Appears in the proceedings of the *1st International Conference on Runtime Verification* (RV 2010).

Copilot: A Hard Real-Time Runtime Monitor

Lee Pike[1], Alwyn Goodloe[2], Robin Morisset[3], and Sebastian Niller[4]

[1] Galois, Inc. leepike@galois.com
[2] National Institute of Aerospace alwyn.goodloe@nianet.org
[3] École Normale Supérieure robin.morisset@ens.fr
[4] Technische Universität Ilmenau sebastian.niller@stud.tu-ilmenau.de

Abstract. We address the problem of runtime monitoring for hard real-time programs—a domain in which correctness is critical yet has largely been overlooked in the runtime monitoring community. We describe the challenges to runtime monitoring for this domain as well as an approach to satisfy the challenges. The core of our approach is a language and compiler called *Copilot*. Copilot is a stream-based dataflow language that generates small constant-time and constant-space C programs, implementing embedded monitors. Copilot also generates its own scheduler, obviating the need for an underlying real-time operating system.

1 Introduction

Safety-critical control systems, such as avionics and drive-by-wire systems, are well-tested, sometimes certified, and perhaps even formally verified. Yet undetected errors or incorrect environmental assumptions can cause failures resulting in the loss of life—as these control systems become more complex and pervasive, the risk of software failure grows. Hence, this domain begs for the application of runtime monitoring.

Hard real-time systems are ones in which correctness depends on execution occurring within a fixed period of time [GR04]. Surprisingly, most previous research in runtime monitoring focuses either on non real-time programs or *soft real-time* systems, in which occasionally missing deadlines is tolerated. To partially redress this deficiency in the literature, we address the problem of monitoring the class of hard real-time systems: in particular, we develop a monitoring framework for periodically-scheduled hard real-time systems.

In designing our monitoring framework, we apply four guiding principles we believe are fundamental constraints for any monitoring approach treating this domain:

1. *Functionality*: Monitors cannot change the functionality of the observed program unless a failure is observed.
2. *Schedulability*: Monitors cannot alter the schedule of the observed program.
3. *Certifiability*: Monitors must minimize the difficulty in re-validating the observed program; in particular, we make it our goal to avoid modifying the observed program's source code.

4. *SWaP overhead*: Monitors must minimize the additional overhead required including size, weight, and power (SWaP).

To satisfy these objectives, we have developed a simple stream language called *Copilot* that compiles into small constant-time and constant-space (i.e., no dynamic memory allocation) C programs. The language follows a sampling-based monitoring strategy in which global variables of the observed program (or programs) are periodically sampled; Copilot provides mechanisms for controlling when to observe the variables. Furthermore, using the Atom compiler [Haw08] as a back-end, Copilot automatically generates its own periodic schedule, allowing for easy integration into the periodic schedule of the observed program. By generating its own schedule, the monitor obviates the need for a real-time operating system (RTOS) for scheduling and concurrency control and so can be executed on minimal embedded hardware. The language is implemented as an embedded domain-specific language (eDSL) in the popular functional language Haskell [Jon02].

Outline The remainder of the paper is organized as follows. Related work is described in Section 2. We describe and defend the use of state-variable sampling as our monitoring approach in Section 3. In Section 4, we present the syntax, types, and semantics for our Copilot language. We then present a lower-level semantics of the language with respect to logical time in Section 5; we also discuss our scheduling assumptions in more detail there. In Section 6, we present a synthesis (or compilation) algorithm for transforming a Copilot specification into a state-machine and briefly describe the implementation. We make concluding remarks and point to future work in Section 7.

2 Related Work

Monitoring and Checking (MaC) [KLKS04] and Monitor Oriented Programming (MOP) [CDR04] represent the state-of-the-art in monitoring frameworks, but are targeted at Java applications that are not hard real-time systems (a version of MaC targeted at C programs is also under development). The Requirement Monitoring and Recovery (RMOR) [Hav08] framework is (one of the first monitoring frameworks) targeting C programs. RMOR differs from our approach in that it requires that probes, built using aspect-oriented techniques, be inserted in the code at each location where state is updated, and it does not address the issues of monitoring real-time programs. Recent work on time-aware instrumentation applies static analysis techniques and novel algorithms to calculate an instrumentation that, when possible, satisfies the time budget [FL09]. Although its focus is on soft real-time systems, predictable runtime monitoring defines a monitor budget restricting the resources allowed to the monitor so that the composed system can perform in a predictable fashion [ZDG09]. We have taken an alternative approach that does not require modifying the monitored program (see Section 3). Pellizzoni *et al.* have constructed no-overhead monitors in which

the monitors are implemented on FPGAs; the framework targets properties of a PCI bus [PMCR08].

The Copilot language is influenced by functional and stream-based languages. The syntax and semantics of infinite Haskell lists influence the syntax and the untimed semantics (see Section 4.3) of Copilot [Jon02]. The languages Lustre [HCRP91], μCryptol [PSM06], and Lola [DSS$^+$05] are all stream-based languages that influence the design of Copilot; in particular, Lustre and μCryptol are designed for use on embedded microprocessors.

