E-FRP With Priorities*

Roumen Kaiabachev
Rice University
roumen@rice.edu

Walid Taha
Rice University
taha@rice.edu

Angela Yun Zhu
Rice University
angela.zhu@rice.edu

ABSTRACT

E-FRP is declarative language for programming resource-bounded, event-driven systems. The original high-level semantics of E-FRP requires that each event handler execute atomically. This requirement facilitates reasoning about E-FRP programs, and therefore it is a desirable feature of the language. But the original compilation strategy requires that each handler complete execution before another event can occur. This implementation choice treats all events equally, in that it forces the upper bound on the time needed to respond to any event to be the same. While this is acceptable for many applications, it is often the case that some events are more urgent than others.

In this paper, we show that we can improve the compilation strategy without altering the high-level semantics. With this new compilation strategy, we give the programmer more control over responsiveness without taking away the ability to reason about programs at a high level. The programmer controls responsiveness by declaring priorities for events, and the compilation strategy produces code that uses preemption to enforce these priorities. We show that the compilation strategy enjoys the same properties as the original strategy, with the only change being that the programmer reasons modulo permutations on the order of event arrivals.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Functional Programming

General Terms

Design, Languages, Reliability

Keywords

Resource-Aware Programming, Event-Driven Programming

*This work was supported by NSF SoD award 0439017 "Synthesizing Device Drivers".

EMSOFT'07, September 30–October 3, 2007, Salzburg, Austria.

1. INTRODUCTION

Reactive systems are ones that continually respond to an environment. Functional Reactive Programming (FRP) [14, 29] is a declarative programming paradigm based on time-varying reactive values (*behaviors*) and timed discrete *events*. An FRP *program* is a set of mutually recursive behaviors and event definitions. FRP has been used successfully for programming a variety of reactive systems in the domain of interactive computer animation [6], computer vision [22], robotics [20], and control systems.

Real-time systems are reactive software systems that are required to respond to an environment in a bounded amount of time [28]. In addition, essentially all real-time systems need to execute using a fixed amount of memory because physical resources on their host platforms are constrained. FRP is implemented as an embedded language in Haskell [21]. A language embedded in a general-purpose language such as Haskell cannot provide real-time guarantees, and to address this problem, focus turned to a real-time subset in which one global clock is used to synchronously update the whole program state [30]. The global clock was then generalized to arbitrary events in a stand-alone language called E-FRP [31]. Any E-FRP program guarantees (1) response to every event by the execution of its handler, (2) complete execution of each handler, and (3) execution in bounded space and time. E-FRP has been used for programming event-driven reactive systems such as interrupt-driven micro-controllers, which are otherwise typically programmed in C or assembly language. The E-FRP compiler generates resource-bounded C code that is a group of event handlers in which each handler is responsible for one event source.

1.1 Problem

The original high-level semantics of E-FRP requires that each event handler execute atomically. This requirement facilitates reasoning about E-FRP programs, and therefore it is a desirable feature of the language. But the original compilation strategy requires that each handler complete execution before another event can occur. This implementation choice treats all events equally in that it forces the upper bound on the time needed to respond to any event to be the same. While this is acceptable for many applications, it is often the case that some events are more urgent than others.

1.2 Contributions

In this paper, we show that we can improve the compilation strategy for E-FRP while preserving the original high-level semantics. This new compilation strategy, which we

call P-FRP, gives the programmer more control over responsiveness without compromising any of the high-level reasoning principles. The programmer controls responsiveness by declaring priorities for events (Section 2). To model prioritized interrupts in the target platform, we refine the original big-step semantics used for the target language (called SimpleC) into a small-step semantics, and then we augment it with explicit notions of interrupt and context switch (Section 3). We develop a compilation strategy that produces code that uses preemption to enforce these priorities (Section 4). Preemption is implemented using a roll-back strategy that is comparable to a simple form of software transaction [26, 11, 23]. We show that the compilation strategy enjoys the same properties as the original strategy modulo permutations on the order of event arrivals (Section 5). Finally, we formalize the sense in which the programmer has more control over responsiveness by giving analytic expressions for upper bounds under reasonable conditions for handlers with and without priorities, and we validate these bounds experimentally (Section 6).

Due to space limitations, the paper only shows the P-FRP compilation strategy and technical results. The original E-FRP semantics and compilation function, and formal proofs of our technical results, can be found in an extended version of the paper available online [15].

2. P-FRP SYNTAX AND SEMANTICS

We use the following notational conventions in the rest of the paper:

Notation

- $\langle f_j \rangle^{j \in \{1 \ldots n\}}$ denotes the sequence $\langle f_1, f_2, \ldots, f_n \rangle$. We will occasionally omit the superscript $j \in \{1 \ldots n\}$ and write $\langle f_j \rangle$ when the range of j is clear from context.
- $\{f_j\}^{j \in \{1 \ldots n\}}$ or $\{f_j\}$ denotes the set $\{f_1, f_2, \ldots, f_n\}$.
- $x_1 :: \langle x_2, \ldots, x_n \rangle$ denotes the sequence $\langle x_1, x_2, \ldots, x_n \rangle$.
- $A \# A'$ denotes the concatenation of the sequences A and A'. We write $A \uplus B$ for $A \cup B$ when we require that $A \cap B = \emptyset$. We also write $A - B$ for set difference.
- $\texttt{prim}(f, \langle c_i \rangle) \equiv c$ denotes the application of a primitive function f on arguments $\langle c_i \rangle$ resulting in c.
- With the exception of $\texttt{prim}(f, \langle c_i \rangle) \equiv c$, $\equiv$ denotes that two sets of syntax elements are the same (such as in $H \equiv \{I \Rightarrow d\ \varphi\}$). This is different from $=$ used in syntax, which is assignment in the language represented by the grammar.

The syntax of P-FRP is the same as E-FRP, although we add an environment L to allow the programmer to declare a priority for each event:

Variable	x	$\in$	$\mathcal{X}$
Constant	c	$\in$	$\mathbb{N}$
Event name	I	$\in$	$\mathcal{I}$
Function	f	::=	`||` | `&&` | `!` | `+` | `-` | `*` | `/` | `>` | `>=` | `>` | `<=` | `==` | `!=` | `if`
Passive behaviors	d	::=	$x \mid c \mid f\ \langle d_i \rangle$
Active behaviors	r	::=	$\mathsf{init}\ c\ \mathsf{in}\ H$
Behaviors	b	::=	$d \mid r$
Phases	$\varphi \in \Phi$	::=	$\epsilon \mid \mathsf{later}$
Event handlers	H	::=	$\{I_i \Rightarrow d_i\ \varphi_i\}$
Programs	P	::=	$\{x_i = b_i\}$
Priority level	l	::=	$n \in \{l_{min}, \ldots, l_{max}\} \in \mathsf{Nat}$
Environment	E	::=	$\{I_i \mapsto l_i\}$

In E-FRP, *passive behavior* expressions can be variables, constants, or function applications to other passive behaviors. The terminals x and c are the syntactic categories of variables and constants, respectively, and f is the syntactic category for function application. The only *active behavior* r has the form $\mathsf{init}\ c\ \mathsf{in}\ \{I_i \Rightarrow d_i\ \varphi_i\}$ where c is the initial value, and the part in parentheses is a set of *event handlers*. When an event I_i occurs, the behavior value c changes to the value of d_i computed at the time of the event.

E-FRP programs are evaluated in two *phases* w.r.t. to the occurrence of the event, and the computation of d_i is associated with either phase. Depending upon whether φ_i is ϵ or later, the value of r is changed either immediately (in the first phase) or after all other immediate updates triggered by the event (in the second phase). An E-FRP program P is a set of mutually recursive behavior definitions: the value of a behavior might depend upon values computed for other behaviors.

