Design and Verification of Multi-Rate Distributed Systems

Wenchao Li
SRI International
li@csl.sri.com

Léonard Gérard
SRI International
leonard.gerard@sri.com

Natarajan Shankar
SRI International
shankar@csl.sri.com

Abstract—Multi-rate systems arise naturally in distributed settings where computing units execute periodically according to their local clocks and communicate among themselves via message passing. We present a systematic way of designing and verifying such systems with the assumption of bounded drift for local clocks and bounded communication latency. First, we capture the system model through an architecture definition language (called RADL) that has a precise model of computation and communication. The RADL paradigm is simple, compositional, and resilient against denial-of-service attacks. Our radler build tool takes the architecture definition and individual local functions as inputs and generate executables for the overall system as output. In addition, we present a modular encoding of multi-rate systems using calendar automata and describe how to verify real-time properties of these systems using SMT-based infinite-state bounded model checking. Lastly, we discuss our experiences in applying this methodology to building high-assurance cyber-physical systems.

I. INTRODUCTION

A cyber-physical system consists of sensors, controllers, actuators controlling the behavior of physical systems operating in real time and space. Examples of such systems can range from surgical robots to planes and power grids. Cyber-physical systems are typically physically distributed. A modern car, for example, can feature 30 to 100 electronic control units (ECUs) for handling functions such as brakes, fuel, engine control, entertainment, skid control, airbags, and windows. The ECUs are connected by means of one or more buses, and the buses themselves are connected through gateways. For the safe and secure operation of the bus, it is important to have an architecture that keeps the nodes decoupled so that the nodes only interact through protected channels with non-blocking communication. We describe the Robot Architecture Definition Language (RADL) and its model of computation. This model of computation consists of nodes operating quasi-periodically at their individual periods and communicating through bounded latency channels. We present the semantics of RADL’s multi-rate, quasi-periodic model of computation and along with a number of useful properties of this model.

Cyber-physical systems typically operate at multiple rates. For example, the sensors operate at different rates given by the device so that a Global Positioning System (GPS) sensor might operate at 10 Hz, whereas an Inertial Measurement Unit (IMU) might operate at 100 Hz. Real-time controllers also operate at different frequencies depending on the desired stability and convergence properties. Monitors for system safety would typically operate at a relatively low frequency compared to the control components. A multi-rate framework is therefore a natural choice for RADL. A multi-rate model also makes sense for security and resilience. Interaction in cyber-physical systems should be decoupled by having components that execute in isolation and communicate through non-blocking channels. Otherwise, it will be possible for one component to either block another one or to overload it with input leading to potential timing failures. The multi-rate model is immune to such failures: when a fast component generates high-frequency input that is received and processed by a slow component at a low frequency, message buffers might be overwritten in the process, but it is not possible to cause the slow component to react faster than its frequency.

Multi-rate designs must be sensitive to message loss since the sender might be sending messages faster than the receiver can process them. There is a bound on the number of consecutive messages that can be lost, and it is also possible to reduce or eliminate message loss by buffering. Cyber-physical systems are control-intensive and often only the latest information is significant, so it is better to design a system that is resilient with respect to bounded message loss. For this purpose, messages should not record events such as button pushes but must instead capture state information such as the toggling of a value. Additionally, state information must not be transmitted in the form of increments, but should instead be represented in absolute form so that the loss of a few messages has little impact on the physical behavior.

The RADL framework takes its inspiration from the Robot Operating System (ROS). In the RADL framework, the sensors, controllers, and actuators are constructed from functional units called nodes. Each node executes independently with a period determined by a local clock and scheduling constraints. RADL supports a publish/subscribe architecture where nodes communicate by publishing on certain topics and subscribing to other topics.

In short, multi-rate architecture is predominant in cyber-physical systems. In this paper, we present a comprehensive solution that addresses various aspects of the design and verification of multi-rate distributed systems. Figure 1 illustrates the overall flow of our proposed approach. The centerpiece of this flow is RADL, which captures both the logical architecture and the physical architecture of the target distributed system. The
logical architecture specifies a model of computation (details in Section II), and drives both model generation for verification and code generation for implementation. On the verification side, a formal model adhering to the RADL specification is generated from user code or Simulink subsystems that describe individual node functions. In the case of Simulink models, the translation is automatic, handled by our tool developed in this work called Sim2SAL. The formal model is further supported by backend tools that enable the verification of real-time properties. On the implementation side, our automatic build system radler synthesizes glue code for the communication layer based on the logical description and binds it to the source code for node functions to generate a code executable. In addition, radler utilizes the physical architecture part of RADL to realize the final system. During the build process, it can instrument the system so that platform assumptions such as node periods and channel latencies can be checked at runtime. This helps to validate the formal model used in verification.

In an earlier publication [16], we proved several theorems about quasi-periodic multi-rate systems. In this paper, we make the following additional contributions.

- Introduce a formal model of computation based on message streams for quasi-periodic multi-rate systems;
- Describe an architecture definition language and its associated build system that supports the design and integration of such systems;
- Present a framework of verifying real-time multi-rate systems based on a modular encoding of such systems using calendar automata;
- Demonstrate the application of the RADL framework in designing a high-assurance ground robot.

The rest of the paper is organized as follows. Section II presents a formalization of the quasi-periodic multi-rate model of computation. Section III describes our architecture definition language, RADL, in detail, and the accompanied build system radler. The verification framework including the encoding of multi-rate systems using calendar automata is given in Section IV. Section V describes an application of the RADL framework to the design and verification of a high-assurance ground robot. We survey related work in Section VI and give concluding remarks and future directions in Section VII.

