A Verifiable Language for Programming Real-Time Communication Schedules

Sebastian Fischmeister
University of Pennsylvania, sfischme@seas.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

A Verifiable Language for Programming Real-Time Communication Schedules

Abstract

Distributed hard real-time systems require predictable communication at the network level and verifiable communication behavior at the application level. At the network level, communication between nodes must be guaranteed to happen within bounded time and one common approach is to restrict the network access by enforcing a time-division multiple access (TDMA) schedule. At the application level, the application's communication behavior should be verified to ensure that the application uses the predictable communication in the intended way. Network Code is a domain-specific programming language to write a predictable verifiable distributed communication for distributed real-time applications. In this paper, we present the syntax and semantics of Network Code, how we can implement different scheduling policies, and how we can use tools such as model checking to formally verify the properties of Network Code programs. We also present an implementation of a runtime system for executing Network Code on top of RTLinux and measure the overhead incurred from the runtime system.

Keywords

real-time systems, scheduling, time division multiaccess, networks, software verification and validation.

Comments

A Verifiable Language for Programming Real-Time Communication Schedules

Sebastian Fischmeister, *Member*, *IEEE*, Oleg Sokolsky, *Member*, *IEEE*, and Insup Lee, *Fellow*, *IEEE*

Abstract—Distributed hard real-time systems require predictable communication at the network level and verifiable communication behavior at the application level. At the network level, communication between nodes must be guaranteed to happen within bounded time and one common approach is to restrict the network access by enforcing a time-division multiple access (TDMA) schedule. At the application level, the application's communication behavior should be verified to ensure that the application uses the predictable communication in the intended way. Network Code is a domain-specific programming language to write a predictable verifiable distributed communication for distributed real-time applications. In this paper, we present the syntax and semantics of Network Code, how we can implement different scheduling policies, and how we can use tools such as model checking to formally verify the properties of Network Code programs. We also present an implementation of a runtime system for executing Network Code on top of RTLinux and measure the overhead incurred from the runtime system.

Index Terms—Real-time systems, scheduling, time division multiaccess, networks, software verification and validation.

1 Introduction

Distributed hard real-time systems such as industrial process control, drive-by-wire systems, or hardware-in-the-loop require two properties: predictability and verifiability.

Predictability. Predictability means that an external observer can predict the application's behavior without knowing its internal state. Real-time applications require value and timing control to be predictable in the value and time domains. Failing in one of them could be as disastrous as failing in the other, meaning that a wrong value at the right time could be as bad as the right value at the wrong time.

Predictability for communication means communication with guaranteed bounds on the delivery time. A common approach to provide such bounds is by using time-division multiple access (TDMA). TDMA restricts access to the communication medium by dividing time into slots and assigning slots to individual nodes. If the TDMA schedule is correct and all nodes have synchronized clocks, then there will be no collision in the medium and all communication will happen in a bounded time.

Predictability for outputs means that the application's output is invariant between runs, that is, given the same inputs at the same time, the application will always produce the same outputs. Predictable outputs are also called *value determinism* [1]. A value-deterministic application is not affected by, for instance, scheduling decisions, communication peculiarities such as collisions, race conditions, or operating system tasks such as memory allocation and disk access.

Verification. Claiming predictability without evidence is unsatisfactory for safety-critical applications such as chemical process control. Here, claims need to be verified and, in some applications, verification may even be required by regulatory agencies in the future. Verification assists the evaluation of the system's correctness and is useful in detecting potential problems before they occur in an implemented system.

In our work, we present a programming language to write the communication part of predictable verifiable distributed hard real-time applications. The basic idea is to separate communication and computation code and put communication behavior in a separate layer to which we can apply formal verification techniques. Additionally, this allows the developer to adapt the communication behavior to the application demands and, for example, allow one task to monopolize the communication medium without compromising message deadlines or system integrity.

The programming language, called Network Code, is an executable abstraction for specifying a behavioral model for 1) medium access control (MAC) algorithms for real-time communication, specifically dynamic TDMA, and 2) data access for real-time application. This executable abstraction will then run in an interpreter that executes the schedule as programmed. We achieve the predictability and verifiability of these programs through the construction of layered abstractions, where each layer provides precise guarantees which the higher layers can use to provide their own guarantees.

Fig. 1 shows the layered architecture of our approach. The *dynamic TDMA layer* provides a guaranteed communication within a bounded time and communicates via queues with the upper layers. The *data determinism layer* uses the lower dynamic TDMA layer and provides value determinism for all data which passes through this layer. On top of the layered architecture, the real-time application

• *The authors are with the School of Computer and Information Science, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104. E-mail: sfischme@seas.upenn.edu, sokolsky@cis.upenn.edu, lee@central.cis.upenn.edu.*

Manuscript received 11 Apr. 2006; revised 12 Oct. 2006; accepted 2 Apr. 2007; published online 7 June 2007.
Recommended for acceptance by L. Welch.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0142-0406.
Digital Object Identifier no. 10.1109/TC.2007.70747.

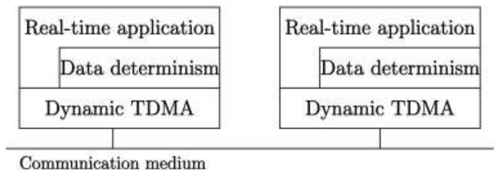


Fig. 1. Overview of the layered architecture used for the runtime system to execute Network Code.

can either use the data determinism layer to have value-deterministic data or directly access the dynamic TDMA layer to send arbitrary data.

The remainder of this paper is structured as follows: Section 2 provides an overview of the model and the system. Section 3 presents the Network Code language. Section 4 defines the concepts behind Network Code. Section 5 shows how we can use our system to implement different policies such as earliest deadline first (EDF), round-robin, or token-based ones. Section 6 elucidates how we use VERSA to verify the code's properties. Section 7 outlines our implementation of the runtime system in RTLinux and shows its performance measurements. Finally, Section 9 presents the conclusions of this work.