In P-FRP, the programmer explicitly declares event priorities by mapping each event to its constant priority, and priorities are selected from a fixed range of integer values. As we will see, the priorities (continue to) play no role in the high-level, big-step semantics of P-FRP.

As a simple example to illustrate the syntax and the informal semantics, consider the following program:

$$
\begin{array}{l}
I_1 \rightarrow 1, I_2 \rightarrow 2 \\
x = \mathsf{init}\ 1\ \mathsf{in}\ \{I_1\ \Rightarrow\ x + y\}, \\
y = \mathsf{init}\ 1\ \mathsf{in}\ \{I_1\ \Rightarrow\ x - y\ \mathsf{later}, I_2\ \Rightarrow\ 1\}
\end{array}
$$

This program defines two behaviors x and y triggered by an event I_1 of priority 1 and an event I_2 of priority 2. When I_1 occurs, the value of x is computed immediately in the first phase. The later annotation indicates that the value of y is not computed until after all other behaviors triggered by I_1 have been computed. The values of the behaviors after several occurrences of I_1 are shown below. The numbers in bold are final (second-phase) values for each behavior on I_1. The fourth occurrence of I_1 is followed by I_2, which resets y.

	(init)	I_1	I_1	I_1	I_1	I_2	I_1
x	**1**	2 **2**	3 **3**	5 **5**	8 **8**	8 **8**	9 **9**
y	**1**	1 **1**	1 **2**	2 **3**	3 **5**	1 **1**	1 **8**

The big-step semantics of P-FRP is the same as that of E-FRP. Event priorities are not part of the semantics because all events are executed atomically. Figure 1 defines four judgments that formalize the notions of updating and computing program behaviors:

- $P \vdash b \xrightarrow{I} c$: "on event I, behavior b yields c."
- $P \vdash b \xrightarrow{I} b'$: "on event I, behavior b is updated to b'"
- $P \vdash b \xrightarrow{I} c; b'$: "on event I, behavior b yields c, and is updated to b'".
- $P \xrightarrow{I} S; P'$: "on event I, program P yields store S and is updated to P'."

When an event I occurs, a program in P-FRP is executed by updating program behaviors. Updating a program behavior requires, first, an evaluation of the behaviors it depends

$\boxed{P \vdash b \xrightarrow{I} c}$

$$\frac{P \vdash b \xrightarrow{I} c}{P \uplus \{x = b\} \vdash x \xrightarrow{I} c} \qquad \frac{}{P \vdash c \xrightarrow{I} c}$$

$$\frac{\left\{P \vdash d_i \xrightarrow{I} c_i\right\} \quad \mathtt{prim}(f, \langle c_i \rangle) \equiv c}{P \vdash f\langle d_i \rangle \xrightarrow{I} c}$$

$$\frac{b \equiv \mathsf{init}\ c\ \mathsf{in}\ \{I \Rightarrow d\ \epsilon\} \uplus H \quad P \uplus \{x = b\} \vdash d[x := c] \xrightarrow{I} c'}{P \uplus \{x = b\} \vdash b \xrightarrow{I} c'}$$

$$\frac{\forall H'.\forall d.H \not\equiv \{I \Rightarrow d\ \epsilon\} \uplus H'}{P \vdash \mathsf{init}\ c\ \mathsf{in}\ H \xrightarrow{I} c}$$

$\boxed{P \vdash b \xrightarrow{I} b'}$

$$\frac{}{P \vdash d \xrightarrow{I} d}$$

$$\frac{b \equiv \mathsf{init}\ c\ \mathsf{in}\ H \quad b' \equiv \mathsf{init}\ c'\ \mathsf{in}\ H \quad H \equiv \{I \Rightarrow d\ \varphi\} \uplus H' \quad P \uplus \{x = b\} \vdash d[x := c] \xrightarrow{I} c'}{P \uplus \{x = b\} \vdash b \xrightarrow{I} b'}$$

$$\frac{\forall H'.\forall d.H \not\equiv \{I \Rightarrow d\ \varphi\} \uplus H'}{P \vdash \mathsf{init}\ c\ \mathsf{in}\ H \xrightarrow{I} \mathsf{init}\ c\ \mathsf{in}\ H}$$

$\boxed{P \vdash b \xrightarrow{I} c; b'}$

$$\frac{P \vdash b \xrightarrow{I} c \quad P \vdash b \xrightarrow{I} b'}{P \vdash b \xrightarrow{I} c; b'}$$

$\boxed{P \xrightarrow{I} S; P'}$

$$\frac{\left\{\{x_i = b_i\}^{i \in K} \vdash b_j \xrightarrow{I} c_j; {b'}_j\right\}^{j \in K}}{\{x_i = b_i\}^{i \in K} \xrightarrow{I} \{x_i \mapsto c_i\}^{i \in K}; \{x_i = {b'}_i\}^{i \in K}}$$

1: Big-step Operational Semantics of P-FRP

upon. On an event, an P-FRP program yields a store, which is the state after the first phase, and an updated program. The updated program contains the final state in the init statements of its reactive behaviors.

A store S maps variables to values: $S ::= \{x_i \mapsto c_i\}$.

The first rule in the judgment $P \vdash b \xrightarrow{I} c$ states that a behavior x yields a ground value after evaluation. The next two rules state how to evaluate a passive behavior that is a constant or a function. The fourth rule states how to evaluate an active behavior: its current value is substituted in the handler body for I, and the body is evaluated to yield a constant. Finally, a behavior that is not triggered by I or whose response is computed in the second phase yields its current value.

The first rule in the judgment $P \vdash b \xrightarrow{I} b'$ states that a passive behavior updates to itself. The next rule states that a behavior updates to a new behavior whose value is produced by evaluating its handler for I after the pre-update value of the behavior is substituted in the handler body. Finally, a behavior that is not triggered by I evaluates to itself.

The rule in the judgment $P \vdash b \xrightarrow{I} c; b'$ is a shorthand for $P \vdash b \xrightarrow{I} c$ and $P \vdash b \xrightarrow{I} b'$. The rule in the judgment $P \xrightarrow{I} S; P'$ states that a program P is updated on I by updating each behavior in the program on I.

The trace of the simple example introduced above illustrates a key point about the P-FRP semantics: when an event I_1 occurs, behavior x is evaluated in the first phase. Evaluating x requires evaluating y before it changes on I_1. Since y evaluates to its current value, 1, x evaluates to $1+1 = 2$. Now behavior x is updated to $x = \mathsf{init}\ 2\ \mathsf{in}\ \{\ I\ \Rightarrow\ x+y\ \}$. Next, behavior y is evaluated in the second phase to $2-1 = 1$ using the new value of x, 2, which was computed in the first phase. Then behavior y is updated to what it was before: $y = \mathsf{init}\ 1\ \mathsf{in}\ \{\ I_1\ \Rightarrow\ x - y\ \mathsf{later}\}$. The big-step semantics of P-FRP treats I_2 as an event of priority equal to I_1. I_2 resets y to 1, and the next execution of I_1 uses this value.

3. PREEMPTABLE SIMPLEC

As a model of the hardware of the target embedded platform, we use a calculus called SimpleC. Terms in this calculus have a direct mapping to C code. The syntax of SimpleC is as follows:

Computations	$d ::= x \mid c \mid f\ \langle d_i \rangle$
Statements	$A ::= \langle x_i := d_i \rangle \mid \mathsf{off} \mid \mathsf{on}$
Programs	$Q ::= \{(I_i, A_i)\}$

A SimpleC program Q is a collection of event handler definitions of the form (I, A), where I is an event and also the handler name. The body of the handler is divided into two consecutive parts, which are the first phase and the second phase statements (in the original E-FRP [31], Q is generated by the compilation function from the two phases in the source program as $Q ::= \{(I_i, A_i, A'_i\}$ to explicitly separate the phases). The statements include primitives (off and on) that enable or disable all interrupts (which are the only change from the original SimpleC), and also assignments.