II. A MODEL OF COMPUTATION FOR QUASI-PERIODIC MULTI-RATE SYSTEMS

A RADL architecture consists of nodes $N_1, \ldots, N_{\text{nodes}}$ and topics $A_1, \ldots, A_{\text{topics}}$. A node $N$ is specified in terms of its minimum and maximum periods $\min(N)$ and $\max(N)$, its list of topic subscriptions $\text{subscribes}(N)$, together with the maximum latency $L(A, N)$ for each topic $A$ in $\text{subscribes}(N)$, and the list of published topics $\text{publishes}(N)$. Each topic $A$ has a type $\text{type}(A)$ and a default value $\text{init}(A)$ of type $\text{type}(A)$. Each topic has exactly one node as its publisher.

The formal semantics of RADL can be summarized as follows. A clock $\tau$ is a monotonically increasing sequence of time points $\tau(i)$, for $0 \leq i$ that is progressive, that is, for any $t > 0$, there is a $j$ such that $\tau(j) > t$. A quasi-periodic clock with period in $[\min, \max]$ with $0 < \min < \max$ and start-up time $t_0 \geq 0$ is a clock $\tau$ where $\tau(0) = 0$, $\tau(1) \leq t_0$, and the period $\min \leq \tau(i + 2) - \tau(i + 1) \leq \max$, for $i \geq 0$. An event $e$ consists of a time $\text{time}(e)$ and a value $\text{value}(e)$. A timed stream $\sigma$ is a sequence of events such that the sequence $\langle \text{time}(\sigma(i)) | i \geq 0 \rangle$ is a clock. A timed stream is quasi-periodic if its clock is quasi-periodic.

For each topic $A$ of type $\text{type}(A)$ with default value $v$, there is a timed stream $\text{stream}(A)$ such that $\text{value}(\text{stream}(A)(0)) = v$. The interpretation is that the value $\text{value}(\text{stream}(A)(i + 1))$ is generated at time $\text{time}(\text{stream}(A)(i + 1))$, for $i \geq 0$.

The communication of messages on a topic stream $A$ to a subscriber node $N$ for the topic is through a mailbox with a bounded latency $L(A, N)$. The maximum latency depends on the subscriber node since some nodes might have more responsive mailboxes than others. Let $\text{rstream}(A, N)$ represent the timed stream of messages received by node $N$ on topic $A$. Then there must be a bijection $I$ on the indices so that $I(0) = 0$, $\text{time}(\text{rstream}(A, N)(0)) = 0$, for each $i \geq 0$, $\text{value}(\text{rstream}(A, N)(i)) = \text{value}(\text{stream}(A)(I(i)))$, and for each $j \geq 1$, $\text{time}(\text{rstream}(A, N)(j)) < \text{time}(\text{rstream}(A, N)(j)) \leq \text{time}(\text{stream}(A)(I(j))) + L(A, N)$. This means that every message on topic $A$ sent at time $t$ is received by node $N$ no later than $t + L(A, N)$. Since $\text{rstream}(A, N)$ is a timed stream, it must have a monotonically (strictly) increasing clock, and hence no two messages are received at the same time. The messages received by $N$ on topic $A$ are therefore totally ordered so that at any time $t$, there is exactly one message that is the last message received at or before $t$. For a timed stream $\sigma$, let $\sigma_i$ represent the finite stream of events $\sigma(0), \ldots, \sigma(n)$, where $t \geq 0$, $\text{time}(\sigma)(n) \leq t$, and $\text{time}(\sigma)(n + 1) > t$.

Each node $N$ has a step function $\text{step}(N, A)$ that maps the input streams from the subscribed topics $\text{subscribes}(N)$ to the values for each published topic $A$ in $\text{publishes}(N)$. Let $\tau_N$ be the quasi-periodic clock of node $N$ operating at a period in $[\min(N), \max(N)]$. Then the clock of stream $\text{stream}(A)$ is equal to $\tau_N$. The value $\text{value}(\text{stream}(A)(i))$ is equal to $\text{step}(N, A)(\text{rstream}(B_1, N)(i), \ldots, \text{rstream}(B_n, N)(i))$, where $t = \tau_N(i)$ and $\text{subscribes}(N) = \{B_1, \ldots, B_n\}$. In other words, the topics published by node $N$ all share the same clock.\footnote{Each step function is assumed to have finished its computation before the node activates in the next period.}

In the typical case, each node will only have a bounded buffer of length $k_i$ for the input messages on each stream. Define $\sigma^k$ as the finite sequence consisting of the last $k$ elements of $\sigma$ if the length of $\sigma$ is at least $k$, and as $\sigma$, otherwise. Then $\text{step}(N, A)(\text{rstream}(B_1, N)(i), \ldots, \text{rstream}(B_n, N)(i))$ can be rewritten using a buffered step $\text{bstep}(N, A)$ as $\text{bstep}(N, A)(\text{rstream}(B_1, N)(i)^{k_1}, \ldots, \text{rstream}(B_n, N)(i)^{k_n})$, where $t = \tau_N(i)$. A buffer size of one means each node
processes the last message received on each topic.

We highlight below several properties about this model of computation that were proven in an earlier publication [16].

1) Bounded processing latency for message: A message sent by publisher node $P$ at time $t$ to subscriber node $S$ on topic $A$ with maximal latency $L(A, S)$ is processed by node $S$ within time $t + L(A, S) + \text{max}(S)$ unless it is superseded by a subsequent message from $P$. The maximal delay occurs when a message sent by $P$ at time $t$ is received by $S$ at a time just after $\tau_S(i)$ and processed by $S$ at $\tau_S(i+1)$ where $\tau_S(i+1) - \tau_S(i) = \text{max}(S)$.

