P: Safe Asynchronous Event-Driven Programming

Ankush Desai Vivek Gupta
Ethan Jackson Shaz Qadeer
Sriram Rajamani
Microsoft

Damien Zufferey
IST Austria

Abstract

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 tested using model checking techniques. P allows the programmer to specify the environment, used to "close" the system during testing, 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 designed so that a P program can be checked for responsiveness—the ability to handle every event in a timely manner. By default, a machine needs to handle every event that arrives in every state. But handling every event in 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 default safety checker looks for presence of unhandled events. The language also provides default liveness checks that an event cannot be potentially deferred forever.

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); we have more confidence in the robustness of its design due to the language abstractions and verification provided by P.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification– Model checking; D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.2 [*Programming Languages*]: Language Classifications– Specialized application languages, Concurrent, distributed, and parallel languages; D.3.3 [*Programming Languages*]: Language Constructs and Features– Concurrent programming structures, Control structures

Keywords domain-specific language; device driver; event-driven programming; state machine; verification; systematic testing

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 a language 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 maintain. 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 are now specified as state machines and asynchronous driver code is generated from these state machines. We are able to use state exploration techniques directly on the state machines to find and fix design bugs. Since the executable code was 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 is 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, we represent P as a textual language with a simple core calculus, so that we can give a formal treatment of the language, compiler and verification algorithms. A P program is a collection of state machines communicating via events. Each state machine has a collection of states, local variables, and actions. The states and actions are annotated with code statements to read and update local variables, send events to other state machines, raise events locally, or call external C functions. The external C functions are used to write parts of the code that have do with data transfer. A machine responds to received events by executing transitions and actions which in turn causes the aforementioned code fragments to execute. For programming convenience, call transitions are used to factor out common code that needs to be reused (similar to nested modes in state charts [11]).

Components in an operating system 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 [4] (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 [12] (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.

Reactive systems, such as device drivers, typically interact with their environment, both synchonously via function calls and asynchronously via events. The reliability of the system depends critically on the correct handling of all interactions via stateful protocols. To allow reasoning about such interactions, we allow programmers to model the environment of a P program using *ghost* machines and variables. These ghost elements 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 of the program. It is worth noting that both the real and ghost parts of a P program are based on the computational model of communicating state machines. This aspect of the P language effectively blurs the distinction between modeling and programming and makes the specification capabilities in the language more accessible to programmers.

A P program is validated via *systematic testing* [9, 17] of its inherent nondeterminism. Systematic testing is accomplished by interpreting the operational semantics of a P program (closed using ghost machines) in the explicit-state model checker Zing [2]. All aspects of the operational semantics of the program are interpreted including the code statements labeling the states and actions of a P machine. The model checker takes care of systematically enumerating all implicit scheduling and explicit modeling choices in the program. The number of states and executions of a P program is unbounded in general (in fact, reachability analysis of P programs is undecidable). Therefore, in practice, the enumeration is controlled by bounding techniques.

The simplest approach to bounding the exploration of nondeterministic transition systems is depth-bounding [19]. We have implemented this approach and found it useful for discovering errors witnessed by short executions. However, the complexity of depth-bounded search increases exponentially with execution depth, and consequently does not scale for systematic testing of large P programs, in which errors may be lurking in long executions. We use delay-bounded scheduling [6] to overcome this problem. A delaying scheduler is a deterministic scheduler with a "delay" operation, whose invocation causes the scheduler to change its default scheduling strategy. Given a delay budget d, a delaying scheduler naturally defines a set of schedules obtained by nondeterministically invoking the "delay" operation at most d times; the number of generated schedules (under the assumption that scheduling is the only source of nondeterminism) is independent of execution length and exponential in d; thus arbitrarily long executions can be generated even with a delay bound of 0. We expect most bugs that occur in practice to be found using low values of the delay bound. We have developed a new delaying scheduler for P programs; our scheduler prioritizes schedules that follow the causal sequence of events in the program. We provide empirical evidence to demonstrate that our scheduler indeed finds common errors with a small 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 and a compiler and runtime that enables P programs to run as KMDF (Kernel Mode Driver Framework) device drivers.
- We show how to validate P programs using delay-bounded scheduling and provide a novel delaying scheduler that, by default, attempts to schedule events according to their causal order.
- We report on the use of P in a production environment; our case study is the USB stack in Windows 8.