2 Overview

Our goal is to create an interpreted language and its runtime system to facilitate the design and implementation of predictable and verifiable communication for distributed real-time applications. To facilitate this, the programming language and the runtime system have to provide control of timing, control of values, control of resources (that is, the shared communication medium), control of dynamic behavior (that is, on-the-fly scheduling decisions), and support for the verification of schedulability.

2.1 Assumptions and the Basic Model

A distributed real-time program consists of a set of periodic preemptible tasks. Tasks communicate with other tasks via channels, which have logical identifiers. Channels logically separate independent message streams. All channels are mapped onto one shared communication medium. A channel must be used by at most one sender for a specified amount of time. The communication medium is accessed in a slotted fashion. Each slot has a defined start time and duration. Tasks use predictable values (*hard values*) and unpredictable values (*soft values*). Hard values are accessed through the data determinism layer and we assume that the access behavior follows repetitive temporal patterns, which are specified in an external document and remain static over the course of the program. Delivery of hard values in the distributed application is guaranteed to happen within bounded time. Access to soft values is arbitrary and bypasses the data determinism layer. Delivery of soft values is not guaranteed within a bounded time. Typically, hard values are used for computation results from tasks, whereas soft values are used for event notification.

Tasks run on processors, which are connected by a shared communication medium. Processors communicate with each other exclusively via the medium that is controlled by the Network Code during runtime; particularly, we assume that there is no shared memory. We assume that time is given in discrete units. Further, we assume the presence of a global

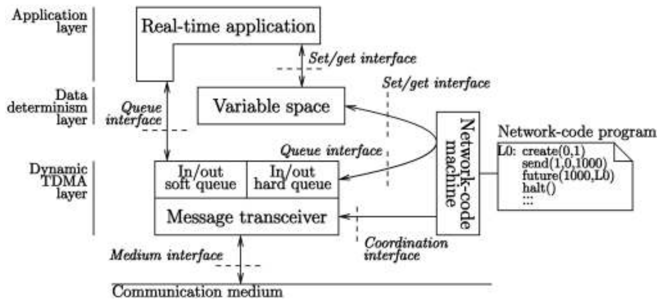


Fig. 2. Overview of the runtime system components, with the interfaces between them for all three layers.

clock and that all times are measured on this clock. The communication medium provides a reliable atomic broadcast service; therefore, either all processors receive a message or none of them do. This is a common assumption for our target domain of embedded systems, which often use single segmented bus networks.

The Network Code program and all auxiliary data structures are generated offline and all necessary communication messages are known in advance. We allow dynamic creation of tasks, but such tasks are limited to accessing soft values only.

The scope of Network Code is to provide language constructs for controlling timing, messaging (sending and receiving), and control flow. Its scope does not include encoding of branching conditions and coordination with the software layer. It also does not include operations on the physical layer; instead, it communicates via queues with a transceiver.

We note that our approach is not necessarily to provide better throughput than specific protocols but to make them verifiable and predictable. It is our aim to provide an effective abstraction for real-time communication which can then act as input for the runtime and the verification system. However, it has been shown in [2] that it can achieve better throughput than protocols with a round-robin allocation.

2.2 System Overview

Fig. 2 shows the concrete system architecture and how we implement and coordinate the different layers shown in Fig. 1. The real-time task sits on top of the architecture. It accesses hard values via a set/get interface from the variable space and it accesses soft values via a queue interface directly from the input/output soft queue. The variable space implements the data determinism layer. It manages a list of timestamped data values $id = \langle ts, val \rangle_i$, where ts and val define the release time and the value, respectively, of the ith entry in this list. The variable space provides control of values as it always releases the most recent value (that is, $get(id, t) = \langle ts, val \rangle_j$, where $ts_j \leq t \wedge \forall i \neq j : ts_i < ts_j$). The soft queues, the hard queues, and the message transceiver implement the dynamic TDMA layer. The soft queue stores messages containing soft values. It consists of an input queue and an output queue, where the input queue contains the received messages of soft values and the output queue contains soft value messages to be transmitted. The hard queues store messages containing hard messages and the system also has an input and output hard queue. The message transceiver provides a coordination

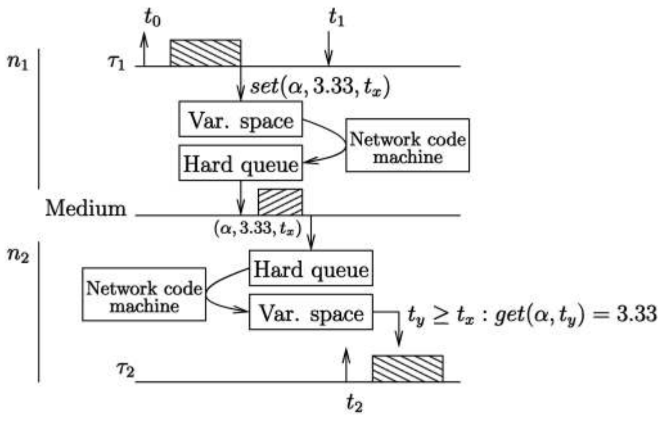


Fig. 3. An example showing the behavior of the runtime system when task τ_1 at node n_1 communicates a value to task τ_2 at node n_2.

interface which provides control of 1) when it is allowed to transmit data and 2) whether it should transmit packets from the soft queue or the hard queue.

The Network Code machine provides timing control and realizes a dynamic behavior of communication schedules by executing a Network Code program. The machine applies timing control onto the variable space and the message transceiver while interpreting the program. It exerts timing control on the message transceiver by calling its coordination interfaces at specific times and it exerts timing control on the variable space by enqueuing and dequeuing messages from the hard queues at specific times. The Network Code machine implements on-the-fly scheduling decisions via guard functions and thereby allows implementing a dynamic behavior in Network Code programs. A guard is similar to the expression of an if clause in traditional programming languages: It decides whether or not the if clause will be taken. Finally, the Network Code program is an abstraction of the communication behavior, which is also amenable to a formal verification and analysis.