As explained in detail in Section 6.2, Copilot is a domain specific language (DSL) that is embedded in the functional programming language Haskell. Similar DSLs used to generate embedded C code include Feldspar [ACD$^+$10], used for digital signal processing, and Atom [Haw08], used for embedded control system design. Indeed, Copilot uses Atom as a "back-end" in the compiler.

3 Sampling-Based Monitoring

Monitoring based on sampling state-variables has largely been disregarded as a runtime monitoring approach, for good reason: without the assumption of synchrony between the monitor and observed software, monitoring via sampling may lead to false positives and false negatives [DDE08]. For example, consider the property $(0; 1; 1)^*$, written as a regular expression, denoting the sequence of values a monitored variable may take. If the monitor samples the variable at the inappropriate time, then both false negatives (the monitor erroneously rejects the sequence of values) and false positives (the monitor erroneously accepts the sequence) are possible. For example, if the actual sequence of values is $0, 1, 1, 0, 1, 1$, then an observation of $0, 1, 1, 1, 1$ is a false negative by skipping a value, and if the actual sequence is $0, 1, 0, 1, 1$, then an observation of $0, 1, 1, 0, 1, 1$ is a false positive by sampling a value twice.

However, in a hard real-time context, sampling is a suitable strategy. Under the assumption that the monitor and the observed program share a global clock and a static periodic schedule, while false positives are possible, false negatives are not. A false positive is possible, for example, if the program does not execute according to its schedule but just happens to have the expected values when sampled. If a monitor samples an unacceptable sequence of values, then either the program is in error, the monitor is in error, or they are not synchronized, all of which are faults to be reported.

Most of the popular runtime monitoring frameworks described in Section 2 inline monitors in the observed program to avoid the aforementioned problems with sampling. However, in the domain of embedded real-time systems, that approach suffers the following problems, recalling our four criteria from Section 1. First, inlining monitors changes the real-time behavior of the observed program, perhaps in unpredicable ways. In a sampling-based approach, the monitor can be integrated as a separate scheduled process during available time-slices (this is made possible by generating efficient constant-time monitors). Indeed, sampling-based monitors may even be scheduled on a separate processor (albeit doing so

requires additional synchronization mechanisms), ensuring time and space partitioning from the observed programs. Such an architecture may even be necessary if the monitored program is physically distributed. Another shortcoming of inlining monitors is that certified code (e.g., DO-178B for avionics [Inc92]) is common in this domain. Inlining monitors could necessitate re-certifying the observed program. We cannot claim our approach would obviate the need for re-certification, but it is a more modular approach than one based on instrumenting the source code of the observed program, which may result in a less onerous re-certification process.

4 The Copilot Language

In this section, we overview the syntax, type system, and semantics of Copilot.

4.1 Syntax

The Copilot language is a synchronous language described by a set of *stream equations*. A *stream* is an infinite sequence of values from some type. A *stream index* i is a non-negative integer; for stream σ, $\sigma(i)$ is the stream's value at index i. It is assumed that the value stored in stream index zero $\sigma(0)$ is an initial value.

To get a feel for the Copilot language, consider the following property of an engine controller:

> *If the temperature rises more than 2.3 degrees within 0.2 seconds, then the engine is immediately shut off.*

Assume the period at which the temperature variable *temp* is sampled is 0.1 seconds, and the shut-off variable is *shutoff*. Then the property can be specified as follows in Copilot:

$$\begin{aligned} temps &= [0,\ 0,\ 0] \mathbin{++} extF\ temp\ 1 \\ overTempRise &= drop\ 2\ var\ temps > const\ 2.3 + var\ temps \\ trigger &= (var\ overTempRise)\ \text{implies}\ (extB\ shutoff\ 2) \end{aligned}$$

When the stream *trigger* becomes false, the property has failed.

A Copilot *monitor specification* is a nonempty set of stream equations defining typed *monitor variables* $m_0,\ m_1,\ \ldots,\ m_n$ of the form $m_i = EXP$ where EXP is an expression built from the BNF grammar in Figure 1. (We slightly simplify the grammar from our implementation, omitting expression terminals and type declarations.) In the grammar, the terminal $<Identifier>$ is a valid C99 variable name, and $<n>$ is a non-negative integer. Streams of Boolean values are used as triggers, signalling a property succeeding or failing.