2. Overview

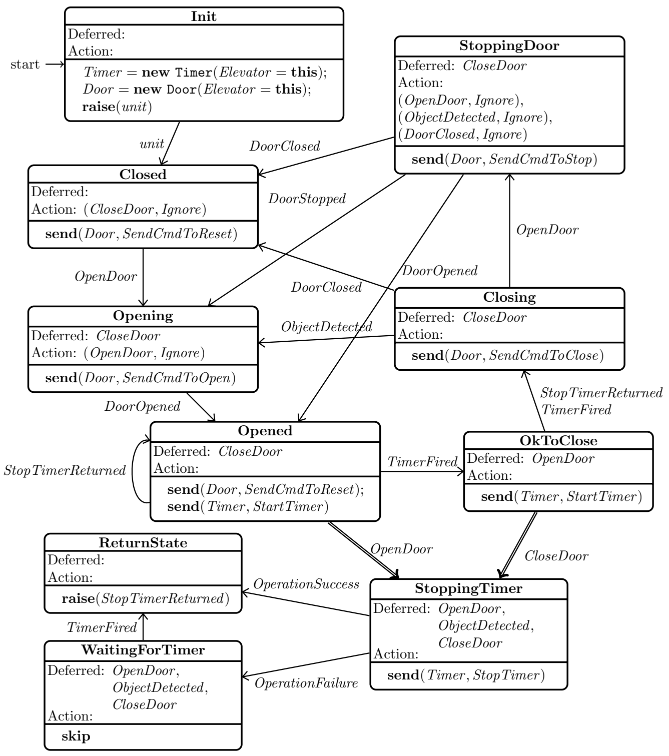


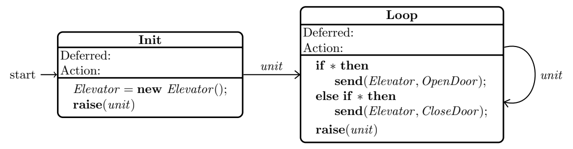
Figure 1: Elevator example

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*.

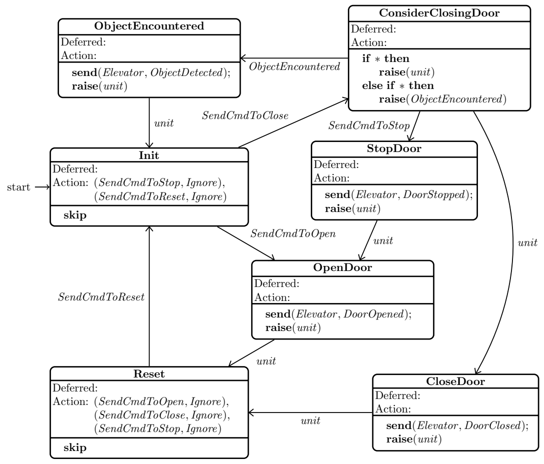
A P program is a collection of *machines*. Machines communicate with each other asynchronously through *events*. Events are queued, but machines are required to handle them in a responsive manner (defined precisely later)—failure to handle events is detected by automatic verification.

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.

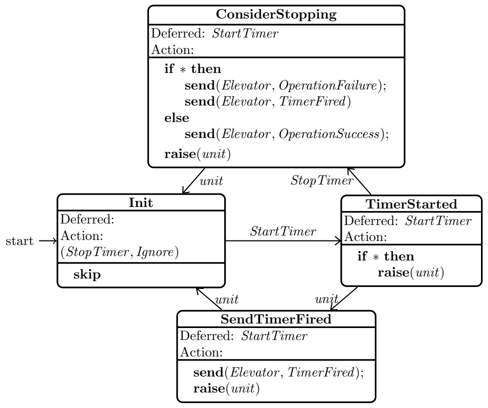
Machines communicate with each other using events. An event can be sent from one machine to another and or raised within a machine. Each machine is composed of control states, transitions, actions, and variables. The elevator machine has events *unit* and *StopTimerReturned* (which are used for communication locally inside the elevator machine), an action called *Ignore*, and two *ghost*



(a) `User` ghost machine



(b) `Door` ghost machine



(c) `Timer` ghost machine

Figure 2: Environment for elevator

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, a, s), where (1) n is a state name, (2) d is a set of events (called *deferred set*), (3) a is a set of (event, action) pairs (called *action handlers*), 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, no action handlers, and an entry statement that creates an instance of the `Timer` and `Door` machines and raises the event *unit*. As another example, the **Opening** state has $\{CloseDoor\}$ as the deferred set, a single action handler (*OpenDoor*, *Ignore*), 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 state of this machine instance is initialized to **Init**.