2) No overtaking, with timing assumptions: If $L(A, S) < \text{min}(P)$, then messages are received by $S$ in the order sent by $P$ since the $i$'th message from $P$ will be received by $S$ before the $i+1$'st message is sent.

3) Bounded consecutive message loss: Assuming a buffer size of one, if $M$ is the smallest integer such that $M \times \text{min}(P) > L(A, S) + \text{max}(S)$, then at least one of $M$ consecutive messages is read by the subscriber (assuming no overtaking). The fastest rate at which message can be sent is $1/\text{min}(P)$, and the maximum number of messages that can arrive in the interval from $\tau_S(i)$ to $\tau_S(i+1)$ are those sent in the interval from $\tau_S(i) - L(A, S)$ to $\tau_S(i+1)$.

4) Bounded queue length to eliminate message loss: Under the same assumptions as the previous bullet, with a queue length of $Q$, at most $M - Q$ consecutive messages are lost.

5) Bounded age $MA(m)$ of a message input $m$ used by subscriber $S$ in a step function: $MA(m) < L(A, S) + \text{max}(P)$ (without overtaking). The quantity $L(A, S) + \text{max}(P)$ is the biggest gap between $\text{time} \langle \text{rstream}(P)(i) \rangle$ and $\text{time} \langle \text{rstream}(P)(i+1) \rangle$.

Bounds can also be computed for the scenario where overtaking is possible [16].

These theorems are important for verifying properties of multi-rate systems. Consider the following example of a simple semi-autonomous robot shown in Figure 2. It consists of nodes sensor, operator, controller, and actuator, and topics go published by operator, speed and danger published by sensor, and power and in_danger published by the controller. We can calculate that when the operator sets go to false at time $t$, the controller sets power to 0 within $t + \lambda$, where $\lambda = \text{max} \langle \text{operator} \rangle + L(g, \text{controller}) + \text{max} \langle \text{controller} \rangle$. The explanation for this is that if the operator's action of setting go to false occurs at time $t$, then the operator node might take up to $\text{max} \langle \text{operator} \rangle$ to detect this. The go message might be received by the controller after a delay of at most $L(g, \text{controller})$. The controller processes this message within a delay of at most $\text{max} \langle \text{controller} \rangle$. Similarly, if sensor senses an obstacle and sets danger to true, it may take up to $\text{max} \langle \text{sensor} \rangle + L \langle \text{danger}, \text{controller} \rangle + \text{max} \langle \text{controller} \rangle$ time later for the controller to process this message. These bounds are thus crucial in designing a semi-autonomous robot that can respond timely both to operator and sensor signals. In addition, we need to ensure that the robot can always come to a safe stop even in the setting where go and danger might be produced and arrived out-of-order. We discuss how to formally verify such properties in a multi-rate system later in Section IV.
### III. RADL and Its Build System

**Language Overview.** A RADL specification is a set of typed value definitions following this simple grammar:

\[
definition ::= \text{aggregate} \mid \text{basic}
\]

\[
basic ::= \text{name} \? (? \text{"type"} \? \text{primitive_value})
\]

\[
aggregate ::= \text{name} \? (? \text{"type"} \? \text{"(fieldname value+)"})
\]

\[
value ::= \text{value_definition} \mid \text{name}
\]

As the grammar indicates, naming of a value is not mandatory. This is seen in Figure 3 with \text{PERIOD 5msec} where \text{PERIOD} is the \text{fieldname} and 5msec is a \text{primitive_value} whose implicit type is a \text{duration}. A unit system is implemented on some primitive types like durations which can have \text{sec, msec, usec, nsec} as units.

When a value is to be used multiple times like a topic value, one has to reference it with its name. For example \text{sys_input} is the name of a value of type \text{topic} used in the subscription and publication of nodes \text{sensor} and \text{controller}. Names are scoped and RADL allows the specification to be separated in multiple modules. A system of qualified names is used to reference a name out of the current scope. Some short hands like aliasing and value duplication are also part of the language.

Types and their fieldnames are all predefined by the RADL language, as is the syntax of primitive values. This allows specifications to be typechecked and types for non-annotated values to be inferred. Type ambiguity raises an error and requires the user to annotate the value with the intended type.

The type of RADL values can be separated in two groups, the logical and the physical. The logical specification describes a model of the system in terms of nodes and topics. The physical specification describes the mapping of the logical model on to actual machines.

**Logical Architecture.** The logical specification corresponds to the intended model decided by the user. It has two main value types, \text{node} and \text{topic}. Figure 3 gives a somewhat minimal specification of the simple robot example in Section I.

A topic value consists of the fields of its messages. For example, the \text{sys_input} has two fields named \text{danger} and \text{speed} whose values define the default value of the topic.

A node value has three principal logical fields:

- \text{PERIOD} which gives the node execution period.
- \text{PUBLISHES} which lists the different publications of the node defined through topics.
- \text{SUBSCRIBES} which lists the different subscriptions of the node, where a subscription is defined by a topic and a maximum latency. By default, each subscription is associated with a mailbox (buffer) of size 1, which keeps the last received message. A bigger queue size may be specified here.