Before presenting the formal semantics for this language, we consider a simple example to illustrate both the syntax and the essence of the compilation strategy. The SimpleC code corresponding to the simple example is as follows:

```
int x, _x, y, _y = 1;   int xt, _xt, yt, _yt;
I1, {off; xt = x; _xt = _x; yt = y; _yt = _y; on;
\*1*\   xt = (_xt + yt); _yt = (xt − yt);
\*2*\   _xt = xt; yt = _yt;
  off; x = xt; _x = _xt; y = yt; _y = _yt; on; }
I2, {   off; yt = y; _yt = _y; on;
\*1*\     yt = 1;     \*2*\  _yt = yt;
    off; y = yt; _y = _yt; on; }
```

The code is a group of event handlers. Each handler is a C function that is executed when an event occurs and consists of two sequences of assignments, one for each of the two phases. In addition, there is a preamble for copying values to temporaries and a postamble to commit the temporary values. In particular, for each behavior we have committed values (x, y), first-phase values (_x, _y), and temporary values for each of these (xt, yt, _xt, _yt).

SimpleC was originally defined using a big-step semantics [31], but an equivalent small-step semantics [15] can be defined , which makes it easier to model preemption. The

semantics presented here uses the following elements:

Master Bit	m	$::= \mathsf{on} \mid \mathsf{off}$
Interrupt Context	$\triangle$	$::= \top \mid \bot$
Stack	σ	$::= \mathsf{nil} \mid (I, A, \triangle) :: \sigma$
Queue	q	$::= \mathsf{nil} \mid I :: q$
Step	W	$::= I \mid \diamond$

We will model how lower-priority events that occur while a higher-priority event is handled are stored in a *queue*, sorted by priorities. Higher-priority events interrupt lower-priority ones when the CPU's *master bit* m is enabled ($m \equiv \mathsf{en}$). Otherwise, when the master bit is disabled ($m \equiv \mathsf{dis}$) interrupts are globally turned off and higher-priority events are queued.

A program stack σ contains event-handler statements with the active handler on top of the stack. The stack also contains an *interrupt context* flag ($\top$ or $\bot$) that indicates whether the active event handler has been interrupted. When an event occurs that is of higher priority than the currently handled one, its handler is placed on top of the stack, and the flag for the interrupted event is toggled. The value of the flag determines whether the interrupted handler is later re-executed from its beginning just like a software transaction.

A *step* denotes whether the program has received an interrupt or has made progress on a computation. Progress is looking up a variable in the environment, evaluating a function argument, applying a function, or updating the store with the results of an assignment. A *priority environment* maps interrupts to their priorities. The notation $x_1 :: \langle x_2, \ldots, x_n \rangle$ denotes the sequence $\langle x_1, x_2, \ldots, x_n \rangle$, and we use $\equiv$ to denote syntactic equivalence.

To model pending interrupts, in the judgment on program states we use the function insert, which inserts events into the queue and keeps the queue sorted by priority. If two events have the same priority, it sorts them by time of occurrence, with the older event at the front of the queue.

$$
\begin{array}{lcl}
\mathsf{insert}(I, \mathsf{nil}) & \equiv & I :: \mathsf{nil} \\
\mathsf{insert}(I, U :: q) & \equiv & I :: U :: q \text{ if } E(I) > E(U) \\
\mathsf{insert}(I, U :: q) & \equiv & U :: \mathsf{insert}(I, q) \text{ if } E(I) \leq E(U) \\
\mathsf{insert}(I, I :: q) & \equiv & I :: q
\end{array}
$$

The function top is also used in the judgment on program states to peek at the queue's top and return the priority of the first element. If the queue is empty, top returns the lowest possible priority l_{min}.

$$\mathsf{top}(q) \equiv l_{min} \text{ if } q \equiv \mathsf{nil} \quad \mathsf{top}(q) \equiv E(I) \text{ if } q \equiv I :: q'$$

The original E-FRP compilation strategy assumes that no other interrupts will occur while statements are processed. Under this assumption, the execution of the handler is atomic, and E-FRP ignores other events at any step of processing a handler. In reality, if code runs in an environment where events are prioritized, it will be preempted.

We present an extended semantics that models preemption. In particular, our design models handling of interrupts with priorities in the Windows and Linux kernels ([15, 27, 25, 3, 24]) and borrows ideas for atomic handler execution from *software transactions* [10, 23, 5], a concurrency primitive for atomicity that disallows interleaved computation while ensuring fairness (we return to software transactions in the related works in Section 7).

The trace below for the SimpleC code for our simple example shows a preemption that executes like a software transaction:

	(init)	I_1	I_1	I_1	I_1	I_2	${I_1}^R$
x	**1**	2 **2**	3 **3**	5 **5**	8 /	5 **5**	6 **6**
y	**1**	1 **1**	1 **2**	2 **3**	3 /	1 **1**	1 **5**

The fourth occurrence of I_1 is interrupted by I_2 at the end of the first phase. The computed values in this phase are discarded, I_2 executes, and I_1's handler is restarted. In particular, when the fourth occurrence of I_1 is interrupted by I_2, y is reset to 1, while any computations on x during the interrupted handler are discarded. When I_1's handler is restarted, x is 5, as before the fourth occurrence of I_1, and the new value for x computed is 6. The new value for x is used in the second phase to compute the value of y, $5 = 6 - 1$.

In the SimpleC small-step semantics with priorities and restarting, we have the judgments below:

- $S \vdash d \longmapsto d'$: "under store S, d evaluates to d' in one step."
- $(A, S, m) \longmapsto (A', S', m')$: "executing one step of assignment sequence A produces assignment sequence A', updates store S to S', and leaves all interrupts in state m'."
- $(S, Q, m, \sigma, q) \xmapsto{W} (S', m', \sigma', q')$: " one step of the execution of program Q updates store S to S', changes the master bit from m to m', updates the pending event queue from q to q', and updates the program stack σ to σ'."

We first define the most basic step of our execution model. The judgment $S \vdash d \longmapsto d'$ states that a variable is evaluated by looking up its value in the environment, and a function is then applied after evaluation of its arguments.

$$\frac{}{S \uplus \{x \mapsto c\} \vdash x \longmapsto c} \qquad \frac{\mathtt{prim}(f, \langle c_0, \ldots, c_n \rangle) \equiv c}{S \vdash f\langle c_0, \ldots, c_n \rangle \longmapsto c}$$

$$\frac{S \vdash d_i \longmapsto d_i'}{S \vdash f\langle c_0, \ldots, c_{i-1}, d_i, \ldots, d_n \rangle \longmapsto f\langle c_0, \ldots, c_{i-1}, d_i', \ldots, d_n \rangle}$$

The first rule of $(A, S, m) \longmapsto (A', S', m')$ states that an assignment is evaluated in one step by evaluating its computation part one step and updating the assignment. The second rule states that an assignment whose computation part is a ground value is evaluated by updating the store with the ground value and removing the assignment from sequence A. The last two rules toggle the interrupt state.