Deferred events and action handlers. 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. When trying to receive an event a machine scans its event queue, starting from the front dequeuing the first event that is not in the deferred set. A dequeued event is either processed by executing an action handler or executing an outgoing transition. An action is simply a named piece of code. The *Elevator* machine has a single action called *Ignore* that does nothing. For instance, in the **Opening** state, the event *CloseDoor* is deferred and therefore never dequeued. If the event *OpenDoor* is dequeued, the *Ignore* action is executed (which just drops the event on the floor) and control stays in **Opening**. If the event *DoorOpened* is dequeued, the outgoing transition labeled by *DoorOpened* is taken and control moves to state **Opened**.

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 shown by simple edges and call transitions by double edges. For instance, when the machine is in the **Init** state, if an *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, 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 to model the environment during verification, but no code is generated 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 "$\mathbf{if} * \mathbf{then}\ \mathbf{raise}(unit)$". 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

program	::=	evdecl machine$^+$ m(init*)
machine	::=	optghost **machine** m
		vrdecl* actdecl* stdecl*
		spdecl* cldecl* acdecl*
optghost	::=	$\epsilon \mid$ **ghost**
evdecl	::=	**event** edecl$^+$
vrdecl	::=	optghost **var** vdecl$^+$
actdecl	::=	**action** $(a, \text{stmt})^+$
stdecl	::=	**state** $(n, \{e_1, e_2, \ldots, e_k\}, \text{stmt}, \text{stmt})^+$
spdecl	::=	**step** $(n, e, n)^+$
cldecl	::=	**call** $(n, e, n)^+$
acdecl	::=	**act** $(n, e, a)^+$
edecl	::=	e(type)
vdecl	::=	x : type
type	::=	**void** $\mid$ **bool** $\mid$ **int** $\mid$ **event** $\mid$ **id**
stmt	::=	**skip**
	$\mid$	$x :=$ expr
	$\mid$	$x :=$ **new** m(init*)
	$\mid$	**delete**
	$\mid$	**send**(expr, e, expr)
	$\mid$	**raise** (e, expr)
	$\mid$	**leave**
	$\mid$	**return**
	$\mid$	**assert**(expr)
	$\mid$	stmt; stmt
	$\mid$	**if** expr **then** stmt **else** stmt
	$\mid$	**while** expr stmt
init	::=	$x =$ expr
expr	::=	**this** $\mid$ **msg** $\mid$ **arg** $\mid b \mid c \mid\perp\mid x \mid *$
		$\mid$ uop expr $\mid$ expr bop expr
$c \in$ **int**		$b \in$ **bool**
$\neg, - \in$ uop		$+, -, \wedge, \vee \in$ bop
$r \in$ expr		$a, e, m, x \in$ name

Figure 3: Syntax

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 a 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, the external code is responsible for creating an instance of the `Elevator` machine.

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_1, s_2), where n is a state name, d is a set of deferred events, s_1 is an entry statement, and s_2 is an exit statement.

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 actions, (5) a list of states, (6) a list of transitions, and (7) a list of action bindings. Each variable has a declared type, which can be int, byte, bool, event or machine identifier type (denoted **id**). Actions associate an action name with a statement. Transitions are one of two types: steps or calls, and action bindings associate state-event pairs with actions.

A machine can optionally be declared as *ghost* by prefixing its declaration by the keyword **ghost**. Variables can be also declared as **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 two statements: (1) an entry statement and (2) an exit statement. Each state declaration must have a distinct name. Thus, we can use the name n to denote the state. The entry statement associated with a state n is executed whenever control enters n, and the exit statement associated with state n is executed whenever control leaves 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 $\mathit{Action}(m, n, e)$ be an that action a is associated with event e in state n, if such a binding exists or $\perp$ otherwise. Let $\mathit{Entry}(m, n)$ denote the associated entry statement, and let $\mathit{Exit}(m, n)$ denote the associated exit statement. The initial state of the machine m is the first state in the state list and is denoted by $\mathit{Init}(m)$.

Each action declaration consists of an action name and a statement. Let $\mathit{Stmt}(m, a)$ denote the statement associated with action a in machine m.