2.3 Temporal End-to-End Behavior

Fig. 3 shows the temporal end-to-end behavior of hard values in our system. Node n_1 hosts task τ_1. Task τ_1 produces an output labeled α and stores this value in the variable space via the set interface of the Network Code runtime system. The value becomes valid after time t_x. The Network Code machine reads this new value, creates a message, and stores it in the output hard queue. The message eventually gets transmitted and node n_2 receives it and stores it in its input hard queue. At some time, node n_2's Network Code machine dequeues the message and updates the variable space. The receiving task τ_2 will read α's new value 3.33 if its query time t_y is greater than t_x since α's new value is to be released at time t_x.

We use the above-described mechanism to transparently distribute task output values. As the recipient always receives a valid value from the get operation, the developer does not need to take special precautions for this. The communication schedule is generated in such a way that it can transmit updated values before they are accessed at a remote node. Note, as stated in Section 2.1, that we assume access to hard values to follow a repetitive known temporal pattern.

3 Network Code

Network Code is quite similar to the assembly language and it consists of a series of instructions with a fixed number of parameters. In this section, we present the instruction set and provide an example that demonstrates the potential of the concept and the language.

3.1 Basic Instruction Set

The basic instruction set contains the minimal set of primitives to code precomputed communication schedules. All instructions affect only the local node (for example, destroy frees the message's memory locally only, whereas handle handles an error locally).

Dynamic TDMA Layer. There are four operations that handle timing and MAC:

- **Future.** The instruction future(dl, jmp) schedules a wake-up call for the interpreter. The parameter dl specifies a relative time span for which the interpreter suspends and the parameter jmp specifies where it resumes in the program.
- **Halt.** The instruction halt() halts execution until it resumes.
- **If.** The instruction if(g, jmp) implements a conditional jump. If the guard g evaluates to true, the execution will continue at address jmp. Otherwise, the program counter will increase by one.
- **Mode.** The instruction mode(m) switches between operational modes. Currently, the supported modes are *sched*, *usched*, and *init*. In the *sched* mode, the transceiver uses the hard queues (input and output queue) and, in the *usched* mode, it uses the soft queues. The intention of these two modes, besides the queue selection, is that, whenever hard values are transmitted, the *sched* mode must be used and, for soft values, it is the *usched* mode. We will show in a later example that we can switch from the *sched* mode to the *usched* mode to utilize otherwise unused slots for soft values. The *init* mode invokes the routines to synchronize with other nodes at start time.

Data determinism layer. There are four operations to handle value control. The value layer also uses instructions from the previous layer. Network Code only controls access to the hard queues, whereas the soft queue can be directly accessed by the tasks:

- **Create.** The instruction create(msgid,loc) creates a message from a memory location. The parameter msgid identifies the message to be created. The parameter loc identifies the memory location from which the message's values will be taken.
- **Destroy.** The instruction destroy(msgid) destroys a message. The parameter msgid identifies the message to be destroyed. The message msgid is only accessible at the local node after being created and before being destroyed.
- **Send.** The instruction send(ch, msgid, lifetime) enqueues a message in the hard output queue. The parameter ch specifies the channel on which messages are to be sent and received. The parameter msgid identifies the message to be communicated. The parameter lifetime specifies the message's relative lifetime. The lifetime is the time span during which the message's packets are alive and valid. After expiry of that value, the message can be

cleared from the input buffers. In the normal case, the lifetime of a message is the TDMA slot length. Note that the send instruction needs no parameter for message length or its deadline because we check offline whether these parameters are satisfied and, thus, at runtime, they are no longer needed.

- **Receive.** The instruction receive(ch, loc) retrieves a message from the message queue. The parameter ch specifies the channel from which a message is to be retrieved. The location loc specifies the memory address in the input/output layer to be written to.

Error handling. Our failure model includes integrity errors (for example, errors in the code or the system's software) and network errors (for example, errors while sending and receiving values). Errors are limited to those that are detectable by the runtime system, specifically either while executing an instruction (for example, it tries to receive a message, but the input queue is empty because the sender failed) or while communicating (for example, it tries to send a packet, but the network card reports an error). The model excludes hardware errors that do not manifest as the above ones (for example, memory bit flips or power failures):

- **Handle.** The instruction handle(err, jmp) registers an error handler. The parameter err specifies the error to be handled. The parameter jmp specifies the address at which the execution should immediately continue if an exception occurs. Available errors that can be observed at runtime fall into the categories of integrity, sending, and receiving errors.

  If an error occurs, all queues are flushed and the execution continues at address jmp. If no handler for error err is present, then the execution at the local node stops. Flushing the queues is necessary because, in the event of a failure, nothing meaningful can be assumed about the state. This allows the developer to return to a well-defined state in the local node, which can then be used to return to a well-defined consistent state in the distributed application.

Annotations. The verification framework requires more information than what the Network Code program contains. This additional information is provided via annotation of the programs. Annotations are encoded in comments by using the token "//@ann:". Example annotations are message transmission times or, as described in [2], guard function execution times and labeling of messages for overhead analysis.