$$\frac{\{x \mapsto c\} \uplus S \vdash d \longmapsto d'}{(x := d :: A, \{x \mapsto c\} \uplus S, m) \longmapsto (x := d' :: A, \{x \mapsto c\} \uplus S, m)}$$

$$\frac{}{(x := c' :: A, \{x \mapsto c\} \uplus S, m) \longmapsto (A, \{x \mapsto c'\} \uplus S, m)}$$

$$\frac{}{(\mathsf{off} :: A, S, m) \longmapsto (A, S, \mathsf{dis})} \qquad \frac{}{(\mathsf{on} :: A, S, m) \longmapsto (A, S, \mathsf{en})}$$

Next we present the judgment on *program states* $(S, Q, m, \sigma, q) \xmapsto{W} (S', m', \sigma', q')$. Rules (Unh), $(Start)$ and (Pop) are essentially the same as the original SimpleC ([31], [15]) with the difference that $(Start)$ only executes when interrupts are enabled.

$$\frac{}{S \vdash c \longmapsto^0 c} \qquad \frac{S \vdash d \longmapsto d' \quad S \vdash d' \longmapsto^n d''}{S \vdash d \longmapsto^{n+1} d''}$$

$$\frac{}{(\langle\rangle, S, m) \longmapsto^0 (\langle\rangle, S, m)} \qquad \frac{(A, S, m) \longmapsto (A', S', m') \quad (A', S', m') \longmapsto^n (A'', S'', m'')}{(A, S, m) \longmapsto^{n+1} (A'', S'', m'')}$$

$$\frac{}{(S, Q, m, \mathsf{nil}, \mathsf{nil}) \overset{\diamond_0}{\longmapsto} (S, m, \mathsf{nil}, \mathsf{nil})} \qquad \frac{(S, Q, m, \sigma, q) \overset{\diamond}{\longmapsto} (S', m', \sigma', q') \quad (S', Q, m', \sigma', q') \overset{\diamond_n}{\longmapsto} (S'', m'', \sigma'', q'')}{(S, Q, m, \sigma, q) \overset{\diamond_{n+1}}{\longmapsto} (S'', m'', \sigma'', q'')}$$

$$\frac{(S, Q, m, \sigma, q) \overset{I}{\longmapsto} (S', m', \sigma', q') \quad (S', Q, m', \sigma', q') \overset{\diamond_n}{\longmapsto} (S'', m'', \sigma'', q'')}{(S, Q, m, \sigma, q) \overset{I, \diamond_n}{\longmapsto} (S'', m'', \sigma'', q'')}$$

$$\frac{\begin{array}{c}(S, Q, m, \sigma, q) \overset{I_1, \diamond_{k_1}}{\longmapsto} (S', m', \sigma', q') \\ (S', Q, m', \sigma', q') \overset{Z'}{\longmapsto} (S'', m'', \sigma'', q'') \\ Z' \equiv I_2, \diamond_{k_2}, \ldots, I_n, \diamond_{k_n}\end{array}}{\begin{array}{c}(S, Q, m, \sigma, q) \overset{Z}{\longmapsto} (S'', m'', \sigma'', q'') \\ Z \equiv I_1, \diamond_{k_1}, I_2, \diamond_{k_2}, \ldots, I_n, \diamond_{k_n}\end{array}}$$

2: Multiple Steps in the Small-step Operational Semantics of SimpleC With Priorities

$$\frac{I \notin \{I_i\}}{(S, \{(I_i, A_i)\}, m, \sigma, q) \overset{I}{\longmapsto} (S, m, \sigma, q)}\,(Unh)$$

$$\frac{m \equiv \mathsf{en}}{(S, \{(I, A)\} \uplus Q, m, \mathsf{nil}, \mathsf{nil}) \overset{I}{\longmapsto} (S, m, (I, A, \bot), \mathsf{nil})}\,(Start)$$

$$\frac{}{(S, Q, m, (I, \langle\rangle, \triangle) :: \sigma, \mathsf{nil}) \overset{\diamond}{\longmapsto} (S, m, \sigma, \mathsf{nil})}\,(Pop)$$

Rule (*Step*) performs computation on a non-empty handler and checks to see if either the interrupts are disabled or the currently handled event is of the highest priority compared with events in the queue.

Condition for progress on I's handler:
$\mathsf{p} \equiv (m \equiv \mathsf{dis}\ or\ (m \equiv \mathsf{en}\ and\ E(I) \geq \mathsf{top}(q)))$

$$\frac{\mathsf{p} \quad (A, S, m) \longmapsto (A_1, S_1, m_1)}{\begin{array}{c}(S, Q, m, (I, A, \bot) :: \sigma, q) \overset{\diamond}{\longmapsto} \\ (S_1, m_1, (I, A_1, \bot) :: \sigma, q)\end{array}}\,(Step)$$

The next rule (*Restart*) is new and is the most interesting one. This rule re-executes interrupted handlers when a handler has been interrupted. The interrupted handler is popped off the stack, and the original handler for the same event is placed back on the stack.

$$\frac{\mathsf{p} \quad Q \equiv \{(I, A_I)\} \uplus Q' \quad A \not\equiv \langle\rangle}{\begin{array}{c}(S, Q, m, (I, A, \top) :: \sigma, q) \overset{\diamond}{\longmapsto} \\ (S, m, (I, A_I, \bot) :: \sigma, q)\end{array}}\,(Restart)$$

Rules ($Deq1$) and ($Deq2$) model dequeuing for empty handlers:

$$\frac{\sigma \equiv (U, A, \triangle) :: \sigma'}{(S, Q, m, (I, \langle\rangle, \triangle) :: \sigma, U :: q) \overset{\diamond}{\longmapsto} (S, m, \sigma, q)}\,(Deq1)$$

$$\frac{\begin{array}{c}m \equiv \mathsf{en} \quad \sigma \equiv \{\mathsf{nil} \mid (P, A_P, \triangle) :: \sigma'\} \\ Q \equiv \{(U, A)\} \uplus Q'\end{array}}{\begin{array}{c}(S, Q, m, (I, \langle\rangle, \triangle) :: \sigma, U :: q) \overset{\diamond}{\longmapsto} \\ (S, m, (U, A, \bot) :: \sigma, q)\end{array}}\,(Deq2)$$

There are two types of events in the queue. The first type are those whose handlers are on the stack and that occurred while a higher-priority event handler was executing or while interrupts were disabled. The second type are events that have been interrupted but their handlers are still on the stack. In ($Deq1$), a handler is removed from the stack when the stack has a next handler and the handler is for the event at the front of the queue. In ($Deq2$), there is a pending event in the queue with a priority between that of the finished handler and the next handler on the stack. The pending event's handler is placed on the stack, and the event is removed from the front of the queue. Alternatively, if the stack is empty, the handler for the event at the front of the queue is placed on the stack.