Transition declarations describe how a state responds to events. The list of transitions is partitioned into step transitions, and call transitions. A step transition from state n to another state n_1 involves executing the exit statement of n and the entry statement of n_1. A call transition is similar to function calls in programming languages and is implemented using a stack (more details below). 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 $n_1 = n_2$.

An action binding does not change the state, but merely executes the statement associated with the action.

A statement (be it an entry statement or exit statement associated with a state, or associated with an action) is obtained by composing primitive statements using standard control flow constructs such as sequential composition, conditionals, and loops. Primitive statements are described below. The **skip** statement does nothing. The assignment $x := r$ evaluates an expression r and writes the result into x. The statement $x :=$ **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 **delete** statement terminates the current machine (which is executing the statement) and release its resources. The statement **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. When e does not have any argument `null` is expected. In the examples, we use **send**(r_1, e) as syntactic sugar for **send**$(r_1, e,$ `null`$)$. The statement **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 **leave** statement jumps control to end of the entry function to wait for an event to be dequeued. The **return** statement terminates the evaluation of the statement and returns to the caller (see below for more details). The statement **assert**(r) moves the machine to an error state of the expression r evaluates to false, and behaves like **skip** otherwise.

Expressions and evaluation. The expressions in the language, in addition to the declared variables, can also refer to three special variables—**this**, **msg** and **arg**. While **this** is a constant containing the identifier of the executing machine, **msg** contains the event that is last received from the input buffer of the machine, and **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 $\perp$. The value $\perp$ arises either as a constant, or if an expression reads a variable whose value is uninitialized, and propagate through operators in an expression. The expression $*$ represents nondeterministic choice of a Boolean value. Nondeterministic expressions are allowed only in ghost machines to conveniently model the environment.

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.

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 corresponding to identifier id is of the form (γ, σ, s, q) with components defined as follows:

- γ is a sequence of pairs (n, α), where n is a state name, and α is map from events to $A \cup \{\top, \perp\}$, where A is the set of all actions declared in machine $Name(id)$. This sequence functions as a call stack, to implement call and return, and the α values are used to inherit deferred events and actions from caller to callee. For an event e, $\alpha(e)$ can be an action a, or the value $\top$ indicating that the event is deferred, or the value $\perp$ which indicates that the event does not have an associated action and it is not deferred.
- σ is a map from variables declared in machine $Name(id)$ 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.

Our type checker verifies that for any event e and state n, there is at most one outgoing transition labeled with e out of n and at most one action bound to e in s. We define $Step(m, n, e)$ to be equal to n' if there is a step transition labeled e between n and n' in machine m and $\perp$ otherwise. Similarly, we define $Call(m, n, e)$ to be equal to n' if there is a call transition labeled e between n and n' in machine m and $\perp$ otherwise. We define $Trans(m, n, e)$ to be the union

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

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

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

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

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

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

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

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

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

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

Figure 4: Operational semantics: statement execution

of $Step(m, n, e)$ and $Call(m, n, e)$. Note that $Trans(m, n, e)$ is the static transition in state s on event e, $Action(m, n, e)$ is the static action bound with state n and event e, and $Deferred(m, n)$ is the static set of events deferred in state n. During execution, both deferred events and actions associated with a state can be inherited from callers, and these are modeled in the second component of the call stack, which is a sequence of pairs (n, α).

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 Figures 4, 5, and 6 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 $M[id_0] = ((Init(m), \lambda e.\ \perp), \lambda x.\ \perp, Entry(m, Init(m)), \epsilon)$. 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 and therefore omitted.

Figure 4 gives the rules for executing code statements inside a state. These rules execute small steps performed during the computation of the entry function of a state. During this computation,

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, S[\mathbf{raise}\ (e,r)], q) \\ \sigma(r) \downarrow v \qquad \sigma' = \sigma[\mathbf{msg} := e][\mathbf{arg} := v] \qquad m = \mathit{Name}(id) \\ s = \begin{array}{l} \mathbf{if}\ \mathit{Pop}(m,n,\alpha,e) \vee \mathit{Step}(m,n,e) \neq \bot \\ \mathbf{then}\ \mathit{Exit}(m,n) \\ \mathbf{else\ skip} \end{array} \end{array}}{M \longrightarrow M[id := ((n,\alpha)\cdot\gamma, \sigma', s; \overline{\mathbf{raise}}\ (e,v), q)]}\ (\textsc{Raise})$$