Informally, the intended semantics for Copilot is the semantics of lazy streams, like in Haskell [Jon02]. In particular, the operation $++$ is lazy list-append, and appends a finite list onto a stream. The operation $drop\ s\ n$ drops the first n indexes from stream s.

stream definition	EXP	$= VAR \mid CVAR \mid APP \mid DROP \mid FUN \mid CONST$
monitor variable	VAR	$= var\ \langle Identifier \rangle$
sample expression	$CVAR$	$= CTYPE\ \langle Identifier \rangle\ \langle n \rangle$
typed program variable	$CTYPE$	$= extB \mid extI8 \mid extI16 \mid extI32 \mid extI64 \mid extW8 \mid extW16 \mid extW32 \mid extW64 \mid extF \mid extD$
stream drop	$DROP$	$= drop\ \langle n \rangle\ e$ where $e = VAR \mid CVAR \mid DROP \mid CONST$
stream append	APP	$= l +\!\!+ EXP$ where l is a finite list of constants
function application	FUN	$= f(e_0,\ e_1,\ \ldots,\ e_n),$ where $e_i = VAR \mid CVAR \mid DROP \mid FUN \mid CONST$
constant stream	$CONST$	$= const\ c$ where c is a constant

Fig. 1. Simplified Copilot Grammar.

Besides monitor variables, the other class of variables in Copilot are *program variables*. Program variables reference global variables being sampled. Program variables can be any shared state accessible by the compiled C program monitor, including hardware registers or other C program variables. In a sampling expression, e.g., $extW64\ v\ 3$, the integer refers to the *phase* (or offset) into the periodic schedule at which v is to be sampled (see Section 5). In $CTYPE$ expressions, the '`ext`' in the constructor denotes 'external', '`W` denotes 'word', '`I` denotes 'int', '`F`' denotes 'float', and '`D`' denotes 'double'. An expression containing no program variables is a *closed expression*; otherwise it is an *open expression*. Monitors are defined by open expressions, but closed expressions are useful as "helper streams"—e.g., counters to create new clocks [HCRP91]—for other definitions.

An expression *const* 3 denotes a stream of the value 3.

The functions of the language include the usual arithmetic operators (e.g., $+,\ -,\ *,\ /,$ *modulo*, $==,\ <$ and the other comparison operators), and the logical operators *not*, *and*, *or*, *implies*. Other operators can be easily added to the language.

The append operator binds more weakly than function application, which binds more weakly than the drop operator. Variable operators bind most tightly.

Example 1 (Closed Monitor Expressions). We present simple closed monitor expressions below along with their intended semantics.

Monitor	Intended semantics
$m_0 = [\mathbf{T},\ \mathbf{F}] +\!\!+ var\ m_0$	$\mathbf{T},\ \mathbf{F},\ \mathbf{T},\ \mathbf{F},\ \ldots$
$m_1 = [\mathbf{T}] +\!\!+ const\ \mathbf{F}$	$\mathbf{T},\ \mathbf{F},\ \mathbf{F},\ \mathbf{F},\ \ldots$
$m_2 = drop\ 1\ (var\ m_3)$	$1,\ 2,\ 1,\ 2,\ \ldots$
$m_3 = [0,\ 1,\ 2] +\!\!+ var\ m_2$	$0,\ 1,\ 2,\ 1,\ 2,\ \ldots$
$m_4 = [0,\ 1] +\!\!+ var\ m_4 + drop\ 1\ (var\ m_4)$	$0,\ 1,\ 1,\ 2,\ 3,\ \ldots$

Note that m_4 generates the Fibonacci sequence.

One design choice with the language is to disallow stream append expressions to appear within the context other operators. For example, the expressions $drop\ 1\ ([0,\ 1]\ ++\ var\ m)$ and $([0,\ 1]\ ++\ var\ m) + (const\ 3)$ are ill-formed. This decision ensures there are no "anonymous streams" in the language—i.e., each newly-constructed stream is either constant (*const*) or a function of streams assigned to a monitoring variable. The choice provides better control over the memory usage required by the monitor (it is a linear function of each monitor variable defined; see Section 6).

Example 2 (Embedding past-time LTL). The past-time LTL (ptLTL) operators are past-time analogues of the standard LTL operators. The ptLTL operators include previously ($\mathcal{P}$), has always been ($\mathcal{A}$), eventually previously ($\mathcal{E}$), and since ($\mathcal{S}$). Given their semantics defined in [MP92], they are defined in Copilot as follows (assume the appropriate *cType* and fixed phases n and l):

$$\begin{aligned}
\mathcal{P}p &\equiv m_0 = [\mathbf{F}]\ ++\ cType\ p\ n \\
\mathcal{A}p &\equiv m_1 = var\ m_2 \wedge\ cType\ p\ n, \text{ where} \\
&\quad\ m_2 = [\mathbf{T}]\ ++\ var\ m_2\ \wedge\ cType\ p\ n \\
\mathcal{E}p &\equiv m_3 = var\ m_4 \vee\ cType\ p\ n, \text{ where} \\
&\quad\ m_4 = [\mathbf{T}]\ ++\ var\ m_4\ \vee\ cType\ p\ n \\
p_0 \mathcal{S} p_1 &\equiv m_5 = cType\ p_1\ n \vee (cType\ p_0\ l\ \wedge\ m_6), \text{ where} \\
&\quad\ m_6 = [\mathbf{F}]\ ++\ m_6
\end{aligned}$$

4.2 Types

Copilot is statically and strongly typed—i.e., type-checking is done at compile-time, and type-incorrect function application is not possible. In our implementation, Copilot types are embedded into Haskell's type system (see Section 6.2). Copilot specifications lift C types to streams. The C types lifted are the C types corresponding to $CTYPE$ in Figure 1.