Rule ($Deq3$) allows handlers to start for higher-priority events that occur while interrupts are disabled. As soon as interrupts are enabled, the current handler is preempted and a higher-priority handler is pushed onto the stack.

$$\frac{m \equiv \mathsf{en} \quad E(U) > E(I) \quad A' \not\equiv \langle\rangle}{\begin{array}{c}(S, \{(U, A)\} \uplus Q, m, (I, A', \triangle) :: \sigma, U :: q) \overset{\diamond}{\longmapsto} \\ (S, m, (U, A, \bot) :: (I, A', \top) :: \sigma, q)\end{array}}\,(Deq3)$$

The last two rules specify how an interrupt is handled based on its priority and current interrupt priority. It is queued (Enq) when it is of the same or lower priority or when interrupts are disabled. Otherwise, its handler is placed on top of the stack (Int). In the latter case, we indicate that the previous handler was interrupted and place this handler's corresponding event in the queue.

$$\frac{\begin{array}{c}(E(I) \leq E(U)\ or\ (m \equiv \mathsf{dis}\ and\ E(I) > E(U))) \\ \sigma \equiv (U, A, \bot) :: \sigma'\end{array}}{(S, Q, m, \sigma, q) \overset{I}{\longmapsto} (S, m, \sigma, \mathsf{insert}(I, q))}\,(Enq)$$

$$\frac{m \equiv \mathsf{en} \quad E(I) > E(U)}{\begin{array}{c}(S, \{(I, A_I)\} \uplus Q, m, (U, A, \bot) :: \sigma, q) \overset{I}{\longmapsto} \\ (S, m, (I, A_I, \bot) :: (U, A, \top) :: \sigma, \mathsf{insert}(U, q))\end{array}}\,(Int)$$

Taking multiple steps in the semantics consists of executing a sequence of interrupts with none or several computation steps in between, such as the sequence $I_1, \diamond_{k_1}, I_2, \diamond_{k_2}, \ldots, I_n, \diamond_{k_n}$. We define the judgment for modeling taking multiple steps at one time in the semantics of P-FRP in Figure 2.

4. COMPILATION

A P-FRP program is compiled to a set of pairs, which are the same as the input to the SimpleC semantics, in which

$$\frac{FV(d) \cap (\{x\} \uplus \{x_i\}) \equiv \emptyset}{(x := d) < \langle x_i := d_i \rangle}$$

$$\frac{}{\langle\!|\{\}|\!\rangle_I^n = \langle\rangle} \quad \frac{\langle\!|P|\!\rangle_I^n = A \quad (xt := d) < A}{\langle\!|\{x = d\} \uplus P|\!\rangle_I^n = xt := d :: A}$$

$$\frac{\langle\!|P|\!\rangle_I^1 = A \quad (xt := d[x := \_xt]) < A}{\langle\!|\{x = \mathsf{init}\ c\ \mathsf{in}\ \{I \to d\} \uplus H\} \uplus P|\!\rangle_I^1 = xt := d[x := \_xt] :: A}$$

$$\frac{\langle\!|P|\!\rangle_I^1 = A \quad (\_xt := d[x := xt]) < A}{\langle\!|\{x = \mathsf{init}\ c\ \mathsf{in}\ \{I \to d\ \mathsf{later}\} \uplus H\} \uplus P|\!\rangle_I^1 = \_xt := d[x := xt] :: A}$$

$$\frac{}{\langle\!|\{\}|\!\rangle_I^n = \langle\rangle} \quad \frac{\langle\!|P|\!\rangle_I^n = A \quad (xt := d) < A}{\langle\!|\{x = d\} \uplus P|\!\rangle_I^n = xt := d :: A}$$

$$\frac{\langle\!|P|\!\rangle_I^2 = A}{\langle\!|\{x = \mathsf{init}\ c\ \mathsf{in}\ \{I \to d\} \uplus H\} \uplus P|\!\rangle_I^2 = \_xt := xt :: A}$$

$$\frac{\langle\!|P|\!\rangle_I^2 = A}{\langle\!|\{x = \mathsf{init}\ c\ \mathsf{in}\ \{I \to d\ \mathsf{later}\} \uplus H\} \uplus P|\!\rangle_I^2 = xt := \_xt :: A}$$

$$\frac{P \equiv \{x_i = b_i\} \quad \{\langle\!|P|\!\rangle_I^1 = A_I \ \ \langle\!|P|\!\rangle_I^2 = A'_I\}^{I \in I} \quad \bigcup_i FV_i(b_i) \subseteq \{x_i\} \quad x_j \in \mathtt{Updated\_by\_I}(P)}{[\![P]\!] = \{(I, \mathsf{off} :: \langle xt_j = x_j :: \_xt_j = \_x_j\rangle :: \mathsf{on} :: A_I :: A'_I :: \mathsf{off} :: \langle x_j = xt_j :: \_x_j = \_xt_j\rangle :: \mathsf{on}\}^{I \in I}}$$

3: Compilation of P-FRP

each pair consists of an event and a sequence of statements for that event. The compilation function extracts the statements for each phase by searching for behaviors triggered by the event in the P-FRP program. It also checks for circular references of variables during a phase and returns an error if there are some.

To allow for correct restarting of handlers, compilation is extended to generate statements that store variables modified in an event handler into fresh temporary (or *scratch*) variables in the beginning of the handler while interrupts are turned off, and to restore variables from the temporary variables at the end of the handler while interrupts are turned off. We call these the *backup*, *computation* and *restoration* parts. Most importantly, the temporary variables are used throughout the computation part. If the computation part of a handler is interrupted, values in the temporary variables are discarded. A handler does not affect program state until the restoring part.

The compilation rules define how active and passive behaviors in P-FRP compile to SimpleC. For each event, compilation builds an event handler in SimpleC, which scans all P-FRP behaviors for handlers for that event and for each handler found and emits statements to the SimpleC handler.

The rules are the same as the original compilation with two exceptions. First, event handlers update scratch variables corresponding to the original variables, and scratch variables are not used for values that are only read in a handler. In this way, restarting guarantees that a consistent value will always be read. Second, the top-level rule is extended with backup and restore parts.

Figure 3 defines the following:

- $(x := d) < A$: "d does not depend on x or any variable updated in A."
- $\langle\!|P|\!\rangle_I^1 = A$: "A is the first phase of P's event handler for I"
- $\langle\!|P|\!\rangle_I^2 = A$: "A is the second phase of P's event handler for I"
- $[\![P]\!] = Q$: "P compiles to Q"

The set of all variables declaring behaviors dependent on I is defined as the set $\mathtt{Updated\_by\_I}(P)$.

$$\mathtt{Passive}(P) \equiv \{x \mid \{x = d\} \uplus P\}$$
$$\mathtt{Updated\_by\_I}(P) \equiv \{x \mid \{x = \mathsf{init}\ c\ \mathsf{in}\ (\{I \to d\ \varphi\} \uplus H)\} \uplus P\ \cup\ \mathtt{Passive}(P)\}$$

A function FV that computes a set of free variables in a behavior b is defined as follows:

$$FV(x) \equiv \{x\}, \quad FV(c) \equiv \emptyset, \quad FV(f\langle d_i\rangle) \equiv \bigcup_i FV(d_i)$$
$$FV(\mathsf{init}\ c\ \mathsf{in}\ \{I_i \Rightarrow d_i\ \varphi_i\}) \equiv \bigcup_i FV(d_i)$$

The function collects all references to variables in the behavior's handler and excludes the ones referring to the behavior.

The first rule in figure 3 for the judgment $\langle\!|P|\!\rangle_I^1 = A$ states that an empty P-FRP program produces an empty handler for I. The second rule with $n = 1$ states that a passive behavior compiles to equivalent SimpleC. The third rule states that an active behavior executed in the first phase compiles to SimpleC code, in which the value of the behavior is changed in the first phase. The next rule states that an active behavior executed in the second phase compiles to SimpleC, where only a temporary copy of the behavior value is changed in the first phase. The fifth rule, with $n = 1$, states no handler is produced for an unhandled event.

The second rule with $n = 2$, and the sixth rule for the judgment $\langle\!|P|\!\rangle_I^2 = A$, are the same as in the previous judgment. The seventh rule compiles an active behavior executed in the first phase to SimpleC that copies the computed value in the first phase. The fifth rule with $n = 2$ generates SimpleC that updates a behavior value in the second phase. The last rule is the same as in the previous judgment.

In the top-level rule, there is a check that there are no references to undeclared behaviors.

Compilation produces the example SimpleC programs we presented in Section 3.