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

$$\frac{M[id] = (\gamma, \sigma, S[\mathbf{return}], q)}{M \longrightarrow M[id := (\gamma, \sigma, \mathit{Exit}(m,n); \overline{\mathbf{return}}, q)]}\ (\textsc{Return})$$

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, \mathbf{skip}, q_1\cdot(e,v)\cdot q_2) \qquad m = \mathit{Name}(id) \\ t = \{e \mid \mathit{Trans}(m,n,e) \neq\bot \vee \mathit{Action}(m,n,e) \neq\bot\} \\ d = \{e \mid \alpha(e) = \top\} \qquad d' = (d \cup \mathit{Deferred}(m,n)) - t \\ |q_1| \subseteq d' \qquad e \notin d' \qquad \sigma' = \sigma[\mathbf{msg} := e][\mathbf{arg} := v] \\ s = \begin{array}{l} \mathbf{if}\ \mathit{Pop}(m,n,\alpha,e) \vee \mathit{Step}(m,n,e) \neq\bot \\ \mathbf{then}\ \mathit{Exit}(m,n) \\ \mathbf{else\ skip} \end{array} \end{array}}{M \longrightarrow M[id := ((n,\alpha)\cdot\gamma, \sigma', s; \overline{\mathbf{raise}}\ (e,v), q_1\cdot q_2)]}\ (\textsc{Dequeue})$$

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, \overline{\mathbf{raise}}\ (e,v), q) \\ m = \mathit{Name}(id) \qquad \mathit{Step}(m,n,e) = n' \end{array}}{M \longrightarrow M[id := ((n',\alpha)\cdot\gamma, \sigma, \mathit{Entry}(m,n'), q)]}\ (\textsc{Step})$$

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, \overline{\mathbf{raise}}\ (e,v), q) \\ m = \mathit{Name}(id) \qquad \mathit{Trans}(m,n,e) =\bot \\ (\alpha(e) = a \wedge \mathit{Action}(m,n,e) =\bot) \vee \mathit{Action}(m,n,e) = a \\ a \notin \{\bot,\top\} \end{array}}{M \longrightarrow M[id := ((n,\alpha)\cdot\gamma, \sigma, \mathit{Stmt}(m,a), q)]}\ (\textsc{Action})$$

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, \overline{\mathbf{raise}}\ (e,v), q) \\ m = \mathit{Name}(id) \qquad \mathit{Call}(m,n,e) = n' \\ \alpha' = \lambda e. \begin{array}{l} \mathbf{if}\ (\mathit{Trans}(m,n,e) \neq\bot)\ \mathbf{then}\ \bot \\ \mathbf{else\ if}\ (\mathit{Action}(m,n,e) \neq\bot)\ \mathbf{then}\ \mathit{Action}(m,n,e) \\ \mathbf{else\ if}\ (e \in \mathit{Deferred}(m,n))\ \mathbf{then}\ \top \\ \mathbf{else}\ \alpha(e) \end{array} \end{array}}{M \longrightarrow M[id := ((n',\alpha')\cdot(n,\alpha)\cdot\gamma, \sigma, \mathit{Entry}(m,n'), q)]}\ (\textsc{Call})$$

$$\frac{\begin{array}{c} M[id] = ((n,\alpha)\cdot\gamma, \sigma, \overline{\mathbf{raise}}\ (e,v), q) \\ m = \mathit{Name}(id) \qquad \mathit{Pop}(m,n,\alpha,e) \end{array}}{M \longrightarrow M[id := (\gamma, \sigma, \mathbf{raise}\ (e,v), q)]}\ (\textsc{Pop1})$$

$$\frac{M[id] = ((n,\alpha)\cdot\gamma, \sigma, \overline{\mathbf{return}}, q) \qquad m = \mathit{Name}(id)}{M \longrightarrow M[id := (\gamma, \sigma, \mathbf{skip}, q)]}\ (\textsc{Pop2})$$

Figure 5: Operational semantics: event handling

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

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

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

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

Figure 6: Operational semantics: error transitions

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.

Figure 5 gives the rules for how events are generated and processed. These rules use $\overline{\mathbf{raise}}$ and $\overline{\mathbf{return}}$, which are dynamic instances of **raise** and **return** statements respectively. The computation terminates either normally via completion of all statements in the entry statement, execution of **leave** to jump control to the end of the entry function, 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 raising the retrieved 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 states is 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).