**User Code.** At the logical level, nodes are abstract Mealy machines, but an actual implementation is required to enable the system generation. Currently, RADL nodes can be implemented in C, C++ or Simulink\(^1\). The implementation is expected to define a state structure, an initialization function, a step function and a finish function. In Figure 3, every node uses the simplest option of a \text{C++} header file defined with the \text{CX} field. This header needs to contain the declaration of a class named after the \text{CLASS} field, with at least three public methods: a default constructor, default destructor and a method \text{step}, corresponding to the initialization, finish and step function respectively. Accessory source files and libraries required to compile the user code have to be specified in the user code field to allow the system generation to build correctly. An example of user source code for the \text{operator} node of our previous example is shown in Figure 4.

The first thing to note is the inclusion of \text{RADL_HEADER}, a macro defined specifically for each node to allow the inclusion of the required generated header. It is in this header that the step method input (\text{radl_in_t} for messages and \text{radl_in_flags_t} for associated flags) and output (\text{radl_out_t} and \text{radl_out_flags_t}) types are defined according to the RADL specification. In our example, the input structure contains one field named \text{display} corresponding to the \text{display} subscription of the node, itself linked to the topic \text{sys_display} which has one field named \text{in_danger} that will be printed out at every step with the expression \text{in->display.in_danger} where \text{in} is the input structure. The output structure of the step function is implemented in C, C++ or Simulink\(^1\).

\[\text{sys_input : topic \{ FIELDS}
\]

\[
\text{danger : bool true}
\]

\[
\text{speed : float64 0.0}
\]

\[
\text{sys_command : topic \{ FIELDS go : bool false }
\]

\[
\text{sys_output : topic \{ FIELDS power : int32 0 }
\]

\[
\text{sys_display : topic \{ FIELDS in_danger : bool true }
\]

\[
\text{sensor : node \{
\text{PERIOD 10msec
\text{PUBLISHES} ol \{ TOPIC sys_input }
\text{CX} \text{\{ HEADER "sensor.h" CLASS "Sensor" }}\}
\text{controller : node \{
\text{PERIOD 50msec
\text{PUBLISHES} input \{ TOPIC sys_input MAXLATENCY 5msec \}
\text{command \{ TOPIC sys_command MAXLATENCY 10msec \}
\text{PUBLISHES} output \{ TOPIC sys_output \}
\text{display \{ TOPIC sys_display MAXLATENCY 10msec \}
\text{CX} \text{\{ HEADER "controller.h" CLASS "Controller" }}\}
\text{operator : node \{
\text{PERIOD 100msec
\text{PUBLISHES} command \{ TOPIC sys_command \}
\text{SUBSCRIBES} display \{ TOPIC sys_display MAXLATENCY 10msec \}
\text{CX} \text{\{ HEADER "operator.h" CLASS "Operator" }}\}
\text{actuator : node \{
\text{PERIOD 10msec
\text{SUBSCRIBES} output \{ TOPIC sys_output \}
\text{CX} \text{\{ HEADER "actuator.h" CLASS "Actuator" }}\}
\]

Fig. 3. The logical RADL description of a simplified semi-autonomous vehicle example.

\(^1\) Embedded Coder is required for Simulink code generation and full integration is still a work in progress.
used to collect the values to be published as seen by the assignment done to o->command.go.

By forcing the user to use the generated types for input and output structures, we ensure that the types of the fields are correct according to the specification while allowing padding, extra fields, etc. that may be required by the communication layer. The generated header also enables introspection in that it contains every values defined in the RADL specification. This allows the user code to use the specification as parameters. For example, one can access the default speed value of topic sys_input with the C expression RADL_MODULE->sys_input.speed.

**Runtime Monitoring.** The generated glue code and communication layer are in charge of checking the system health. Those checks are designed to be lightweight, conservative and locally computed at the level of each node. Two main checks are done, respective for the node period and the subscribed topic health. According to the Theorem 5 in Section II, any subscription should receive a message at least every \( L(A,S) + \max(P) \). If we define that an input is stale at a step when no new message has been received since the previous step, we can derive from the theorem that an input should not be stale for more than \( \lceil L(A,S)+\max(P) \rceil \) times consecutively. At that point we will say that the input has a timeout flag.

This local information is provided to the nodes in the form of flags associated to each input. There are two categories of flags: the so-called operational and the failure ones. The operational flags are expected to be raised even in normal operation of the system and are meant to be informative for the user. The main one is the stale flags radl_STALE_MBOX, which indicate to the user whether the input is stale. On the other hand, the failure flags point to an abnormal behavior of the system. The main failure flag is the timeout flag radl_TIMEOUT_MBOX, which indicates whether the input in the mailbox has timed out.

Even if these locally computed flags are not checked by a node’s user code, two automatic mechanisms make use of them. The first one is a global health monitoring system, collecting every node flags at a low frequency and reporting it. The second is the automatic flag propagation. We will cover only the second one here.

By default, a message published by a node will be flagged with the conjunction of the input flags of the node. So that if one of the inputs of the node has timed out, all its outputs will be flagged as timeout. At the subscriber level, such a flagged value will have the radl_TIMEOUT_VALUE flag on. In turn, this timeout at the input will cause all outputs to be flagged as timeout, continuing the propagation. The radl_TIMEOUT flag is the conjunction of the radl_TIMEOUT_MBOX and radl_TIMEOUT_VALUE. It can be used by critical nodes to go into a safe mode without relying on the user code of the other nodes to behave well. In our example (Figure 3), by watching the radl_TIMEOUT flag, the actuator node would be able to go into a safe mode (stopping the vehicle) after the sensor node dies, even if the controller node does not check its input flags.

To bypass the default propagation, the step function is given the chance to change the output flags. A node such as the operator node doesn’t rely on its subscriptions to provide an accurate output. Being a source, it can choose how its outputs are flagged. In Figure 4, the user code does always send the same output and make sure that no flag is on. A non-reliable sensor like a GPS would be able to flag its output as stale or even as timeout when no satellites are found, to indicate to its subscriber that the sent value might not be trustworthy.