5. TECHNICAL RESULTS

This section presents the technical results establishing the correctness of the P-FRP compilation strategy.

5.1 Correctness of Compiling

Our proof follows that of Wan et al.'s proof [31]. In particular, after handling any event I, the updated E-FRP program should compile to the same SimpleC as the original E-FRP program. This property holds because E-FRP programs carry all of the relevant state, while the SimpleC

only contains the executable instructions. At the same time, the state embodied in the new E-FRP program must match those produced by executing the resulting SimpleC program. Diagrammatically,

$$\begin{array}{ccc} \text{P-FRP program } P & \xrightarrow{I} & S \times P' \\ \Big\downarrow [\![\,]\!] & & \Big\downarrow \langle \equiv, [\![\,]\!] \rangle \\ \text{SimpleC program } Q & \overset{I,\diamond_n}{\longmapsto} & S \times Q \end{array}$$

One auxiliary notion is needed to state the result. The *state* of a P-FRP program P, written as $\texttt{state}(P)$, is a store defined by:

Definition $\texttt{state}(P) \equiv \{x_i \mapsto \texttt{state}_P(d_i)\} \uplus \{x_j \mapsto \texttt{state}_p(r_j), \_x_j \mapsto \texttt{state}_P(r_j)\}$ where $P \equiv \{x_i = d_i\} \uplus \{x_j = r_j\}$

A `state` function collects the value of each behavior in a P-FRP program. After collection, the state contains the values of all behaviors in a P-FRP program.

Definition

$$\begin{array}{lcl} \texttt{state}_{P \uplus \{x=b\}}(x) & \equiv & \texttt{state}_P(b) \\ \texttt{state}_P(c) & \equiv & c \\ \texttt{state}_P(f\langle d_i \rangle) & \equiv & \mathsf{prim}(f, \langle \texttt{state}_P(d_i) \rangle) \\ \texttt{state}_P(\mathsf{init}\ c\ \mathsf{in}\ H) & \equiv & c \end{array}$$

Let $[\![P]\!]$ be the unique Q such that $[\![P]\!] = Q$ (We have this Q by compilation determinism [15]). Then,

THEOREM 5.1 (CORRECTNESS OF COMPILATION).

1. $P \xrightarrow{I} S; P' \Longrightarrow [\![P']\!] \equiv [\![P]\!]$.
2. $P \xrightarrow{I} S; P' \Longrightarrow \exists n \geq 0.(\texttt{state}(P) \uplus T, [\![P]\!], \mathsf{en}, \mathsf{nil}, \mathsf{nil}) \overset{I,\diamond_n}{\longmapsto} (\texttt{state}(P') \uplus T', \mathsf{en}, \mathsf{nil}, \mathsf{nil})$.

5.2 Resource Boundedness

Now, we turn to resource boundedness. Because physical resources are constrained to a fixed limit, it is important that stack growth is bounded. We prove that our stack size is bounded and provide a way to calculate it given some initial stack configuration.

Definition We define the *size* of a stack σ, written as $\texttt{size}(\sigma)$, as:

$$\texttt{size}(\mathsf{nil}) \equiv 0 \qquad \texttt{size}((I, A, \triangle) :: \sigma) \equiv \texttt{size}(\sigma) + 1$$

THEOREM 5.2 (STACK BOUNDEDNESS). *If* $(S, Q, \mathsf{en}, (I_0, A, \bot) :: \sigma, q) \overset{Z}{\longmapsto} (S', m', \sigma' :: (I_0, A, \bot) :: \sigma, q')$, *then the maximum value of size*(σ') *is* $l_{max} - E(I_0)$ *where* $Z \equiv I_1, \diamond_{k_1}, I_2, \diamond_{k_2}, \ldots, I_n, \diamond_{k_n}$ *and* l_{max} *is the greatest interrupt priority at the system.*

5.3 Atomicity

Finally, we justify the claims about the preservation of the atomicity property for handling events. We begin with a simple example that illustrates the problem addressed by atomicity. Consider the following code:

$$\begin{array}{ll} L \rightarrow 1, H \rightarrow 2 & x = \mathsf{init}\ 0\ \mathsf{in}\{H_1\ \Rightarrow\ x + z\}, \\ y = \mathsf{init}\ 0\ \mathsf{in}\{H_1\ \Rightarrow\ y - z\}, & z = \mathsf{init}\ 1\ \mathsf{in}\{H_2\ \Rightarrow\ z + 1\} \end{array}$$

E-FRP semantics tells us that the value of $x + y$ should always be zero. Naively allowing preemption would violate this property. The higher priority event H_2 can interrupt the execution of the handler for the lower event H_1 and update z before the statement $y - z$ in H_1's handler for y has executed. We will show that this cannot happen in our model.

If an event J of lower priority occurs while a higher priority event is running, J is always queued, and its handler is executed after I's handler completes. If an event I of higher priority occurs while a lower priority event J is running, then there are three possibilities: (1) If I was copying, then J is queued and its handler runs as soon as copying is over. When J is done, I is restarted. (2) If I was computing, then J's handler runs immediately and after it is done, I is restarted. (3) If I was restoring, then J is queued and its handler runs after I is done. The following result addresses each of these three cases.

THEOREM 5.3 (REORDERING). *Assuming* $E(J) > E(I) > E(L_t)$ *for all* $L_t \in q$, *if* $(S, Q, \mathsf{en}, \sigma, q) \overset{Z}{\longmapsto} (S', \mathsf{en}, (I, A, \bot) :: \sigma, q)$ *and* $(S', Q, \mathsf{en}, (I, A, \bot) :: \sigma, q) \overset{J,\diamond_m}{\longmapsto} (S'', \mathsf{en}, \sigma, q)$ *for some* n *and* m, *then either of 1-3 holds:*

1. $A \not\equiv \langle\rangle$, $Z \equiv I, \diamond_n, J, \diamond_{n_1}$ *and* $(S, Q, \mathsf{en}, \sigma, q) \overset{J,\diamond_j}{\longmapsto} (\hat{S}, \mathsf{en}, \sigma, q)$ *and* $(\hat{S}, Q, \mathsf{en}, \sigma, q) \overset{I,\diamond_i}{\longmapsto} (S'', \mathsf{en}, \sigma, q)$ *for some* j *and* i
2. $A \not\equiv \langle\rangle$, $Z \equiv I, \diamond_n$ *and* $(S, Q, \mathsf{en}, \sigma, q) \overset{J,\diamond_j}{\longmapsto} (\hat{S}, \mathsf{en}, \sigma, q)$ *and* $(\hat{S}, Q, \mathsf{en}, \sigma, q) \overset{I,\diamond_i}{\longmapsto} (S'', \mathsf{en}, \sigma, q)$ *for some* j *and* i
3. $A \equiv \langle\rangle$, $Z \equiv I, \diamond_n, J, \diamond_{n_1}$, *and* $(S, Q, \mathsf{en}, \sigma, q) \overset{I,\diamond_i}{\longmapsto} (\hat{S}, \mathsf{en}, \sigma, q)$ *and* $(\hat{S}, Q, \mathsf{en}, \sigma, q) \overset{J,\diamond_j}{\longmapsto} (S'', \mathsf{en}, \sigma, q)$ *for some* j *and* i.

This result states that if two events occur, one after the other, with the second interrupting the handler for the first, then the resulting C state is equivalent to the C state resulting from some reordering where each handler is executed sequentially, without being interrupted.