Once an event is raised, using either dequeuing or a raise statement, it is handled using one of the three transition rules STEP, ACTION or CALL. The STEP transition results in leaving the current state n and entering a target state n'. The ACTION transition picks an appropriate action a either from $\mathit{Action}(m, n, e)$ or from the partial map α on the call stack, with the caveat that an action bound on the current state using $\mathit{Action}(m, n, e)$ overrides the action inherited in the call stack using α. Once a suitable action α is picked, the statement $\mathit{Stmt}(m, a)$ is executed. Also, if $\mathit{Step}(m, n, e)$ or $\mathit{Call}(m, n, e)$ is defined, it takes higher priority over actions. The CALL transition computes new values for the map α' in terms of the existing value of the map α on the top of the stack and the set of transitions and actions defined on the current state n. The map $\alpha'(e)$ is defined as follows: if a transition is defined for e then it is bound to $\bot$, otherwise if the event e is bound to an action in n then that binding is used, otherwise if event e is deferred in n then it is mapped to $\top$, and all the other events are mapped to the old value $\alpha(e)$. As a result of the transition, the machine enters the target state n' by pushing the pair (n', α') on the stack.

If these transition rules are not applicable due to the unavailability of a suitable transition, then the top most state on the machine stack is popped via the rules POP1 and POP2 to allow the next state to continue processing. These rules use the predicate $\mathit{Pop}(m, n, \alpha, e)$ to represent the condition under which a state is popped.

$$\begin{array}{rl} \mathit{Pop}(m,n,\alpha,e) = & \mathit{Step}(m,n,e) =\bot \wedge \\ & \mathit{Call}(m,n,e) =\bot \wedge \\ & \mathit{Action}(m,n,e) =\bot \wedge \\ & \alpha(e) \in \{\bot,\top\} \end{array}$$

If rule POP1 executes, the next state must process the unhandled event; if rule POP2 executes, the next state dequeues a new event from the input queue.

The exit function $Exit(m, n)$ of a state n in machine m is executed either when a step transition out of s is taken or s is popped. When an event e is raised or dequeued, the available transitions in s are examined to determined whether $Exit(m, n)$ needs to be executed, and if so, the code statement $Exit(m, n)$ is inserted into the state. $Exit(m, n)$ is always executed if the entry function terminates with a **return**. The rules in Figure 5 assume that $Exit(m, n)$ itself does not contain any explicit raise or return; however, our implementation allows that.

Figure 6 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 $\perp$ (rule SEND-FAIL1), (3) by executing a statement $\mathbf{send}(r_1, e, r_2)$ with r_1 evaluating to some id', but with $M[id'] = \perp$, thereby attempting to send to an uninitialized or deleted machine (rule SEND-FAIL1), and (4) The stack becomes empty after a pop (rule POP-FAIL). In Section 5, we show how to detect all these 4 types of errors using systematic testing.

Other features. We conclude the description of the core language and semantics with two additional features: (1) foreign functions, and (2) call statements. Both these features are very important to write real-world asynchronous code, but their semantics is standard. Consequently, we describe them informally below.

To interact with external 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 give a P body to a foreign function. The body has to be erasable, i.e. uses only ghost variables and expressions.

The P language also has a call statement of the form **call** n', where n' is a state. This can be used to transition to the target state n' by pushing n' on the call stack, much like a call transition. Unlike a call transition, the call statement requires saving a continuation associated with the caller on the stack, so that execution can resume and complete the remaining statements when the callee state is popped.

3.2 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 two liveness properties that must be satisfied by a P program. We describe these two properties below in terms of a precise description of those infinite executions which *violate* these properties; we use linear temporal logic [18] for our specifications. Our specifications use the following predicates over events that occur during an execution:

1. $en(m)$ holds iff machine m is enabled, i.e., m can take a step.
2. $sched(m)$ holds iff machine m is enabled and takes a step.
3. $enq(m, e, m')$ holds iff machine m enqueues an event e into machine m'.
4. $deq(m', e)$ holds iff machine m' dequeues an event e.