**Physical Architecture.** The physical specification describes the machines used in the realization of the overall system. A physical description is provided by a value of type plant. The most important field is the MACHINES field listing the machines of the system. A machine is typically defined by the operating system that it runs (OS) fields. Other properties like processor kind and device physical port are not described here. Their use is mostly geared towards system code generation and configuration, done by the radler tool. Three operating systems are currently supported, the Linux Ubuntu 14.04 LTS with recent enough kernel to use the earliest deadline first scheduler [19], and two hypervisors CertiKOS [11] and LynxSecure by the company Lynx Software Technologies. Each operating system has a list of nodes to run under the field NODES. Note that we also support running nodes directly as secure processes of the hypervisor, but this drastically limits the user code since usual system libraries are not available in this setting.

Communication channels between nodes are automatically computed from the plant. The algorithm is straightforward and prioritizes on fast local and secure communication channels. If the two nodes needing to communicate are in the same Linux

---

```cpp
#include <iostream>
#include RADL_HEADER

class Operator {
public:
    void step(const radl_in_t* in, const radl_in_flags_t* in_f, radl_out_t* o, radl_out_flags_t* o_f) {
        std::cout << "In Danger" << in->display.in_danger << std::endl;
        if (radl_is_failing(in_f->display))
            std::cout << "The controller is failing to send correct messages!" << std::endl;
        o->command.go = true; // This basic operator is always trying to go.
        o_f->command = radl_NO_FLAGS; // The output command is fully in our control, clear its flags. }
}
```

---

**Fig. 4.** A rudimentary "operator.h" source file for the operator node of Figure 3
Two machines system : plant {
  MACHINES
  vehicle_box { OS {
    NODES sensor controller actuator
    IP 192.168.0.11 }
  } ocu { OS {
    NODES operator
    IP 192.168.0.10 }}
}

Fig. 5. A basic physical mapping of the example of Figure 3. Two machines running the default operating system are connected by an Ethernet network. One is dedicated to the operator and the other is the computer of the vehicle.

One machine system : plant {
  MACHINES
  main_computer { OS {
    hypervisor : certikos {
      VMS
      operator_vm { OS {
        NODES operator }
      controller_vm { OS {
        NODES sensor controller actuator }}}} }}
}

Fig. 6. A basic physical mapping of the example of Figure 3. Here the operator directly interacts with the computer of the vehicle, but to ensure a higher security, the computer is handled by the CertiKOS hypervisor, isolating the controller part in a separate virtual machine. Communication is implicitly handled by CertiKOS shared memory.

The mathematical model of sections II and IV requires minimum and maximum periods for each node to be specified. The base value is the logical value given by the user in the logical specification. From the base, the minimum and maximum depend on many properties of the machines, from the physical clock drift of the machine, to the task scheduler used in the operating system. Without further specifications in the plant, such properties are assumed to be the default conservative values, for example, a clock drift of 1.8 sec/hr.

System Generation using radler. We have developed radler\(^4\), a tool used to type-check RADL specifications and generate a system ready to be run from it. The main idea is to ensure that the running system is faithful to the model, hence preserving any proven property. The radler system generation remove most of the manual handling by generating the glue code, the overall compilation script with CMake and some key system and hypervisor configuration files. Then, the tool will drive a few external tools to generate a system image. For example, it will compile the CertiKOS kernel with the required configuration. The end result is a packaged system image for each machine, ready to be deployed.

For each machine, a main executable is generated. It is in charge of setting up the system and the communication channels. It then forks into a process per node, handling the node state and scheduling the communications. All the actual communications are handled by a high-assurance library synthesized for RADL by the Kestrel Institute.

The security of the generated system rely on the hypervisors ensuring perfect isolation of virtual machines and protection of shared memory regions. For IP-based communications, the generated Linux VMs are setup to use IPSec, which provides authentication and protects against replay attacks. Other potential vulnerabilities of the system are protected by conservative firewall configurations of the generated Linux OS.

To ease the development process, radler also supports ROS [23] as a backend. When using it, each node can be run independently, communication channels are dynamically setup by ROS. This allows easy logging, debugging, replaying with all the ROS tools. Obviously, all real-time guarantees are lost and precise timing properties might not be preserved.

The radler tool is also able to generate an instrumented version of the system. The instrumentation allows us to test timing properties such as maximum latency of channels, node periods, execution times, etc. The tool also generates local system log that radler can be analyzed and validated against the architecture specification. Such runtime checks are often the only way of validating the timing parameters assumed by the model.

IV. Verification of RADL Systems

In this section, we describe our approach for verifying distributed systems with RADL architectures. The two key elements of our approach are (1) modeling the real-time aspect using an event calendar [10] and (2) modeling the interaction among the quasi-periodic multi-rate components through a careful combination of synchronous and asynchronous compositions. The modeling and verification are done in the Symbolic Analysis Laboratory (SAL) [6], which is a general framework for modeling transition systems and is supported by a suite of tools for verifying both finite-state and infinite-state systems. In the following, we first give an overview of SAL and calendar automata, and then describe how we can use them to model quasi-periodic multi-rate systems. In addition, we show that distributed systems designed in Simulink can be systematically mapped to formal SAL models. This enables the integration of Simulink in a high-assurance tool flow.

Background on SAL and Calendar Automata. SAL is a framework for modeling and specifying transition systems.

