\Gamma, \Lambda \vdash (l_1, e, l_2)}\ (\textsc{Call})$$

$$\frac{\begin{array}{c} n \notin \Lambda \quad d \subseteq \Gamma \quad \forall e \in d.\ \mathit{Trans}(\mathit{current}(\Lambda), n, e) = \bot \\ \text{let } t \text{ s.t. } \forall (_, e, n) \in \Lambda, \mathit{Payload}(e) = t \\ \text{if } t \text{ exists } \wedge \neg \mathit{isInit}(n) \text{ then } \Gamma, \Lambda \cup \{\mathbf{arg} : t\} \vdash s \\ \text{otherwise } \Gamma, \Lambda \vdash s \end{array}}{\Gamma, \Lambda \vdash (n, d, s)}\ (\textsc{State})$$

Figure 16: Type checking rules for the program structure

$$\frac{\begin{array}{c} \Lambda(x) = t \\ \mathit{isGhost}(x) \wedge t = \mathit{id} \implies \Gamma, \Lambda, \mathbf{true} \vdash r : \mathit{id} \\ \mathit{isGhost}(x) \wedge t \neq \mathit{id} \implies \Gamma, \Lambda, _ \vdash r : t \\ \neg \mathit{isGhost}(x) \implies \Gamma, \Lambda, \mathbf{false} \vdash r : t \end{array}}{\Gamma, \Lambda \vdash x := r}\ (\textsc{Assign})$$

$$\frac{\Gamma, \Lambda, _ \vdash r : \mathbf{bool}}{\Gamma, \Lambda \vdash \mathbf{assert}(r)}\ (\textsc{Assert}) \qquad \frac{}{\Gamma, \Lambda \vdash \mathbf{delete}}\ (\textsc{Delete})$$

$$\frac{\Gamma, \Lambda, _ \vdash r : \mathit{id} \quad \Gamma(e) = t \quad \Gamma, \Lambda, \mathbf{false} \vdash r' : t}{\Gamma, \Lambda \vdash \mathbf{send}(r, e, r')}\ (\textsc{Send})$$

$$\frac{\Lambda(e) = t \quad \Gamma, \Lambda, \mathbf{false} \vdash r : t}{\Gamma, \Lambda \vdash \mathbf{raise}(e, r)}\ (\textsc{Raise})$$

$$\frac{}{\Gamma, \Lambda \vdash \mathbf{return}}\ (\textsc{Return}) \qquad \frac{}{\Gamma, \Lambda \vdash \mathbf{skip}}\ (\textsc{Skip})$$

$$\frac{\begin{array}{c} (\Gamma, \Lambda, \mathbf{false} \vdash r : \mathbf{bool}) \vee (\mathit{isGhost}(\Lambda) \wedge r = \mathbf{choose}(\mathbf{bool})) \\ \Gamma, \Lambda \vdash s_1 \quad \Gamma, \Lambda \vdash s_2 \end{array}}{\Gamma, \Lambda \vdash \mathbf{if}\ r\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2}\ (\textsc{If-Then-Else})$$

$$\frac{\Gamma, \Lambda, \mathbf{false} \vdash r : \mathbf{bool} \quad \Gamma, \Lambda \vdash s}{\Gamma, \Lambda \vdash \mathbf{while}\ r\ s}\ (\textsc{While})$$

Figure 17: Type checking rules for the statments

$$\frac{m \in \Lambda \quad \mathit{isGhost}(m) = g}{\Gamma, \Lambda, g \vdash \mathbf{this} : m}\ (\textsc{This}) \qquad \frac{\Lambda(\mathbf{arg}) = t}{\Gamma, \Lambda, \mathbf{false} \vdash \mathbf{arg} : t}\ (\textsc{Arg})$$

$$\frac{}{\Gamma, \Lambda, _ \vdash c : \mathbf{int}}\ (\textsc{Constant}) \qquad \frac{\Lambda(x) = t \quad \mathit{isGhost}(t) = g}{\Gamma, \Lambda, g \vdash x : t}\ (\textsc{Variable})$$

$$\frac{m \in \Gamma \quad g = \mathit{isGhost}(m) \quad \forall k \in [1;j].\Gamma, m \vdash i_k}{\Gamma, \Lambda, g \vdash \mathbf{new}\ m(i_1 \dots i_j) : \mathit{id}}\ (\textsc{New})$$

$$\frac{\Gamma, \Lambda, g \vdash r_1 : t_1 \quad \Gamma, \Lambda, g \vdash r_2 : t_2 \quad \mathsf{op} : t_1 \to t_2 \to t_3}{\Gamma, \Lambda, g \vdash r_1\ \mathsf{op}\ r_2 : t}\ (\textsc{Op})$$

$$\frac{\begin{array}{c} \Lambda(f) = \ldots \to t \quad \mathit{arity}(\Lambda(f)) = k \\ \forall i \in [1;k].\ \Gamma, \Lambda, \mathbf{false} \vdash e_i : t_i\ \wedge\ \Lambda(f)[i] = t_i \end{array}}{\Gamma, \Lambda, \mathbf{false} \vdash f(e_1, \ldots, e_k) : t}\ (\textsc{ForeignCall})$$