3.2 Composite Instructions

The basic instructions can be composed to provide high-level operations. This section presents the ones required for the example in Section 3.3 and an extended list is available in [3]:

- **Goto.** The instruction goto(jmp) implements an unconditional jump. It is emulated by the instruction if and a guard, which always returns true. In the code, the guard *alwaystrue* refers to a function that always returns *true* and which is part of a library of standard guard functions implemented for the language.

```
goto (jmp) =
    if (alwaystrue, jmp)
```

- **Wait.** The instruction wait(dl) pauses the execution for some time. It is simulated by a future instruction with a trigger deadline dl and, subsequently, a halt instruction. The wait instruction augments the schedule's readability. In our source examples, labels for conditional and unconditional jumps are written by a letter followed by a numeral and a colon (for example, "L0:").

```
wait (dl) =
    future (dl, L0)
    halt ()
L0: nop ()
```

- **XSend.** The instruction xsend(ch, msgid, loc, lifetime) creates a message, schedules it for transmission, and then destroys it.

```
xsend (ch, msgid, loc, lifetime) =
    create (msgid, loc)
    send (ch, msgid, lifetime)
    destroy (msgid)
```

- **Backup_send.** The instruction backup_send (primaryCh, backupCh, msgid, lifetime, duration) is a composite instruction for sending a value if the primary send failed. The parameter primaryCh identifies the channel in which a message should be present. The parameter duration specifies the duration for which the node switches into the unscheduled mode. The guard checks whether this message has been transmitted. If it has not, then the instruction will send the message msgid with lifetime lifetime on channel backupCh. If the message has been sent, then the instruction will switch into the usched mode.

```
backup_send (primaryCh, backupCh, msgid,
             lifetime, duration) =
    if (primaryCh_NotEmpty, L0)
    send (backupCh, msgid, lifetime)
    goto (L2)
L0: mode (usched)
    future (duration, L1)
    goto (L2)
L1: mode (sched)
    halt ()
L2: nop ()
```

- **Backup_receive.** The instruction backup_receive (primaryCh, backupCh, loc, duration) performs one of these behaviors: It will switch into the usched mode for the duration duration or receive a message from channel backupCh and store its value in the location loc of the input/output layer. It will do the first if a message is present in channel primaryCh and will do the second otherwise.

```
backup_receive (primaryCh, backupCh, loc,
                duration) =
    if (primaryCh_NotEmpty, L0)
```

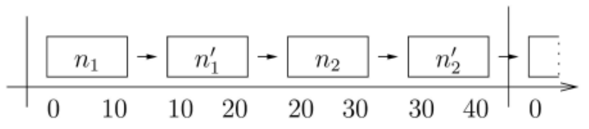


Fig. 4. An example schedule without on-the-fly decisions. Both primary and backup must always communicate their data in each round (for example, both n_1 and n_1').

```
    future (duration, L1)
    goto (L3)
L0: mode (usched)
    future (duration, L2)
    goto (L3)
L1: receive (backupCh, loc)
    halt ()
L2: mode (sched)
    halt ()
L3: nop ()
```

3.3 Example

Network Code programs essentially encode an assignment of slots to particular messages. The program addresses slots the following way: Consider the slot with start time 2 (which is relative to the start of the round) and a duration of 1 time unit. The instructions future and halt are used to suspend the Network Code machine until the start of the slot. The start time is the current time when the last halt has been executed plus the duration specified by future's parameter dl. Assuming that the last halt has been executed at the beginning of the round, the value for dl would be 2 for our example. The slot's duration is encoded in the lifetime of the messages transmitted in the slot. In this example, the instruction send would have the value of 1 for the message's lifetime parameter.

Network Code schedules are more expressive than table-driven ones, where each row represents one communication and this table-driven schedule is executed row by row (c.f., [4]). This additional expressiveness can be utilized to conserve resources [2]. Consider the following scenario: A control system bases its decision on inputs from two nodes. The slot size is 10 time units and, in this example, one communication needs one slot. The application's period is four slots. The two nodes n_1 and n_2 transmit their data. For dependability reasons, each node has a backup node: n_1' for the first node and n_2' for the second. They transmit data if the original node fails to transmit its data. Fig. 4 shows a table-driven schedule that implements this example. Each box represents a communication slot and the label inside identifies the message that the node sends in that specific slot. The table structure prevents on-the-fly decisions, so all four slots are used in each round.

In such a scenario, a table-driven schedule may waste resources. Whenever the original node successfully transmits its message, the backup can remain silent. A schedule with on-the-fly decisions can implement such behavior, as shown in Fig. 5. After each original node's transmission, all nodes determine whether the backup node needs its slot. If not, the slot is available for other traffic. A system that implements such a policy must maintain state information, such as a list of transmitted packets.

Listings 1 and 2 implement Fig. 5's schedule by using Network Code. The symbol "_" masks parameters that are irrelevant for this example. In Listing 1, node n_1' waits for its slot, that is, at time 10, and sends the data, if necessary. Node n_r implements the receiver and follows a similar scheme, as shown in Listing 2. This node receives all transmissions and uses the composite instruction ft_receive to enable an unscheduled communication if the original transmission arrived.

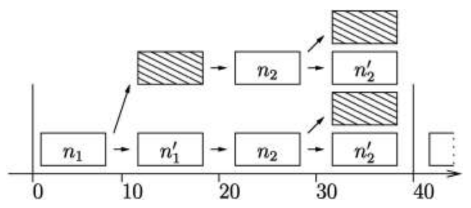


Fig. 5. An example schedule with on-the-fly decisions. If the primary (n_1 or n_2) successfully communicated its data, then the slot assigned to the backups (n_1' and n_2') may be used differently (represented by the hatched box).

Listing 1: communication schedule for node n_1'.

```
L0: wait (10)
    create ($msg_1'$, _)
    ft_send ($ch_1$, $ch_1'$, $msg_1'$, 10, 10)
    destroy ($msg_1'$)
    wait (30)
    goto (L0)
```

Listing 2: communication schedule for node n_r.

```
L0: wait (10)
    receive ($ch_1$, _)
    ft_receive ($ch_1$, $ch_1'$, _, 10)
    wait (10)
    receive ($ch_2$, _)
    ft_receive ($ch_2$, $ch_2'$, _, 10)
    wait (10)
    goto (L0)
```

Note that, in this example, we use redundant nodes and have a fail-over scheme that reacts in the same round. Another scheme could be to store the state across rounds and make a scheduling decision based on which nodes failed to communicate in the last round. Each scheme has different advantages and requires a different level of complexity. The example at hand is intentionally kept simple to show one of the main advantages of Network Code, which is to make scheduling decisions within the communication round.