\(^4\)The radler tool is in the process of being distributed and open sourced.
It has support for infinite data types such as real numbers and allows real-valued clocks to be modeled. A transition system is a tuple $T = (S, S_0, \theta)$, where $S$ is the (possibly infinite) state space, $S_0 \subseteq S$ is the set of initial states, and $\theta \subseteq S \times S$ is the transition relation. SAL allows a modular way of specifying such systems through a construct called module. Each module is itself a transition system. Modules can be composed synchronously or asynchronously with other modules. We use two commutative and associative operators $|$ and $\parallel$ to denote synchronous and asynchronous compositions respectively. In SAL, state transitions can be expressed as guarded commands $guard \rightarrow assign$, where $guard$ is a Boolean predicate over the current state, and $assign$ is an assignment to the state variables that produces the next state when $guard$ is true. When multiple guards are true simultaneously, the execution model in SAL picks an arbitrary activated command to execute. We note that if none of the guards is satisfied for a module, it will deadlock. In addition, a synchronously composed system is deadlocked if any of its component module is. Later in this section, we will show how this mechanism allows us to schedule the computation of nodes in a quasi-periodic multi-rate setting.

One way to model timed systems is through timed automata [1], whose semantics is typically defined with respect to continuously varying clocks. In a multi-rate setting, however, the computation of nodes is activated periodically. This means time can progress in jumps, matching the activation of the nodes. Therefore, we use a more efficient encoding known as calendar automata (CA) [10] to ensure maximal time progress. As opposed to measuring the elapse of time since a clock is last reset (in timed automata), a calendar stores time progress. As opposed to measuring the elapse of time since a clock is last reset (in timed automata), a calendar stores time progress. A calendar-based system uses the current time $t$ to control when discrete and time-progress transitions (which take positive but bounded logical time) and time-progress transitions (which take zero logical time) and time-progress transitions (which take positive but bounded logical time). A calendar-based system uses the current time $ct$ and the calendar $C$ to control when discrete and time-progress transitions occur. We use $T$ to denote the finite set of timeouts in $C$, that is, $T = \{time(e_1), \ldots, time(e_n)\}$. We use $\delta$ to denote the state of the system that maps $V \cup T \cup \{ct\}$ to appropriate domains, where $V$ is the set of variables used in specifying the function of each subsystem. The respective transition rules are given as follows [10].

- For all initial states, $\delta(ct) \leq \delta(t)$, for all $t \in T$.
- If $\delta(ct) < \delta(t)$ for all $t \in T$, then only a time-progress transition is enabled. Specifically, this transition increases $ct$ to $\min\{\delta(t) \mid t \in T\}$ and leaves all other state variables unchanged.
- If $\delta(ct) = \delta(t)$ for some $t \in T$, then only a discrete transition $\delta \rightarrow \delta'$ is enabled. Specifically, $\delta'(ct) = \delta(ct)$; for all $t \in T$, $\delta'(t) = \delta(t)$ or $\delta'(t) > \delta(ct)$; and there exists a $t \in T$ such that $\delta(t) = \delta(ct)$ and $\delta'(t) > \delta(ct)$.

The last condition ensures that discrete transition occurs only when there is an event $e$ on the calendar $C$ with $time(e) = \delta(ct)$ but also constrains timeout updates to prevent infinite consecutive zero-delay discrete transitions. When several discrete transitions are possible at time $t$, i.e. $|E_t| > 1$, one is selected non-deterministically but all must be performed before $t$ can advance. Note that the inequality $\delta(e) \leq \delta(t)$ for all $t \in T$ is an invariant of the system. In a nutshell, CA is a way of describing dense timed systems without the need of continuously varying clocks.

**Modeling Quasi-Periodic Multi-Rate Systems.** Calendar automata admits a natural encoding of quasi-periodic multi-rate systems, thanks to the inherent time gaps between events. For simplicity, we consider the following special case of the model of computation given in Section II.

- For each node $N$, the step function $step(N, P)$ activated at time $t$ uses only the last element of the input stream $\sigma$. This is equivalent to having a buffer of size one for all message channels.
- No message overtaking is possible, that is, we assume $L(A, S) < \min(P)$. This also means no two messages on the same channel can be received at the same time.

The first condition is motivated by the fact that applications are often only interested in fresh input values. We can also relax this condition to include a gatherer function that maps the current buffered messages to a single input message for the step function. Examples such as taking the median or mean of the last $k$ input sensor values, which are common in control algorithms, fall under this generalization. The second condition is typically ensured by a communication protocol, such as TCP/IP or those that make use of shared memory. It can also achieved by simply attaching a monotonically increasing sequence number to published messages and checking possible reversal of sequence numbers at runtime. Without loss of generality, we assume that the delay of any message is strictly greater than 0 but not bounded below otherwise.

The key elements in modeling a quasi-periodic multi-rate system are listed below.

- Each node is modeled as a discrete transition system, or more specifically a Mealy machine.
- Each node has a local clock that behaves quasi-periodically (see Section II). The output of this clock is the timeout for this node which is tracked by a global calendar. This value is initialized according to a value $t_{ini} \in [0, t_0]$ ($t_0$ is the start-up time for this node).
- The discrete transition system at each node is synchronously composed with its local clock.
- Each bounded channel (for some topic $A$) is modeled as a separate module synchronously composed with its own local clock. A discrete transition of this module is simply

\[\text{transition } \delta \rightarrow \delta' \text{ is enabled. Specifically, } \delta'(ct) = \delta(ct); \text{ for all } t \in T, \delta'(t) = \delta(t) \text{ or } \delta'(t) > \delta(ct); \text{ and there exists a } t \in T \text{ such that } \delta(t) = \delta(ct) \text{ and } \delta'(t) > \delta(ct).\]
the recording of the last published value. Additionally, its clock observes the clock time $t$ of the publisher $N$ and advances its time by a value non-deterministically chosen in the interval $[t, t + L(A, N)]$. Initially, this clock is set to a value in the interval $[t^{min}, t^{min} + L(A, N)]$.