The first property specifies that a machine cannot execute indefinitely without getting disabled. In particular, a machine should not get into a cycle of private operations, causing it to loop forever. The set of erroneous executions is given by

$$\exists m.\ \Diamond\Box(sched(m)).$$

The second property specifies that events are not deferred excessively; the goal is to prevent events from being always deferred, thus never processed. We first define the notion of *fairly scheduling* a machine m.

$$fair(m) \triangleq \Box\Diamond(en(m) \Rightarrow sched(m))$$

An erroneous execution is one in which every machine is fairly scheduled and an event is enqueued which is never subsequently dequeued. The set of all erroneous executions is given by

$$\forall m.\ fair(m) \wedge \exists m, e, m'.\ \Diamond(enq(m, e, m') \wedge \Box\neg deq(m', e)).$$

Our experience with real drivers suggests that in some cases, this property may be too strong. For instance, in a system with prioritized events, enough high priority events from the environment may postpone forever the processing of lower priority events. We allow programmers to indicate this expectation in a state s by annotating s with a list of postponed events. Our refinement of the second liveness specification uses the predicate $ppn(m, e)$ that holds whenever m is in a state whose list of postponed events contains e. The smaller set of erroneous executions is given by

$$\forall m.\ fair(m) \wedge \exists m, e, m'.\ \ \Diamond(enq(m, e, m') \wedge \Box\neg deq(m', e)) \wedge \Diamond\Box\neg ppn(m', e).$$

3.3 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) identifiers for machines, state names, events, and variables are unique, (2) statements inside real machines are deterministic, and (3) ghost machines, ghost variables, and ghost events can be *erased* during compilation and execution.

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 6 cannot be detected by our type checker. Instead, we use state-space exploration techniques (described in Section 5) to check for these errors statically.

4. Execution

This section explains how we generate code from a P program, so that the generated code can run as a Windows device driver. Execution requires a host driver framework; our current implementation uses Windows Kernel Mode Driver Framework (KMDF). The complete driver, which runs inside Windows, has four components:

1. The *generated code* is a C file which is produced by the P compiler from a state machine description.
2. The *runtime* is a library that interacts with the generated code and provides utilities for synchronization and management of state, execution, and memory.
3. The *interface 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. The code generated from a P program comprises a collection of indexed and statically-allocated data structures that are examined by the runtime when it executes the operational semantics of the program. The set of names of events is compiled to a C enumeration, thus giving a globally-known unique index to each event. Similarly, names of machine types and names of local variables and states in each machine are also compiled to C enumerations. At the top level, there is a driver structure that contains pointers to an array of events and an array of machine types; these arrays are indexed by the corresponding enumerations for events and machine types respectively. Each entry in the machine array contains pointers to an array of variables and an array of states, each indexed by the corresponding enumeration. Each entry in the state array contains a table of outgoing transitions, a table of deferred events, a table of installed actions, and pointers for entry and exit functions. In addition to these data structures, the compiler also generates C code for the bodies of entry and exit functions of all states and all actions declared in the program.

When P is compiled to C, the state of a machine is wrapped into an object of type `StateMachineContext`. This object contains data structures to represent the state of the machine, as described in the formal operational semantics in Section 3. In addition, it also contains a `void*` pointer to external memory used only by the foreign functions and interface code.

To make the P compiler work for another driver framework or another operating system, the runtime and foreign code needs to be reimplemented appropriately but the generated code does not need to change.

Runtime. The runtime of P provides functionality for all operations required for executing the operational semantics of a P program, such as creating a machine, enqueueing an event into a machine, running a state machine, maintaining the call stack and available event handlers of each machine, etc. This functionality is mostly private to the runtime with a few exceptions for the benefit of interface code, which can interact with the P runtime using three APIs: create a new state machine using `SMCreateMachine`, queue an event into a machine using `SMAddEvent`, or request a pointer to the external memory associated with a machine using `SMGetContext`.

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 deferred procedure call, the driver uses the calling thread to process the event and run to completion. Multiple such threads could be executing inside the runtime at any time; each dynamic instance of a state machine is protected by its own lock for safe synchronization.

Interface code. The interface 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. In our experience, the interface code is generic enough so that it can be automatically generated for a particular class of drivers.

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 external memory that can be used by the programmer to persist some information as part of the state of calling machine. The foreign functions are assumed to terminate and to limit any side effect to the provided memory. Unlike the interface code which is generic, foreign functions are typically driver-specific and consequently need to be specified by the programmer.