4 Semantics

In this section, we define an abstract model for TDMA communication schedules with on-the-fly decisions. We call this abstraction a tree schedule. It allows us to show how we use Network Code to implement arbitrary time-triggered schedules and provides the formal semantics that are necessary for the verification of such schedules.

4.1 Abstract Model for TDMA Communication Schedules

A TDMA system consists of a set of connected nodes that communicate via a broadcast medium. The set N includes all of the system's computation nodes n that use the medium. The set of messages M consists of messages m, where m is a tuple $m = \langle \mathrm{src}, v, \mathrm{Dst}\rangle$, with the sender src, a variable v in the variable space, and the set of recipients Dst.

Example 1. Given the schedule shown in Fig. 4, we have $N = \{n_1, n_1', n_2, n_2', n_r\}$ and

$$M = \{m_1 = \langle n_1, \_, \{n_1', n_r\}\rangle, m_2 = \langle n_1', \_, \{n_r\}\rangle,$$
$$m_3 = \langle n_2, \_, \{n_2', n_r\}\rangle, m_4 = \langle n_2', \_, \{n_r\}\rangle\}.$$

The symbol "_" masks unimportant parameters for this example.

A TDMA schedule restricts access to the network to individual nodes by using time division and slots. Such a schedule requires a definition of timestamps and slots. A timestamp t specifies a point in time. A slot $sl = [t_{st}, t_{end})$ is a time interval and consists of the start time t_{st} and the end time t_{end}, with $t_{st} < t_{end}$. The set SL contains all slots sl.

Example 2. Fig. 4's schedule consists of four slots: $sl_1 = [0, 10)$, $sl_2 = [10, 20)$, $sl_3 = [20, 30)$, and $sl_4 = [30, 40)$.

A *linear schedule* $sched = \langle SL, assign\rangle$ consists of a set of slots SL available for scheduling and an injective function $assign : SL \rightarrow M \cup \{\epsilon\}$, with $assign(sl)$ relating a slot to a specific message or leaving it empty. We use the symbol ϵ to denote such an empty slot. A schedule is *nonoverlapping* if

$$\forall sl \in SL : \nexists x \in SL :$$
$$(sl.t_{st} < x.t_{st} < sl.t_{end}) \vee (sl.t_{st} < x.t_{end} < sl.t_{end}).$$

A nonoverlapping schedule is *collision free* because each slot is assigned to at most one node by construction.

Example 3. Given the previous examples, Fig. 4's schedule, defined as $sched_1$, is $(SL_1 = \{sl_1, sl_2, sl_3, sl_4\}, assign_1)$, with

$$assign_1(sl_1) = m_1, assign_1(sl_2) = m_2, assign_1(sl_3) = m_3,$$

and $assign(sl_4) = m_4$.

The mapping assign relates at most one broadcast message with a single slot (injective function). We now define tree schedules, which implement on-the-fly decisions while scheduling. A tree schedule is the tuple $\mathrm{tsched} = \langle \mathrm{SL}, \mathrm{massign}, \Theta, \mathrm{state}\rangle$, where SL is the set of available slots, the noninjective function massign, defined as $\mathrm{massign} : \mathrm{SL} \rightarrow M \cup \{\epsilon\}$, relates multiple broadcasts with a single slot, $\Theta : \mathrm{SL} \times \mathrm{state} \rightarrow sl$ implements a selector function, and state consists of counters and history variables (see Section 4.2). Since, massign is a noninjective function, multiple messages can be assigned to a single slot. The function $\Theta(B, \mathrm{state})$ takes this set of messages and selects one of them. A tree schedule *tsched* is *collision free* if $\forall sl \in SL : |\Theta(\mathrm{massign}(sl), \mathrm{state})| \leq 1$. The tree schedule tsched is also a linear schedule if each slot has at most one assigned communication ($\forall sl \in \mathrm{SL} : |\mathrm{massign}(sl)| \leq 1$). The structure and properties of a tree schedule, as well as probabilistic tree schedules, are described in [2].

Example 4. An example of the function $\mathrm{massign}_1$ for Fig. 5 is

$$\mathrm{massign}_1(sl_1) = \{m_1\}, \mathrm{massign}_1(sl_2) = \{m_2, \epsilon\},$$
$$\mathrm{massign}_1(sl_3) = \{m_3\}, \text{ and } \mathrm{massign}(sl_4) = \{m_4, \epsilon\}.$$

4.2 Network Code as Tree Schedules

In this section, we show that we can map Network Code programs to tree schedules, as defined in Section 4.1. As described in Section 3.3, Network Code addresses slots by using the instructions future and halt. What happens in this slot is then defined between two halt instructions, now called a block. The conversion of a Network Code program to a tree schedule is then straightforward. For a given block, the time at which the Network Code machine enters the block represents the start time of the slot. The messages transmitted in this block represent the messages associated with this slot. The value of the parameter dl of the future instruction inside this block represents the duration of the slot.

Example 5. Considering Fig. 5, we now specify the message $m_1 = \langle n_1, x_1, \{n_1', n_r\}\rangle$ to be communicated in slot $sl_1 = [0, 10)$ by using channel 4. Since the broadcast involves three nodes, we need three network programs. Node n_1 runs the program create(1,x1); send (4,1,10); destroy(1);. Nodes n_1' and n_r run the program wait(10), receive(4,x1).