- Each clocked module forms a separate component. All the components are asynchronously composed.
- The calendar is also asynchronously composed with the rest of the system.

Figure 7 illustrates a system of two communicating nodes. In this system, node $M_1$ produces a value $y_1$ at each activation and sends it to $M_2$ through a bounded delay channel. $M_2$ keeps a mailbox of size one and upon activation, uses the value in this mailbox as its input $x_2$. Both nodes are synchronously composed (as shown by the operator $||$) with their respective local clocks. The connection from $M_1$’s clock to the buffer’s clock indicates that the buffer’s clock advances time based on the last activation time of $M_1$. Finally, the three clocked modules are asynchronously composed together (as shown by the operator $|$).

We argue that this encoding of a multi-rate system as a calendar automaton is sound. The synchronous composition of each node with its local clock ensures that its step function is activated according to the quasi-periodic clock. The separately clocked buffer module also follows the mailbox semantics. When a node $N$ publishes a message at time $t$, the fact that the buffer clock advances by a value in the interval $[t, t + L(A, N)]$ ensures that the mailbox receives this message strictly after its publication and not exceeding the latency $L(A, N)$. Message overwrites in the buffer is modeled by the re-assignment of the buffer output to the last received input. Since no message overwriting is possible and the mailbox is of size one, it is sufficient to use a single value in the buffer module to record the possible message of interest in flight. We note that in this model it is possible for the reception of a message and the activation of its subscriber to coincide in time. In this case, the ordering of these two events is non-deterministically chosen, which is consistent with the model of computation given in Section II. Finally, the type information of the messages is not central to this encoding and can be easily preserved.

---

**Sim2SAL: Verification with Simulink Models** Simulink is the prevalent design environment for control engineers, among others. The integration of Simulink into a high-assurance design flow can bring significant benefits especially for safety-critical applications. However, this has been difficult due to the lack of formal semantics, constant evolution of versions, and it being primarily a proprietary system. To facilitate the transition and adoption of Simulink in high-assurance designs, we have developed a tool called Sim2SAL that systematically maps Simulink models to formal SAL models. To the best of our knowledge, this is also the first tool that enables the formal verification of multi-rate systems designed in Simulink.

In Simulink, basic design blocks can be connected and grouped to form a subsystem. Subsystems in turn can be further composed to form larger subsystems. We define a top-level hierarchy for a Simulink design, where each top-level subsystem is intended to be a node in a multi-rate systems. Currently, Sim2SAL handles a subset of Simulink’s discrete blocks. The tool takes as an additional input a logical architecture specification given in RADL. This specification describes the minimum and maximum periods for each node, the latency bound on each input channel as well as its default value. The Simulink model (.mdl file) is first translated into an XML representation using an internal tool developed by Honeywell. Our tool then generates an encoding of the model in SAL using the calendar automata formalism. SAL includes by a suite of verification tools at the backend. We primarily use the infinite-state bounded model checker supported by the Yices SMT solver [9]. The tool also offers an option for abstracting away transmission delays in multi-rate systems, in cases where the delays are negligible. This can significantly reduce the size of the generated SAL model, since each delay channel adds an additional asynchronously composed component. Thus, it offers a tradeoff for scalability with reduced modeling precision. Integrating this verification flow with Simulink’s simulation engine and Embedded Coder creates a holistic design environment for applications that require high-assurance. Sim2SAL has been applied to the design of high-assurance components in the DARPA HACMS project [18] and in a joint project with Honeywell. We are currently working on open-sourcing it to garner wider adoption.

**Verification of Real-Time Properties** Our verification framework primarily revolves around SAL. After encoding the multi-rate system as a calendar automaton in SAL, we use its infinite-state bounded model checker to perform verification. While SAL’s language is expressive enough to model timed transition systems, its requirement specification language is quite limited. In particular, it lacks support for properties involving real-time constraints.

In this section, we show how to verify certain real-time properties by reducing it to an equivalent verification problem involving LTL. In particular, we focus on real-time properties of the form $G(p \Rightarrow F_a q)$, where $p$ and $q$ are state predicates, $G$ and $F$ are the globally and eventually temporal operators in Linear Temporal Logic (LTL) [22], $\mathcal{q} \in \{<,\leq\}$, and $t$ is a real-value constant denoting the time bound. For example,
the requirement of a response must always be received within 12.3 time units of the issuance of a request can be expressed as $G\{\text{request} \Rightarrow F_{\leq 12.3}\{\text{response}\}\}$. In general, extending LTL to model real-time constraint can be done in several ways [4], such as Metric Temporal Logic (MTL) [15] and Timed Propositional Temporal Logic (TPTL) [3]. However, decision procedures for properties expressed in these logics often focus on selected fragments [2], due to either inherent undecidability [3] or efficiency reasons [17]. Our choice of the property pattern, while highly specialized, still captures common properties of interest. We note that this property pattern can be expressed both in MTL and TPTL.

In order to verify this property in SAL, we use two timers $timer_1$ and $timer_2$ to record the events request and response respectively, and then convert the formula into an equivalent LTL formula. Both timers are initialized at $-1$ (or any illegal time value). $\text{trigger}$ is a Boolean signal that is set non-deterministically every time a request event occurs. When $\text{trigger} \land \text{request}$, $timer_1$ records the current time value and the state machine transitions from state $s = 0$ to state $s = 1$. As soon as $s = 1 \land \text{response}$, $timer_2$ registers the current time value and stays at that value forever. We can now verify the equivalent LTL property $G\{timer_1 \geq 0 \Rightarrow F\{timer_2 \geq 0 \land timer_2 - timer_1 < 12.3\}\}$.