In the following, let $\overrightarrow{T}$ denote the type T lifted to the type of an infinite stream of values of type T. We denote that "expression *exp* has type T" by $exp :: T$. The type of an expression is the smallest relation satisfying the following:

- If m is a monitor variable of type $\overrightarrow{T}$, then $(var\ m) :: \overrightarrow{T}$.
- For a program variable expression, $(cType\ v\ n) :: \overrightarrow{T}$, where T is the type corresponding to the operator *ctype*.
- If $c :: T$ for each constant c in the list l and $exp :: \overrightarrow{T'}$, then $(l\ ++\ exp) :: (\overrightarrow{T} \cup \overrightarrow{T'})$.
- If $exp :: \overrightarrow{T}$, then $drop\ i\ exp :: \overrightarrow{T}$.
- If f is a n-ary function such that $f :: T_0,\ T_1,\ \ldots,\ T_n \to T$, and $exp'_0 :: \overrightarrow{T_0},\ exp'_1 :: \overrightarrow{T_1},\ \ldots,\ exp'_n :: \overrightarrow{T_n}$, then $f(exp'_0,\ exp'_1,\ \ldots,\ exp'_n) :: \overrightarrow{T}$.
- If $c :: T$, then $(const\ c) :: \overrightarrow{T}$.

If $exp :: \overrightarrow{T}$ and $\overrightarrow{T_0}, \overrightarrow{T_1} \subseteq \overrightarrow{T}$ and $T_0 \neq T_1$, then the expression is *type incorrect*.

4.3 Untimed Semantics

Due to space considerations, we do not provide a formal semantics for Copilot. Following [DSS+05], Copilot's untimed semantics is defined in terms of *evaluation models*. Informally, an evaluation model is the n-tuple of streams denoted by a monitor specification, assuming a fixed set of streams denoting the values of program variables. Evaluation models are constructed inductively over the syntax of the specification assuming a fixed set of program variable values. A specification is said to be *well defined* if the values of the monitor variables at time t are uniquely defined by the values of the monitored variables at times $0 \ldots t$.

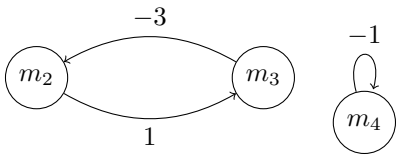
For example, well-definedness rules out specifications of the form $m = \neg(var\ m)$ no value for m can statisfy that equation. Well-definedness also rules out specifications of the form $m = drop\ 1\ (var\ m)$ since it admits several different solutions (e.g., both of the streams $\mathbf{T},\ \mathbf{T},\ \ldots$ and $\mathbf{F},\ \mathbf{F},\ \ldots$).

Monitor specifications can be restricted syntactically to ensure they are well-defined. Define a *dependency graph* to be a directed, weighted, graph $(V,\ E)$ such that the vertexes V are the monitor and program variables. The edges E are constructed as follows: for variables v and v', $v \xrightarrow{w} v' \in E$ if and only if v' appears in some subexpression in the right-hand side of the stream equation for v (note that program variables are only sinks in the graph), and $w = weight(v)$, where

$$
\begin{aligned}
&weight(exp) = \text{case } exp \text{ of} \\
&\qquad \begin{array}{lcl}
l\ ++\ e & \to & weight(e) - length(l) \\
drop\ i\ e & \to & i + weight(e) \\
f(e_0,\ e_1,\ \ldots,\ e_n) & \to & max(weight(e_0),\ \ldots,\ weight(e_n))_i \\
var\ v' & \to & 0 \\
cType\ v'\ n & \to & 0 \\
const\ c & \to & -\infty
\end{array}
\end{aligned}
$$

A *walk* of a dependency graph is a finite sequence of variables $v_0,\ v_1,\ \ldots,\ v_n$ such there exists an edge from v_i to v_{i+1}, for each v_i of the sequence. Variable v_i *depends* on v_j if v_i and v_j both appear in some walk, and $i < j$. A *loop* is a walk $v_0,\ v_1,\ \ldots,\ v_n$ such that $v_0 = v_n$. A *closed walk* is a walk $v_0,\ v_1,\ \ldots,\ v_n$ such that $v_i = v_n$ for some $0 \leq i < n$. The *weight* of a walk $v_0,\ v_1,\ \ldots,\ v_n$ is the sum of the weights in the sequence.

Example 3. The dependency graphs for m_2, m_3, and m_4 in Example 1 are depicted below.