Figure 18: Type checking rules for the expressions

```
Hashtable *map; //maps machines to contexts
int  SMCreateMachine(string machineType){
  int mid = getNewMachineId();
  void *smc = AllocateMachineContext(machineType);
  HasbtableAdd(map, mid, smc);
  return(mid);
}
void  SMDeleteMachine(int mid){
  void *smc = HashtableLookup(map, mid);
  if(smc == null) raise MACHINE_NONEXISTENT;
  HashtableRemove(map, mid);
  DeleteMachineContext(smc);
}
void SMAddEvent(int mid, String n, void *v){
  Event e; e.name = n; e.value = v;
  StateMachineContext *smc = HashtableLookup(map,mid);
  if(smc == null) raise MACHINE_NONEXISTENT;
  SMEnqueueEvent(smc,e, false);
  WdfSpinLockAcquire(smc->lock);
  if(!smc->IsRunning) SMRunStateMachine(smc);
  else WdfSpinLockRelease(smc->lock);
}
void SMAddPrivateEvent(StateMachineContext *smc, String n,
                      void *v){
  Event e; e.name = n; e.value = v;
  SmEnqueueEvent(smc,e,true);
}
void SMReturn(StateMachineContext *smc){
  UpdateDeferredEventsSet(smc);
  Pop(smc->CallStack);
}
```

Figure 19: Runtime API: External methods

```
internal SMRunStateMachine(StateMachineContext *smc){
  while(true){
    Event e = SMGetNextEvent(smc);
    if(e != NULL_EVENT) {
      smc->IsRunning=true;
      WdfSpinLockRelease(smc->lock);
      smc->CurrentState=
        SMExecuteTransition(smc,e);
    } else {
      smc->IsRunning=false;
      WdfSpinLockRelease(smc->lock);
      return;
    }
  }
}

internal State SMExecuteTransition(
    StateMachineContext *smc,
    Event e) {
  bool foundTransition = false;
  while(NotEmpty(smc->CallStack){
    State curState = Top(smc->CallStack);
    if(GNExistsTransition(curState,e)) {
      foundTransition = true;
      break;
    } else {
      UpdateDeferredEventsSet(smc);
      Pop(smc->CallStack);
    }
  }
  if(!foundTransition) raise UNHANDLED_EVENT;
  State newState =
      GNGetTargetStateOfTransition(curState, e);
  Bool isCall = GNIsCallTransition(curState, e);
  if(isCall){
    UpdateDeferredEventsSet(smc);
    Push(smc->CallStack,newState);
  }
  GNExecuteEntryFunction(newState, smc->LocalVariables);
  return(newState);
}
```

Figure 20: Runtime: Internal methods for transitions

```
internal Event SMGetNextEvent(StateMachineContext *smc){
  Event returnedEvent = NULL_EVENT;
  if(smc->privateEvent != NULL_EVENT) {
    returnedEvent = smc->privateEvent;
    smc->privateEvent = NULL_EVENT;
  } else {
    int curr = smc->EventQueue->Head;
    while(curr != smc->EventQueue->Tail){
      if(NotInDeferredEvents(smc, curr->e)){
        returnedEvent = smc->EventQueuecurr;
        break;
      } else {
        curr++;
      }
    }
  }
  return(returnedEvent);
}

internal void SMEnqueueEvent(StateMachineContext *smc,
                    Event e, bool isPrivate){
  WdfSpinLockAcquire(smc->lock);
  //Enqueue event should be an atomic operation
  if(isPrivate){
    assert(smc->PrivateEvent == NULL_EVENT);
    smc->privateEvent = e;
  } else {
    Enqueue(smc->EventQueue,e);
  }
  WdfSpinLockRelease(smc->lock);
}
```

Figure 21: Runtime: Internal methods for events

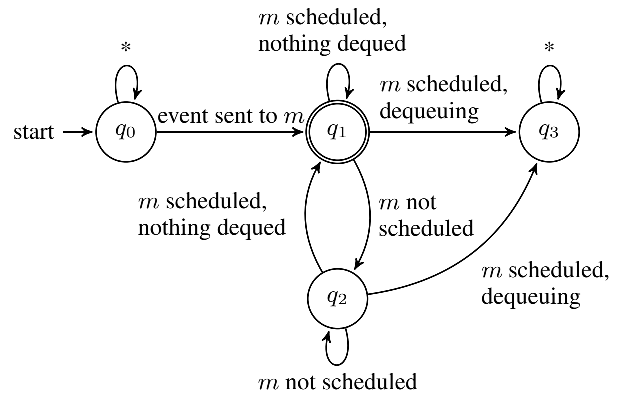


Figure 22: Büchi automaton for responsiveness condition (1)