The non-deterministic trigger is key to the correctness of this transformation, since it essentially corresponds to a global quantification of the event request on the time line.

V. CASE STUDY

We have applied the RADL framework to the design of a high-assurance ground robot in the DARPA HACMS project [18]. The robot is intended to be tele-operated but also features certain autonomous features such as constant speed cruise control (CCC) and obstacle avoidance. These control modes are arbitrated by a central FSM that determines which controller should be in charge. The robot is equipped with a number of sensors including GPS and SONAR for sensing the distance to an obstacle. The RADL node setup of this robot is shown in Figure 8. Safety and security are the two primary concerns in designing this robot. Specifically, it needs to be resilient to a variety of attacks, such as falsification of values by compromised sensors, jamming and spoofing of teleop communication, and VM-level interference (e.g., a compromised VM not connected in RADL should not be able to interfere with the execution of the other VMs or with the communication between two other VMs). The security protection against these attacks is provided at different layers by various technologies, such a resilient-state estimator, CertiKOS and a high-assurance communication stack. RADL is the umbrella framework that ties these various controllers and mechanisms together. It is also used to generate the overall executable for the robot.

One main safety requirement is that when the robot is under cruise control it should always be able to stop before hitting an obstacle. This is a real-time property depending on how far the robot is away from the obstacle and how fast it is cruising. We use RADL to ensure that the end-to-end latency for the control program is satisfied by the middleware layer. By designing a communication protocol between the FSM and the various controllers (e.g., the CCC controller) we can ensure that important control signals, if sent over consecutive periods, are never missed entirely (per the theorem on bounded message loss). The FSM and its interaction with neighboring nodes, as shown in Figure 8, is formalized using the calendar automata encoding. The model contains 7 main periodic RADL nodes and additional clocked modules as described in Section IV for modeling the mailboxes. Several important properties have been proven, including timely engaging and disengaging of controllers, non-interference of different controller speed outputs, and controller priorities being properly respected (e.g., the teleoperator can override CCC).

We summarize our overall experiences below.

- Formalizing and proving the FSM helped define a uniform protocol for talking with the different controllers.
- The runtime flags provide a safety check almost for free: the robot safely stops whenever a crucial component is removed or killed thanks to the flags being automatically propagated all the way to the actuators.
- The full system generation using RADL (e.g., handling the hypervisor) has significantly improved integration and deployment, which are also important factors in high-assurance designs.

VI. RELATED WORK

Quasi-periodic models of computation have been traditionally employed in cyber-physical system architectures. The first formal study of this model was carried out by Caspi [8] in the context of single-rate systems. He noted that under certain timing assumptions, such systems can be abstracted by an untimed system where one node can have at most two executions between two executions of another node. The RADL model of computation is closely related to the Loosely Time-Triggered Architecture (LTTA) [7] which uses a multi-rate, quasi-periodic nodes, but employs a shared memory bus instead of a publish/subscribe communication architecture.
LTTA has been shown to capture various other models of computation [24]. Stream processing operations can be modeled by ensuring that all of the topic channels are lossless. Kahn Process Networks (KPNs) [13] involve nodes communicating through unbounded FIFO channels with non-blocking writes and blocking reads. KPNs with bounded FIFOs, termed Finite FIFO Platforms (FFPs) can be modeled with a single-rate model with queues of length at most 2 [24].

Timed asynchronous models can be contrasted with the time-triggered approach [14], [20] where clocks are synchronized and the underlying communication bus provides a rigorous bound on message latencies. Time-triggered architectures guarantee determinism since the computation can be organized into an orderly sequence of rounds. Synchronized clocks can be used to mediate access to shared resources, while determinism supports fault-tolerance since non-faulty replicated channels must execute identically.

Synchronous programming languages allow multi-rate operation either at some division of the base clock rate or with independent clocks [12]. Physically Asynchronous Logically Synchronous (PALS) architectures build a synchronous abstraction with clocks that operate at roughly the same rate and are synchronized to a global clock to within a small deviation. PALS has also been extended to handle hierarchical multi-rate systems [5]. PRELUDE [21] is another architecture description language for real-time embedded software. It builds upon synchronous data-flow languages and is designed for mono-processor platforms. RADL is similar to PRELUDE in that both are integration languages that assemble local mono-periodic programs into a globally multi-periodic system. However, RADL serves a different model of computation that we believe are suitable for high-assurance distributed systems. In addition, RADL comes with its dedicated build system (also for multi-core and multi-machine platforms), runtime checks and verification backend.

VII. CONCLUSION AND FUTURE WORK

We have presented the RADL framework for designing and verifying multi-rate distributed systems. This framework includes an architecture definition language supported by a well-defined model of computation, a verification flow that encodes such systems into formally verifiable calendar-automata models, and a build system that generates executables with instrumentation for runtime checks. We argue that RADL is suitable for high-assurance design, where safety and security are the primary objectives. In the future, we aim to extend the static architecture in RADL to accommodate dynamic addition, removal and reconfiguration of nodes.

Acknowledgement. This work is supported by NASA NRA NNA13AC55C, NSF Grant CNS-0917375, and DARPA under agreement number FA8750-12-C-0284. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NASA, NSF, DARPA, or the U.S. Government.

REFERENCES