We make two restrictions to ensure that specifications are well-defined; one constrains the dependencies between program variables, and one constrains the dependencies of monitor variables on program variables. For the first constraint, we preclude circular dependencies on future values in stream definitions. For example, $m = \mathit{drop}\ 1\ (\mathit{var}\ m)$, has a circular dependency on its own future values, and in

$$m_0 = \mathit{drop}\ 1\ (\mathit{var}\ m_1)$$
$$m_1 = \mathit{var}\ m_0$$

m_0 and m_1 depend on each others' future values. Both specifications are not well-defined. A sufficient condition is to require the weight of loops in the dependency graph to be less than zero.

The second restriction is analogous but ensures a specification does not attempt to reference future program values: the weight of a walk terminating in a program variable must be less than or equal to zero. Thus, we have the following definition:

Definition 1 (Well-Formed Specification). *A monitor specification is* well-formed *if there exists*

- *No loop with a non-negative walk weight.*
- *No walk with a positive weight terminating in an external variable.*

Theorem 1 (Well-Formedness Theorem). *Every well-formed specification is well-defined.*

As noted in [DSS+05], the converse of the theorem does not hold:

$$m_0 = (\mathit{var}\ m_0) \vee \mathbf{T}$$
$$m_1 = [0,\ 1] +\!\!+ \text{ if } \mathbf{F} \text{ then } \mathit{drop}\ 1\ (\mathit{cType}\ p\ n) \text{ else } \mathit{var}\ m_1$$

Both specifications are well-defined but not well-formed.

In addition to the well-formedness constraints, we introduce two minor additional constraints in Section 6 for the purpose of reducing the worst-case execution time and memory usage in our implementation.

5 Scheduling Semantics

In Section 4.3, we described an untimed semantics for Copilot. In this section, we describe the semantics of a Copilot implementation with respect to logical time [Lam78]. That is, we assume a *global clock* is the sequence of non-negative integers, and every stream shares the global clock. A *(clock) tick* is a value from the global clock sequence. We assume synchronization with respect to the abstract global clock, so every stream agrees on the time, but we do not assume an order of execution *within* a clock tick. Thus, one stream cannot depend on another stream having computed its next-state value during the current tick.

Not assuming an order of execution within a tick provides flexibility in implementing a monitor; for example, a monitor might be distributed on separate processors with the guarantee that synchronization is only required up to the global clock [HCRP91]. The compiler ensures the same behavior regardless of the order in which state variables are updated in the same tick (see Section 6).

We follow a standard model of hard real-time scheduling [GR04]. A monitor is a collection of recurring *tasks* (in our setting, C functions) that obtain inputs and compute output in a statically-bounded amount of time. We assume tasks are *periodic* and have a round-robin non-preemptive schedule. Consequently, all tasks have the same priority and run to completion without interrupts. The global clock is an abstraction of the hardware clock; the duration of each tick of the global clock is expected to be sufficiently long to account for the worst-case execution time (WCET) of all possible computation that occurs within a tick. A tick is triggered by sampling the hardware clock.

Typically, we assume the monitored program also has these scheduling characteristics. In this case, the monitor can be integrated into the round-robin schedule of the observed program, provided WCET constraints are met. However, the monitor can also be scheduled as a single high-priority task that manages its own sub-tasks (e.g., sampling) according to the schedule it generates. Care must be taken that the monitor's temporal assumptions are met under this framework.

At the ticks at which a state variable is scheduled to be assigned a new value, we say the variable *fires*; otherwise, we say the variable *idles*. A variable's schedule can be succinctly stated in terms of a positive integer p that is its *period* and a non-negative integer h, where $h < p$, that is the stream's *phase*. The period denotes the number of ticks between successive firings for a state variable, and the phase denotes the offset into each period for when it fires. For a clock tick C, when $(C - h) \bmod p \equiv 0$, the variable fires; otherwise, it is idle.

Example 4 (Timed Semantics). Consider the stream specifications for m_2 and m_3 from Example 1. Suppose m_2 has period 3 and phase 0, and m_3 has period 3 and phase 1. Then the stream's timed semantics are as follows, where $\perp$ denotes "undefined" or "do not care":

$$\begin{aligned} \text{global clock} &= 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ \ldots \\ m_2 &= 1\ 1\ 1\ 2\ 2\ 2\ 1\ 1\ 1\ 2\ \ 2\ \ldots \\ m_3 &= \perp\ 0\ 0\ 0\ 1\ 1\ 1\ 2\ 2\ 2\ \ 1\ \ldots \end{aligned}$$

While the schedule of a monitor variable denotes when the variable fires, the schedule of a program variable denotes when the monitor samples it. Consequently, a sampling expression (i.e., *cType v n*) denotes that v is sampled at phase n in each period. For period p, we require $0 < n < p$. The constraint $0 < n$ ensures that the compiler has a tick to update state when variables are not being sampled (see Section 6). Recall the initial specification in Section 4.1: there we formalized "the engine is *immediately* shut off" by sampling the program variable *shutoff* in the tick just after sampling *temp*.