To implement a linear schedule $\mathrm{sched} = (\mathrm{SL}, \mathrm{assign})$ with Network Code, we first need to define all slots SL and then generate a code for the mapping assign. We can define slots by using the instructions future and halt, as shown above. To realize the mapping assign, we apply a total order to the mapping's domain (that is, slots) and iterate through the ordering. Slots are totally ordered by their start time and $sl_1 < sl_2$ if and only if $t_{st}(sl_1) < t_{st}(sl_2)$. Remember that slots in a linear collision-free schedule are nonoverlapping. As we iterate, we generate the code for the slot (the future and halt instructions) and the code for the defined message (the create, send, destroy, and receive instructions). Finally, we connect all the slots by concatenating them, starting with the first and ending with the last slot.

If $\mathrm{assign}(x) = \epsilon$ for a slot $x \in SL$, then we use the instruction mode to switch from the scheduled into the unscheduled modes. At the slot's end, we switch back into scheduled mode.

To implement a tree schedule

$$\mathrm{tsched} = (\mathrm{SL}, \mathrm{massign}, \Theta, \mathrm{state})$$

with Network Code, we need to define slots SL, generate code for the multislot mapping massign, and implement Θ. state is defined in the runtime. Regarding slots and the mapping massign, we use the same technique described for linear schedules. However, instead of a simple concatenation, in tree schedules, successors for each slot are defined by Θ. We realize Θ by using the conditional instruction if. if executes an arbitrary function whose output is *true* or *false*. The guard functions used in the if instruction are part of Θ. They use the state information state, which consists of counters (for example, instruction counters, message counters, and transmission counters) and history variables (for example, values in the variable space and queue states). To realize Θ, we create a chain of if instructions similar to a switch statement and concatenate one slot with all successor slots by using this construct. In [4], we provide more details about this construct.

Example 6. Listing 1 shows the implemented tree schedule for Fig. 5. The instruction ft_send is the composite instruction, as explained in Section 3.2. The tree structure is implemented in the ft_send instruction which includes a conditional branch that depends on whether the message should be sent or not.

5 Programming MAC Policies

Network Code can model the arbitrary behavior of time-triggered communication schedules as a MAC policy. In this section, we present several different MAC policies and show the Network Code program that implements them. For the sake of brevity, we show the examples without the start-up code, which basically consists of a mode switch from the *init* to the *sched* modes. The client nodes start up in the *init* mode and, when the controller sends the initialization signal, they switch to the *sched* mode.

5.1 Round-Robin Policy

The round-robin scheduling policy assigns communication slots to all nodes in equal portions and in order. For instance, we consider a distributed system with three nodes. In this system, the round starts with node 1 by using one slot, then node 2, then node 3. After this, the round starts from the beginning again.

Listing 3: a round-robin scheduling policy implemented in Network Code.

```
  L0:  if (isMeNext, L3)
2      wait (100)
       receive (1, DATA)
       goto (L0)
  L3:  xsend (1, 1, DATA, 100)
       wait (100)
7      goto (L0)
```

Listing 3 shows the Network Code program that implements the round-robin scheduling policy, with a fixed slot size of 100 time units. The guard in line 1 decides whether it is the local node's turn to communicate. It is a function that guarantees to only return *true* at exactly one node. Thus, if the guard evaluates to true, the node will transmit the data (and jump to label L3); otherwise, it will receive the data.

The guard *isMeNext* can be implemented in different ways. One simple way is to add a counter to the function that implements *isMeNext* and this function returns *true* if $(k \mod n) = m$, with k being a round counter, n being the number of nodes in the network, and m being the identifier of the current node, which is unique for each node.

5.2 EDF Policy

The EDF scheduling policy assigns the next slot based on its priority queue. The tasks get the priority according to the time until their deadline. The lowest value is the highest priority. The priority queue's dequeue operation returns the element with the highest priority. The EDF scheduling policy usually requires preemption. To implement this policy in our system, we restrict preemptions between fixed-sized slots, whose duration we call quantum.

Listing 4: EDF scheduling policy for node n_1.

```
       // ** Distribute priorities
  L0:  xsend (0, 1, MY_DEADLINE, 10)
3      wait (20)
       receive (1, N2_DEADLINE)
       // ** RX/TX data
       if (isMineHighest, L1)
       wait (100)
8      receive (1, 1)
       goto (L0)
  L1:  xsend (3, 1, DATA, 100)
       wait (100)
       goto (L0)
```

Listing 5: EDF scheduling policy for node n_2.

```
       // ** Distribute priorities
  L0:  wait (10)
3      receive (0, N1_DEADLINE)
       xsend (0, 1, MY_DEADLINE, 10)
       wait (10)
       // ** RX/TX data
       if (isMineHighest, L1)
8      wait (100)
       receive (1, 1)
       goto (L0)
  L1:  xsend (3, 1, DATA, 100)
       wait (100)
13     goto (L0)
```

Listings 4 and 5 implement the EDF scheduling policy for a distributed system with two nodes, where the application tasks update the variables that are used to communicate the priorities. Priorities are communicated in slots with a fixed length of 10 time units and the final data is then transmitted in a fixed length slot of 100 time units.

The priority queue is implemented as an array and contains the variables *N1_DEADLINE* and *N2_DEADLINE.* The guard *isMineHighest* compares these two values and returns *true* if the evaluating host has the highest priority. Consider Listing 4. Above line 6, it communicates its priority and receives the other node's priority by using a standard round-robin TDMA policy. In line 6, it determines whether it will communicate in this round. This example can be easily extended for multiple nodes by increasing the number of transmitted deadlines and adapting the guard function.

5.3 Token-Based Policy

Token-based policies use a token to arbitrate access. This virtual token is passed from one node to another and only the node that currently owns the token may communicate. Generally, the algorithm computing the token owner's successor is arbitrary. A well-known algorithm is the token-ring policy for local area networks [5]. It passes the token in a round-robin-like way from one node to the next one. In the domain of real-time communication, the Real-Time Ethernet (RETHER) Project [6] uses token-based arbitration service on top of Ethernet to avoid collisions on the bus and provide a bounded communication time.