To generalize atomicity to multiple events occurring while an event I is executing, we apply Theorem 5.3 multiple times for each occurring event to produce a state where each event is permuted as if it occurred either before or after I. Given a starting point with a state S_s, queue q_s, master bit *on*, if $I, \diamond_n, \{J_i, \diamond_i\}_{i \in 1 \ldots z}$ are steps of execution that produce a final state S, then there are some steps $\{J_k, \diamond_{kk}\}_{k \ni E(J_k) > E(I)}$, $I, \diamond_j$, $\{J_l, \diamond_{ll}\}_{l \ni E(J_l) \leq E(I)}$ where $\{k\} \cup \{l\} = \{1 \ldots z\}$ whose execution produces a final state S' from the same starting point.

6. RESPONSIVENESS

We next formalize the notion of time to respond to events and illustrate the change in the guaranteed upper bounds for the small example presented earlier.

Suppose we have events I_i, with arrival rates r_i (occurrence per second), and the uninterrupted times to process each of them are t_i. The priority of I_i is i. The following table presents both the assumptions needed for the queue

	Assumption	Maximum Wait	Processing
E-FRP	$r_k \cdot \sum_{i=1}^{n} t_i \leq 1$	$(\sum_{i=1}^{n} t_i) - t_k$	t_k
P-FRP	$t_k \gg t_{k+1}$ $G_k \geq t_k$	$(n-k) \cdot G_k$	t_k

to have length at most one for each priority level and the maximum waiting and processing times for an event I_k in each of E-FRP and P-FRP:

For E-FRP, the longest possible length of the queue is $\sum_{i=1}^{n} t_i$. To ensure that no event is missed because the queue is full, we assume that the same event will not occur before the prior occurrence has been handled.

The maximum gap guaranteed to exist is defined by

$$G_k = 1/max(r_{k+1}, \cdots, r_n, (n-k) \cdot min(r_{k+1}, \cdots, r_n))$$

The maximum wait is the maximum time to find such a gap. Event I_k will be handled given that this gap is larger than t_k. Assuming that $t_{k+1} \gg t_k$, we can omit the processing time when finding the gap. While G_k may seem like a complex term, it is easy to see that I_n with highest priority requires no wait before being processed. Generally, we can guarantee that events with higher priority can be handled much faster in P-FRP than in E-FRP.

To validate our compilation strategy and the expected effect on responsiveness, we implemented interrupt restart in kernel mode under Windows XP. We added several software interrupts directly to the *Interrupt Descriptor Table* (IDT), which collects pointers to the event handlers that the x86 refers to upon an interrupt. Software interrupts allow us to test interrupts of our event handlers at specific points without facing the delay of hardware interrupts. We use the x86 *INT nn* instruction to jump immediately to injected interrupts and bypass Windows interrupt processing. Because Windows no longer handles priorities for us, we implemented the priority handling scheme given by the P-FRP semantics. The scheme is implemented as preamble and postamble sections to each handler, which consist of a set of rules corresponding to the semantics.

We use the *KeQueryPerformanceCounter* Windows function to measure the time to execute each handler. The function is a wrapper around the processor read-time stamp counter (RDTSC) instruction. The instruction returns a 64-bit value that represents the count of ticks from processor reset. We use randomly generated events so that the $i+1$-th occurrence of each event arrives at a time given by the formula

$$T(i+1) = 130 + Random(0,1) * 20 + T(i)$$

The table below shows the maximum number of ticks of wait time before an event handler is completed in P-FRP and E-FRP.[1] Each type of event occurs between 300 and 400 times.

Event	Priority	P-FRP	E-FRP	Speedup
Reset	31	38	56	1.47
H	1	59	64	1.08
L	0	448	250	0.56

[1]We use an Intel Pentium III processor machine, 930 MHz, 512 MB of RAM running Windows XP, Service Pack 2

As expected, the timings show that the priority mechanism allows us to reorganize the upper bounds on maximum process time so that higher-priority events run faster.

7. RELATED WORK

Broadly speaking, there are three kinds of related work, which consist of essential constructs for programming and analyzing interrupt-driven systems, software transactions, and other synchronous languages.

Palsberg and Ma, who present a calculus for programming interrupt-driven systems [19], introduce a type system that (like P-FRP) guarantees boundedness for a stack of incomplete interrupt handlers. To get this guarantee, the programmer "guides" the type system by explicitly declaring the maximum stack size in each handler and globally on the system. Palsberg and Ma's calculus allows for interrupts to be of a different priority: the programmer hardcodes their priority by manipulating a bit mask that determines which interrupts are allowed to occur at any point in a program. The programmer is thus responsible for ensuring that interrupts are correctly prioritized. Furthermore, Palsberg and Ma's work allows the programmer to have atomic sections in handlers: the programmer needs to disable/enable the correct set of (higher-priority) interrupts around such sections.

P-FRP statically guarantees stack boundedness without help from the programmer. In P-FRP, the programmer is also statically guaranteed correct prioritization of events and atomic execution of handlers at the expense of a fine control over atomicity.

Vitek et al. [16] present a concurrency-control abstraction for real-time Java called PAR (preemptible atomic region). PAR facilitates atomic transaction management at the thread level in the Java real-time environment. The authors restrict their analysis of execution guarantees to hard real-time threads, which are not allowed to read references to heap objects and thus must wait for the garbage collector. Even with this restriction, the environment complicates estimating worst-case execution time for threads. Our preemption is interrupt driven rather than context switch driven. Both works use transactions similarly, with the difference being that in Vitek's work an aborting thread blocks until the aborted thread's undo buffer is written back. Our work delays undos until an aborting event completes. While our work evaluates maximum waiting time and processing time for an event, Vitek's work answers a related responsiveness question in the context of threads: can a set of periodically executing threads run and complete within their periods if we know, for each thread, its maximum time in critical section, maximum time to perform an undo, and worst-case response time. Such an analysis could be an extension to our work in which we evaluate whether sequences of periodically occurring events can be handled in fixed blocks of time.

Nordlander et al. [18] discuss why a reactive programming style is useful for embedded and reactive systems. Currently, methods in thread packages and various middleware could block, which makes the enforcement of responsiveness difficult. Instead of allowing blocking, the authors propose setting up actions that react to future events. To enforce time guarantees, Jones at al. [17] focus on reactive programming that models real-timed deadlines. Just as with our responsiveness result, a deadline considers all reactions upon event

occurrence, i.e., every component involved in the handling of a particular event should be able to complete before a given deadline.

Ringenburg and Grossman [23] present a design for software transactions-based atomicity via rollback in the context of threads on a single processor. *Software transactions* are a known concurrency primitive that prohibits interleaved computation in atomic blocks while ensuring fairness (as defined in [10]). An atomic block must execute as though no other thread runs in parallel and must eventually commit its computation results. Ringenburg and Grossman use logging and rollback as a standard approach to undoing uncompleted atomic blocks upon thread preemption, and they retry them when the thread is scheduled to run again. Logging consists of keeping a stack of modified addresses and their previous values, and rollback means reverting every location in the log to its original value and restarting a preempted thread from the beginning.

In an extension to Objective Caml called AtomCaml, Ringenburg and Grossman connect the latter two processes by a special function that lets the programmer pass code to be evaluated atomically. This function catches a rollback exception, which a modified thread scheduler throws when it interrupts an atomic block, and then performs necessary rollback. Thread preemption is determined by a scheduler based on time quotas for each thread.

Like AtomCaml, P-FRP implements a transaction mechanism that allows handlers to execute atomically, even when they are preempted. These approaches are similar and are alternative methods of checking or inferring that lock-based code is actually atomic ([8, 7]). On the other hand, AtomCaml and P-FRP are two design choices for atomicity via rollback in two different environments' threads and event handlers. Threads are not prioritized as event handlers and run only during their time quotas. Ringenburg and Grossman [23] focuses on implementation and evaluation of software transactions and only informally discusses guarantees for time and stack boundedness and for reordering of preempted threads. In contrast, in this paper we define a semantics that allows us to formally establish such properties for our transactional compiler.