4.1 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 with one driver machine and four ghost machines. The driver machine has 15 states and 23 transitions, and each ghost machines has approximately four 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 to 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. Systematic testing

P is designed to enable systematic testing of programs written in it. There are two kinds of nondeterminism in the semantics of P programs—explicit nondeterministic choice in the ghost machines and implicit nondeterministic choice of which machine to schedule next. Systematic testing of a P program is accomplished by interpreting its operational semantics (closed using ghost machines) in the explicit-state model checker Zing [2]. All aspects of the operational semantics of the program are interpreted including the code statements labeling the states and actions of a P machine. The model checker takes care of systematically enumerating all implicit scheduling and explicit modeling choices in the program and checks for the possible errors (see Figure 6) 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. We leave verification of the liveness checks in P programs for future work.

Machines in P programs communicate via message-passing; there are only three operations at which a communication between two machines occurs—creating a machine, sending an event, and receiving an event. It suffices to introduce context-switches after only these operations; since a private operation in a machine commutes with operations of other machines, a context switch after is redundant. In other words, if an error 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 optimization is an example of atomicity reduction [7]. Furthermore, it can be shown that a receive operation is a right mover [15]; therefore, a context switch after a receive operation can also be eliminated and it suffices to introduce context switches only after a machine is created or an event is sent.

Delay-bounded scheduling. Unfortunately, even the coarse-grained context-switching described above is not sufficient to prevent the state space explosion problem. If there are k machines enabled at each interleaving point, the number of possible schedules for runs with n context switches is k^n. Therefore, systematic exploration of long schedules becomes prohibitively expensive. Consequently, we have designed a delaying scheduler to scale our exploration to large P programs with long executions.

Intuitively, our delaying 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. 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. 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, our delaying 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.

The scheduler always chooses to schedule the machine whose identifier is at the top of S. The scheduled machine executes until it reaches a scheduling point (which is a send or the creation of a machine) at which point the scheduler again provides the next machine to be scheduled. The stack S and the delay score is updated in response to events happening in the execution as follows:

- If m is scheduled and m sends an event to machine m' and $m' \notin S$, m' is pushed on S.
- If m is scheduled and m creates a new machine m', then m' is pushed on S.
- If m is delayed, m is moved from top of S to the bottom of S, and the delay score is incremented 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.

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.

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 7 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 7 within which bugs were found), and as we increase the delay bound we eventually explore all states within a delay bound of around 12.

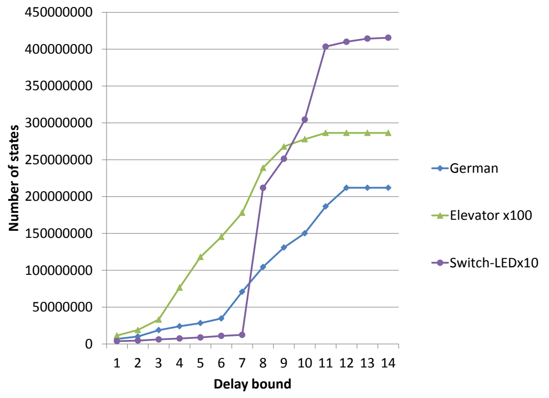


Figure 7: States explored with increasing delay bound.

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 8: 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 8 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 [4], Lustre [10] and Signal [3] 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 [10] 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 [4] 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 [11] 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 [12]) 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 [13] and process calculi, such as CSP [14] and CCS [16], and Join Calculus [8], 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 [5] 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.

Acknowledgments

We would like to acknowledge the contribution of Randy Aull, architect of the USB device driver stack in Windows 8, to the design of P. We would also like to thank him for his enthusiastic adoption of the P programming model. Damien Zufferey would like to acknowledge the support of the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

References

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

[2] T. Andrews, S. Qadeer, S. K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *CAV: International Conference on Computer Aided Verification*, pages 484–487, 2004.

[3] 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.

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

[5] 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.

[6] M. Emmi, S. Qadeer, and Z. Rakamaric. Delay-bounded scheduling. In *POPL: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 411–422, 2011.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349, 2003.

[8] 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.

[9] P. Godefroid. Model checking for programming languages using Verisoft. In *POPL: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, 1997.

[10] 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.

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

[12] 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.

[13] 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.

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

[15] R. J. Lipton. Reduction: a new method of proving properties of systems of processes. In *POPL: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 78–86, 1975.

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

[17] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI : USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, 2008.

[18] A. Pnueli. The temporal logic of programs. In *FOCS: IEEE Symposium on Foundations of Computer Science*, pages 46–67, 1977.

[19] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (3rd edition)*. Prentice Hall, 2009.