Listing 6: a simple token-based policy implemented in Network Code.

```
  L0:  if (haveToken, L1)
2      // code for the recipients
       wait (100)
       if (dataInInputQueue, L4)
       receive (1, TOKEN)
       goto (L0)
7 L4:  receive (1, DATA)
```

```
     goto (L0)
     // code for the sender (token owner)
L1:  if (passToken, L2)
     xsend (0, 1, DATA, 100)
12   goto (L3)
L2:  xsend (0, 1, TOKEN, 100)
L3:  wait (100)
     goto (L0)
```

Listing 6 implements a simple token-based scheduling policy with Network Code. The token is encoded in a variable *TOKEN*. A node owns the token if the variable's value equals the node's identifier. In line 1, each node checks whether it owns the token. If the node owns the token, then it can decide to communicate the data or pass on the token (at label L1). If it communicates the data and keeps the token, then it executes the xsend with *DATA*; otherwise, it executes the xsend with *TOKEN*.

Nodes that do not hold the token receive the data or the updated token value in the receive instruction. The function *dataInInputQueue* decides whether they received the data or the token and, so, each node executes the receive instruction with the correct parameters.

6 Verification

The abstract view of real-time communication provided by Network Code is amenable to formal verification. Given a Network Code program and a message scheduler for each node, we can verify a number of important properties of the schedule. In this section, we present an approach to verification by means of an automatic translation of Network Code into the input for the model-checking tool VERSA [7]. We begin with a brief introduction to VERSA. Then, we discuss our approach for modeling Network Code programs in VERSA and describe the translation algorithm. Then, we discuss what correctness criteria can be checked using VERSA. We construct a formal model that is concerned only with the communication behavior of the system as captured in the Network Code. In this work, we do not model application tasks. Modeling application tasks would allow us to verify additional properties of the communication layer, for example, that no stale values are ever transmitted. At the same time, this would increase the complexity of the model and limit the size of the systems that can be verified.

6.1 Introduction to VERSA

VERSA [7] is a tool for the modeling and analysis of systems with resource and timing constraints. It is based on the real-time process algebra ACSR [8], which allows us to specify the resource requirements of a process and assign timing constraints to its executions. ACSR has been applied to the formal schedulability analysis of complicated task models, as well as to the analysis of safety properties of real-time systems [9]. Below, we present a brief overview of the ACSR language constructs and their informal semantics. For a detailed exposition on ACSR and VERSA, including the specification examples, we refer readers to the earlier published work [8], [9].

The modeling approach of ACSR is to represent a real-time system as a parallel composition of concurrent processes $P_1 \| P_2 \| \dots \| P_n$. The parallel composition is hierarchical in that some of the processes P_i can also contain concurrent processes. Each sequential ACSR process which does not contain the parallel composition within itself is a state machine that can perform two kinds of steps: instantaneous sending or receiving of an event or time-consuming resource access. For readability, we use names starting with a capital letter to denote process names, whereas events and resources have names starting with a lowercase letter. For example, the process P, defined as $e_1?.e_2!.\{r_1, r_2\}.\mathrm{IDLE}$, receives an event e_1, then sends an event e_2, then takes a computation step requiring resources r_1 and r_2, and, finally, becomes an idle process that does not take any steps. As concurrent processes execute, they interact with each other by exchanging events and contending for access to shared resources. For example, two concurrent processes $e?.P_1 \| e!.P_2$ perform a handshake, with the first one receiving the event that the second one sends. Similarly, processes $\{r_1\}.P_1 \| \{r_2\}.P_2$ can proceed together because their computation steps require access to different resources. The handshake that is involved in the synchronous exchange of events is assumed to happen instantaneously, whereas resource access takes time. Conflicts in resource access manifest themselves in models such as deadlocks, which can be detected by exploring the state space of the model. For example, processes $\{r_1\}.P_1 \| \{r_1\}.P_2$ are deadlocked since they both require access to the same resource, and neither provides for an alternative behavior. By contrast, $\{r_1\}.P_1 \| \{r_1\}.P_2 + \{r_2\}.P_2$ is not deadlocked since the second process has an alternative step that uses a different resource.

In modeling Network Code programs, we use ACSR processes in several different ways. There are processes to represent executable entities, such as network instructions and guard bodies, processes that represent system state, such as the message scheduler, and processes for transient entities, such as a message in transit. We use ACSR events to capture the execution of network instructions such as testing guard outcomes, creating messages, and entering them into send queues. These occurrences are instantaneous and fit ACSR events well. We model the bus as the only resource in the model which is shared by all nodes in the network (more complicated bus architectures can be modeled by introducing a separate resource for each bus).

Timing information is associated with ACSR processes in two ways. On one hand, time passage during execution is recorded in resource access steps, specifying how much time an execution fragment takes. Ranges of possible execution times can be specified with the nondeterministic choice of a duration from the given range. On the other hand, time-outs can be associated with execution fragments by specifying a process that serves as a continuation of the execution once the time elapses. ACSR uses the discrete-time approach in which time is represented by integers. Each resource access step takes one unit of time.

An important feature of VERSA, which is used in the modeling of Network Code, is the ability to add and remove processes dynamically. As an example of dynamic process creation, consider the definition $P_s = req?.(P_s \| P_h)$. Intuitively, P_s is a server process that accepts an event req, representing the arrival of some request. When a request arrives, P_s spawns a handler process P_h which asynchronously handles the request, while P_s is free to accept new requests. When P_h completes the processing of a request, it becomes idle. Idle processes are "garbage collected" by the state-space exploration engine.

ACSR modeling is particularly suitable to validate distributed systems that share a communication medium. We utilize the treatment of resources built into the ACSR semantics to arrive at a model that is more concise and more transparent and thus easier to understand and maintain. Network nodes are modeled as processes that access the communication medium as a shared resource when a message is broadcast. If messages overlap, for instance, two nodes are broadcasting messages simultaneously, then VERSA will detect a resource conflict. Here, we take a particular modeling approach that captures the Network Code machine abstraction.