Harris et al. [12] integrate software transactions with Concurrent Haskell. Previous work on software transactions did not prevent threads from bypassing transactional interfaces, but Harris et al. use Haskell's type system to provide several guarantees:

- An atomic block can only perform memory operations, rather than performing irrevocable input/output.
- The programmer cannot read or write mutable memory without wrapping these actions in a transaction. This eases reasoning about interaction of concurrent threads.

We would like to ease the restrictions of the first point and allow revocable input/output from/to specified device memory-mapped registers. Just as other threads can modify transaction-managed variables, C global variables generated by P-FRP can be modified by an external event handler. We do not yet have a solution for read or write protecting these global variables in the operating system kernel mode. Harris et al. also provide support for operations that may block. A blocking function aborts a transaction with no effect, and restarts it from the beginning when at least one of the variables read during the attempted transaction is updated. P-FRP can be extended to support blocking transactions by polling device registers. Furthermore, Harris et al. allow the programmer to build transactional abstractions that compose well sequentially or as alternatives so that only one transaction is executed. It would be a useful, addition to P-FRP, to allow smaller transaction granularity, so that the programmer can specify which parts of an event handler need to execute as a transaction. Harris et al. [13] observe that implementations of software transactions [12] use thread-local transaction logs to record the reads and tentative writes a transaction has performed. These logs must be searched on reads in the atomic block. Harris et al. suggest some improvements over the implementation, as follows:

- Compiler optimizations to reduce logging
- Runtime filtering to detect duplicate log entries that are missed statically
- GC time techniques to compact logs during long running computations .

In P-FRP, it would be useful to analyze variable use and to determine whether a given handler needs to execute as a transaction. Second, it would be useful to reduce the number of shadow variables by guaranteeing their safe reuse.

Other languages support the *synchronous approach* to computation where a program is instantaneously reacting to external events. In the imperative language Esterel [2, 1], *signals* are used as inputs or outputs, with signals having optional values and being broadcast to all processes. They can be emitted simultaneously by several processes, and *sensors* are used as inputs and always carry values. An *event* is a set of simultaneous occurrences of signals. Esterel, like the original E-FRP, assumes that control takes no time. The Esterel code below is an if-like statement that detects whether a signal X is present. Based on the test, a branch of the if-like statement is executed immediately.

```
present S(X) then <statement1>
             else <statement2> end
```

The assumption of atomic execution might not be reasonable if a branch is being executed when X occurs. There are compilers that translate Esterel into finite state machines, and in such a compiler's target, X would occur during multiple transitions in a finite state machine, which might be is undesirable. The same is true for languages Signal [9] and Lustre [4], in which a synchronous stream function corresponds to a finite state machine with multiple transitions. The original E-FRP also has this problem, which P-FRP solves by stating explicitly which actions are taken at any point that an event occurs.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented a compilation strategy that provides programmers with more control over the responsiveness of an E-FRP program. We have formally demonstrated that properties of E-FRP are preserved and that prioritization does not alter the semantics except for altering the order in which events appear to arrive.

There are several important directions for future work. In the immediate future, we are interested in simplifying the underlying calculus of the language, as well as studying

valid optimizations of the generated code. We are also interested in determining quantitative upper bounds for the response time, as well as the space needed to handle each event. Finally, we expect to continue to build increasingly larger applications in E-FRP, which should allow us to validate our analytical results using the real-time performance of programs written in the language.

9. ACKNOWLEDGMENTS

We thank Emir Pasalic and Jeremy Siek for many valuable suggestions and discussions related to this work. David Johnson, and Robert (Corky) Cartwright served on the Masters thesis committee for the first author, and Ray Hardesty helped us improve our writing greatly.

10. REFERENCES

[1] G. Berry. *The Constructive Semantics of Pure Esterel. Draft 3*. ftp://ftp-sop.inria.fr/esterel/pub/papers/constructiveness3.ps, July 1999.

[2] G. Berry and L. Cosserat. The Esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University*, pages 389–448, London, UK, 1985. Springer-Verlag.

[3] D. D. Bovet and M. Cesati. *Understanding the Linux kernel*. O'Reilly & Associates, Inc., Sebastopol, CA, 2000.

[4] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *POPL'87*, pages 178–188, New York, NY, USA, 1987. ACM Press.

[5] K. Donnelly and M. Fluet. Transactional events. In *ICFP'06*, pages 124–135. ACM Press, September 2006.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *ICFP'97*, volume 32(8), pages 263–273, 1997.

[7] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI'03*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[8] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI'03*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[9] T. Gautier, P. L. Guernic, and L. Besnard. Signal: A declarative language for synchronous programming of real-time systems. In *FPCA'87*, pages 257–277, London, UK, 1987. Springer-Verlag.

[10] D. Grossman. Software transactions are to concurrency as garbage collection is to memory management. Technical report, UW-CSE, Apr. 2006.

[11] T. Harris and K. Fraser. Language support for lightweight transactions. In *OOPSLA'03*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[12] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP'05*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[13] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *SIGPLAN*, vol. 41(num. 6):p14–25, 2006.

[14] P. Hudak. *The Haskell School of Expression - Learning Functional Programming Through Multimedia*. Cambridge University Press, 2000.

[15] R. Kaiabachev, W. Taha, and A. Zhu. *E-FRP with Priorities (extended version)*. http://www.cs.rice.edu/taha/publications/preprints, 2007-08-15-TR.pdf.

[16] J. Manson, J. Baker, A. Cunei, S. Jagannathan, M. Prochazka, B. Xin, and J. Vitek. Preemptible atomic regions for real-time java. In *RTSS'05*, pages 62–71, Washington, DC, USA, 2005. IEEE Computer Society.

[17] J. Nordlander, M. Carlsson, M. P. Jones, and J. Jonsson. Programming with time-constrained reactions. In *Submitted for publication*, 2004.

[18] J. Nordlander, M. P. Jones, M. Carlsson, R. B. Kieburtz, and A. Black. Reactive objects. In *ISORC'02*, page 155, Washington, DC, USA, 2002. IEEE Computer Society.

[19] J. Palsberg and D. Ma. A typed interrupt calculus. In *FTRTFT'02, LNCS 2469*, pages 291–310. Springer-Verlag, 2002.

[20] J. Peterson, G. Hager, and P. Hudak. A language for declarative robotic programming. In *ICRA'99*. IEEE, May 1999.

[21] J. Peterson and K. Hammond. Haskell 1.4, a non-strict, purely functional language. Technical report, Haskell comittee, Apr. 1997.

[22] A. Reid, J. Peterson, G. Hager, and P. Hudak. Prototyping real-time vision systems: An experiment in DSL design. In *ICSE'99*, pages 484–493, 1999.

[23] M. F. Ringenburg and D. Grossman. AtomCaml: First-class atomicity via rollback. In *ICFP'05*. ACM, September 2005.

[24] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly & Associates, Inc., Sebastopol, CA, second edition, 2001.

[25] T. Shanley. *Protected Mode Software Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[26] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95*, pages 204–213, August 20–23 1995. Ottawa, Ont. Canada.

[27] D. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, 3rd edition, 2000.

[28] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, Vol. 21(Num. 10):p10–19, 1988.

[29] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *PLDI'00*. ACM, 2000.

[30] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *ICFP'01*, pages 146–156, New York, NY, USA, 2001. ACM Press.

[31] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *PADL'02*, Lecture Notes in Computer Science. Springer, January 19-20 2002.