Given our model, we can state the correctness condition for a stream in a monitor specification to be implemented by a scheduled state variable:

Definition 2 (Stream Implementation). *We say that the state variable v with period p and phase h* implements *the stream σ if for all clock times C, $v(C) = \sigma(idx(C))$, where $idx(C) = \left\lfloor \frac{C-h}{p} \right\rfloor$ if $h \leq C$, and $idx(C) = \left\lceil \frac{C-h}{p} \right\rceil$ otherwise.*

6 Monitor Synthesis

In this section, we describe the synthesis of a Copilot specification to a state machine, which Atom [Haw08] compiles to C code. The state machine is represented by a set of state variables associated with each stream in the specification, an initial state, a state-update function for each variable, and a schedule for applying the state-update function. The synthesis algorithm is very simple and produces code with a low and uniform WCET. However, the simple algorithm requires us to make two additional well-formedness restrictions, generalizing Definition 1 slightly; we describe these additions in Section 6.1.

Besides synthesizing the specification, the compiler schedules the monitors within the overall periodic schedule of the observed program. The synthesis algorithm generates a schedule that (1) respects causality constraints—i.e., the data required to compute a value is available and that (2) interferes with the program's real-time constraints as little as possible. In our implementation, these two criteria are handled at different levels of the compiler. The purpose of (1) is to ensure that the Stream Implementation definition (Definition 2) holds. (2) is an optimization issue; the Atom scheduler handles (2) by optimizing the schedule (see Section 6.2).

Remark 1 (Array and List Notation). We store state values in arrays, and define some functions that operate over arrays and lists. We denote the value at index j in an array or list a by $a[j]$. The function $len(a)$ takes an array or finite list and returns the length of a. The array $<>$ is the empty array. The function l *app* a takes a finite list l and array a and returns an array a' formed by appending the values in l onto the front of a.

In the following, assume a monitor specification consists of a finite sequence of monitor variable definitions of the form

$$m_0 = exp_0,\ m_1 = exp_1, \ldots,\ m_n = exp_n$$

State For each monitor variable m_i, its state contains the following:

- *History variables*: stream values are stored in a *history array*, a_i. The length of the history array is statically-computed from the monitor specification.
- *Update and output indexes*: two elements of the history array are respectively designated as an *update index* ($upIdx$), the index of the next-state value, and an *output index* ($outIdx$), the index of the current output value.

Additionally, for each unique sampling expression $cType\ v\ n$, in a specification, we introduce a *temporary sampling variable* v_n that contains the value sampled from v at phase n in the current period. The variable v_n holds the sampled value until it is used in the next-state function.

State Update The next-state value for stream m_i is computed by $nextSt(exp_i,\ 0)$, where

$$
\begin{array}{lcl}
nextSt(e,\ k) = \text{case } e \text{ of} & & \\
\quad l \mathbin{+\!\!+} e' & \rightarrow & nextSt(e',\ k) \\
\quad drop\ k'\ e' & \rightarrow & nextSt(e',\ k + k') \\
\quad f(e_0,\ e_1,\ \ldots,\ e_n) & \rightarrow & f(nextSt(e_0, k),\ \ldots,\ nextSt(e_n,\ k)) \\
\quad var\ m_j & \rightarrow & \text{if } k < len(a_j) - 1 \\
 & & \quad \text{then } a_j[(k + outIdx_j) \bmod len(a_j)] \\
 & & \quad \text{else } nextSt(exp_j,\ k - (len(a_j) - 1)) \\
\quad cType\ v\ n & \rightarrow & v_n \\
\quad const\ c & \rightarrow & c
\end{array}
$$

Initial State The initial state is computed as follows. For each monitor variable m_i, the initial state of history array $a_i = init(exp_i)\ app\ nextSt(exp_i, 0)$, where

$$
\begin{array}{lcl}
init(e) = \text{case } e \text{ of} & & \\
\quad l \mathbin{+\!\!+} e' & \rightarrow & l\ app\ init(e') \\
\quad \text{otherwise} & \rightarrow & <>
\end{array}
$$

$init(exp_i)$ may produce an empty array, but this array is always augmented by one last index with an initial arbitrary value. Initially, the next-state index points to that last index, while the output index is 0

For each temporary sampling variable v_n, its initial value is $\perp$, pronounced 'undefined', representing the undefined value of a program variable that has not been sampled ($\perp$ is polymorphic and a member of all types).

Example 5. The following are initial values of the history arrays:

specification	history array
$m_0 = [0,\ 1,\ 2] \mathbin{+\!\!+} extW64\ x\ 3 + const\ 3$	$a_i =< 0,\ 1,\ 2,\ \perp >$
$m_1 = var\ m_0 + var\ m_0$	$< 0 >$
$m_2 = drop\ 2\ (var\ m_1)$	$< 4 >$

Scheduling Each monitor variable in a specification has the same period, and each program variable is sampled once each period. Just like in Lustre, new logical clocks can be defined in terms of the underlying period [HCRP91]. This allows control over when variables are sampled. For example, in the following monitor, the program variable x sampled is only used every other period:

$$
\begin{array}{l}
m_0 = [\mathbf{T},\ \mathbf{F}] \mathbin{+\!\!+} var\ m_0 \\
m_1 = \text{if } var\ m_0 \text{ then } (extW8\ x\ 3) \text{ else } var\ m_1
\end{array}
$$

The period for a monitor specification is either provided as an input to the compiler, or the compiler can compute the minimum necessary period. The period must satisfy the following constraint: let n be the largest phase n that appears in a sampling expression (of the form $cType\ v\ n$). Then the period p must satisfy the constraints: $1 < p$ and $n < p$. The first constraint ensures there

are enough ticks per period to perform the actions described below, and the second constraint ensures all the program variables can be sampled within the period. Thus, we have the following order of actions each period:

- Phase 0: apply the state-update function for each monitor variable.
- Phase 1: increment the update and output indexes by 1 mod $len(a_i)$. The output is current for the current period from phase 1 until phase 0 of the next period.

Our algorithm ensures that the output for each stream is updated synchronously, in the same tick.

6.1 Well-Formedness Generalizations

The synthesis algorithm presented is simple and produces efficient code, but it requires two small generalizations to the well-formedness restrictions given in Definition 1. The algorithm guarantees that the Stream Implementation property (Definition 2) is satisfied for any Copilot specification satisfying these the constraints.

- We extend the restriction of no *loop* with a non-negative weight to no *closed walk* with a non-negative weight. Without the extension, the following specification is valid, but it requires pre-computing the next 3 elements of the stream generated by m_0:

$$m_0 = [0] \mathbin{++} var\ m_0 + 1$$
$$m_1 = drop\ 3\ (var\ m_0)$$

  A specification with a closed walk with a non-negative weight that does not contain a loop with a non-negative weight is semantically equivalent to some specification in which all closed walks have negative weights. For example, the following specification is equivalent to the preceding one but does not violate the new restriction:

$$m_0 = [0,\ 1,\ 2,\ 3] \mathbin{++} drop\ 3\ ((var\ m_0) + 1)$$
$$m_1 = drop\ 3\ (var\ m_0)$$

- Let $v_0,\ v_1,\ \ldots,\ v_n$ be a walk of a specification's dependency graph such that v_n is a program variable, and let w be the weight of the walk. Then we require that $w \leq -init(exp_{v_0})$, where exp_{v_0} is the defining expression for monitor variable v_0. The intuition behind this requirement is that our synthesis algorithm does not keep track of previously-sampled values of external variables to be used in stream equations. For example, the following specification violates this condition:

$$m_0 = extW8\ x\ 2$$
$$m_1 = [0,\ 1,\ 2] \mathbin{++} drop\ 1\ (var\ m_0)$$

Our experience is that monitors violating these extended well-formedness constraints are relatively contrived.

6.2 Implementation

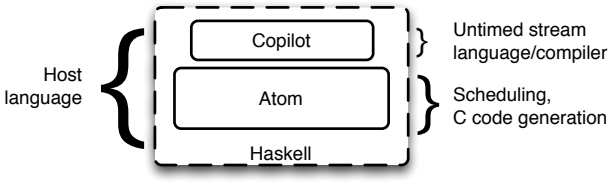


Fig. 2. The eDSL architecture for Copilot.

Copilot is implemented as an *embedded domain-specific language* (eDSL). In the eDSL approach, a DSL is a set of operators defined in a host language. A DSL specification defines data in the host language which can be manipulated; in our case, we rewrite the specification to C code. Because the DSL is embedded, there is no need to build custom compiler infrastructure—the host language's parser, lexer, type system, etc. can all be reused. In particular, the type system of Copilot is embedded in the type system of Haskell, which provides a Hindley-Milner polymorphic type system, extended with type classes [Jon02]. By using a well-tested implementation, we have strong guarantees of correctness, and we can keep the size of the compiler low (3000 lines of code at the time of writing). Finally, in a higher-order host language, one can write combinators over the DSL, acting as a macro system for the language. The architecture of our implementation of Copilot is shown in Figure 2. Copilot uses Atom, an open source eDSL (see Section 2) as an intermediate language that does the C-code generation and scheduler synthesis. Atom performs the schedule generation and optimization (optimization is not described in this paper), too. Informally, the Atom scheduler distributes events across the ticks of a period (without violating causality constraints) to minimize the WCET per tick.

The Copilot compiler has been tested against a simple interpreter on thousands of random streams, which discovered subtle issues, like the additional restrictions on the dependency graph presented in Section 6.1.

We have executed Copilot-generated specifications on the Arduino Duemilanove (ATmega328 microprocessor) as well as on ARM Cortex M3. The monitors generate C99 code, so any processor for which a C compiler exists is a potential target. However, the program's hard real-time guarantees depend on various hardware environmental assumptions; e.g., a cache can break hard real-time guarantees.

We have constructed several small examples to corroborate our design and approach. These examples are drawn from the domain of distributed and fault-tolerant systems and include simple distributed computations, a simple Byzantine agreement protocol, and a simple bus arbiter.

We are currently completing a more substantial case-study involving a fault-tolerant pitot tube sensor (using air pressure for measuring airspeed) on distributed ARM Cortex M3 microprocessors with injected faults.

Copilot will be released open-source (BSD3); please email the authors for an advance copy.

7 Conclusion

Summary In the Introduction, we presented four constraints for a hard real-time monitoring framework: functionality, schedulability, certifiability, and SWaP overhead. We have presented a framework that together satisfies these constraints. In particular, our approach is based on sampling program variables and computing properties over the sampled values. Copilot-generated monitors can be integrated with the observed program without modifying its functionality or real-time guarantees. Finally, no real-time operating system is necessary for scheduling. Our language is a highly-constrained language that makes compilation simple and the ability to statically-compute memory and time usage straightforward. Nevertheless, it is powerful enough to encode typical monitoring formulas, such as past-time LTL and bounded LTL formulas.

Future Work Beyond additional case-studies, one area of future work is to ensure that Copilot monitors are correct. One approach is to borrow from the coinductive verification techniques developed for hardware specifications, since our language is a stream language [Min98]. We have performed initial experiments using Frama-C [Fra] to verify the memory-safety of our generated C code.

We are current developing infrastructure to generate distributed monitors. This allows a global property to be specified for a distributed system, and to distribute the monitors to the system's nodes.

Finally, another topic of interest is to apply statistical techniques to distinguish systematic software faults from transient hardware faults [SLSR07].

Acknowledgements

This work is supported by NASA Contract NNL08AD13T. We thank Ben Di Vito for his direction and input.

References

[ACD+10] E. Axelsson, K. Claessen, G. Dvai, Z. Horvth, K. Keijzer, B. Lyckegrd, A. Persson, M. Sheeran, J. Svenningsson, and A. Vajda. Feldspar: a domain specific language for digital signal processing algorithms. In *8th ACM/IEEE Int. Conf. on Formal Methods and Models for Codesign*, 2010.

[CDR04] F. Chen, M. D'Amorim, and G. Roşu. A formal monitoring-base framewrok for software development analysis. In *Proceedings of the 6th International Conference on Formal Engineering Methods (ICFEM'04)*, number 3308 in LNCS, pages 357–373. Springer-Verlag, 2004.

[DDE08] M.B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 23rd International Conference on Automated Software Engineering*, pages 228–237, 2008.

[DSS+05] B. D'Angelo, S. Sankaranarayanan, C. Snchez, W. Robinson, Zohar Manna, B. Finkbeiner, H. Spima, and S. Mehrotra. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning*, pages 166–174. IEEE, 2005.

[FL09] S. Fishmeister and P. Lam. On time-aware insrumentation of programs. In *RTAS'09: 15h IEEE Real-Time and Embedded Technology and Application Symposium*, 2009.

[Fra] Frama-C. Accessed August, 2010. `http://frama-c.com/index.html`.

[GR04] J. Goossens and P. Richard. Overview of real-time scheduling problems (invited paper). In *Euro Workshop on Project Management and Scheduling*, 2004.

[Hav08] K. Havelund. Runtime verification of C programs. In *TestCom/FATES*, number 5047 in LNCS. Springer-Verlag, 2008.

[Haw08] Tom Hawkins. Controlling hybrid vehicles with Haskell. Presentation. *Commercial Users of Functional Programming* (CUFP), 2008. Available at `http://cufp.galois.com/2008/schedule.html`.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9), September 1991.

[Inc92] RTCA Inc. Software considerations in airborne systems and equipment certification, 1992. RCTA/DO-178B.

[Jon02] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. http://haskell.org/, 2002.

[KLKS04] M. Kim, I. Lee, S. Kannan, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. *Formal Methods in System Design*, 24(1):129–155, 2004.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Min98] Paul Miner. *Hardware verification using coinductive assertions*. PhD thesis, Indiana University, Bloomington, 1998. Adviser-Johnson, Steven D.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer Verlag, 1992.

[PMCR08] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In *RTSS'08: Proceedings of the 29th IEEE Real-Time System Symposium*, pages 481–491, 2008.

[PSM06] L. Pike, M. Shields, and J. Matthews. A verifying core for a cryptographic language compiler. In *Proceedings of the 6th Intl. Workshop on the ACL2 Theorem Prover and its Applications*, pages 1–10. ACM, 2006.

[SLSR07] U. Sammapun, I. Lee, O. Sokolsky, and J. Regehr. Statistical runtime checking of probabilistic properties. In *RV'07: Proceedings of Runtime Verification*, LNS, pages 164–175, 2007.

[ZDG09] H. Zhu, M. Dwyer, and S. Goddard. Predictable runtime monitoring. In *ECRTS'09: 21st Euromicro Conference on Real-Time Systems*, pages 173–183, 2009.