We note that, although ACSR and VERSA are good choices for modeling the framework because of the feature set that they natively support and because of the expertise that we have in using the tool, they are, by no means, the only possible approaches. Many other formal modeling tools can be used, notably the IF model checker [10].

6.2 Modeling Network Code Programs

To encode a faithful representation of Network Code program execution in ACSR, we use the following information: First, for every node in the network, the Network Code that runs on the node needs to be specified. In addition, we need an ACSR process for every guard in the Network Code. Just as guard evaluation is external to the semantics of Network Code, the ACSR representation of a guard is external to the translation process. Finally, we have to consider the time required to transmit the message across the network and we specify this value with an integer value statically.

The overall approach to the translation of Network Code programs into the ACSR process is explained as follows: Suppose that a network consists of n nodes that use m channels to communicate and have k guards that are used in the Network Code. The model of such a network is a parallel composition of processes that represent network nodes: $\mathrm{Node}_1 \| \mathrm{Node}_2 \| \dots \| \mathrm{Node}_n$. Each process Node_i is, in turn, composed of the following processes:

$$\mathrm{Node}_i = \mathrm{NP}_i \| \mathrm{Sch}_i \| \mathrm{Guard}_{i,1} \| \dots \| \mathrm{Guard}_{i,k} \| F_i \| S_i \| D_i.$$

The process NP_i represents the network instruction currently executing at node i. Its evolution is described in detail below. The process Sch_i is the scheduler for the output hard queue. In our current modeling approach, if more than one message is produced simultaneously by the Network Code, one of them is chosen for transmission nondeterministically. However, by making the runtime state of the model more complex, we can experiment with various message-scheduling policies without changing the rest of the model. The guard processes represent the evaluation of guards within the Network Code and are discussed as we consider the representation of the if instruction. The process F_i is a parallel composition of Network Code fragments scheduled for future execution by the future instruction. The process S_i is a parallel composition of processes that represent messages that are scheduled for transmission. Finally, the process D_i is a parallel composition of messages, delivered to the node but not yet processed, that is, it represents the input hard queue of the node. Processes F_i, S_i, and D_i initially do not contain any processes.[1]

1. A parallel composition of an empty set of processes is equivalent to the idle process.

TABLE 1
ACSR Representation of Network Instructions

instr in n_i at a	Run$_{i,a}$
$create(msg, \_)$	$\mathrm{create}_{i,msg}!.\,\mathrm{Run}_{i,a+1}$
$destroy(msg)$	$\mathrm{destroy}_{i,msg}!.\,\mathrm{Run}_{i,a+1}$
$send(ch, msg, t)$	$\mathrm{send}_{i,ch,msg}!.\,\mathrm{Run}_{i,a+1} \Vert \mathrm{Send}_{ch,d(msg),t}$
$receive(ch, \_)$	$\mathrm{receive}_{i,ch}?.\,\mathrm{Run}_{i,a+1}$
$future(d, a_f)$	$\mathrm{Delay}^{d}_{i,a_f} \Vert \mathrm{Run}_{i,a+1}$
$halt()$	IDLE
$if(g_j, a_t)$	$g^{true}_{i,j}?.\,\mathrm{Run}_{i,a_t} + g^{false}_{i,j}?.\,\mathrm{Run}_{i,a+1}$

We view a Network Code program as a linear sequence of instructions. The position of an instruction is its address. An instruction *instr* at address a in the node i is represented as a process $\mathrm{Run}_{i,a}$. The process NP_i at node i is always a process $\mathrm{Run}_{i,a}$ for some address a. Processes in this category are instantaneous, that is, they do not contain time-consuming steps. Table 1 shows the ACSR representation for every instruction in the basic network instruction. The process for the halt() instruction is the idle process, which is removed from further consideration. Processes for other instructions perform an event that corresponds to the executed instruction and becomes the process for the subsequent instruction. For example, in Listing 2 line 2, the instruction receive $(\mathrm{c}_1, \_)$ in the node n_r is represented by using an input event as $\mathrm{Run}_{r,1} = \mathrm{receive}_{r,c_1}?.\mathrm{Run}_{r,2}$. This process will block if no other process can perform the matching output event, that is to say, when the input hard queue of the channel n_1 in the node n_r is empty. If, in the runtime state of the model, $\mathrm{NP}_r = \mathrm{Run}_{r,1}$, the model can take a step such that NP_r becomes $\mathrm{Run}_{r,2}$. In this step, we remove one of the messages from the set D_r, as explained below.

To represent conditional instructions, we introduce an auxiliary process that captures each guard's status. The guard process representing the guard g can send two events, g_{true} and g_{false}, depending on the state of the guard. To execute a conditional instruction, the network-node process has a nondeterministic choice between receiving the two guard events. The choice is resolved by the guard process. Representing guards this way, we significantly simplify the model by making the translation modular.

Two kinds of instructions spawn new concurrent processes. An instruction future(dl,jmp) at address a is represented as $\mathrm{Run}_{i,a} = \mathrm{Run}_{i,a+1} \| Delay^{dl}_{i,jmp}$. The new process $\mathrm{Delay}^{dl}_{i,jmp}$ is added to the set F_i. This process idles for dl time units and then behaves as $\mathrm{Run}_{i,jmp}$. The send instruction send(c,msgid,lifetime) adds a new concurrent process $\mathrm{Send}_{c,l,lifetime}$ to the set S_i. The process represents a message in the output hard queue of node i. The process $\mathrm{Send}_{c,l,lifetime}$ is trying to gain access to the communication medium by interacting with the scheduler process. Once $\mathrm{Send}_{c,l,lifetime}$ gains access to the communication medium, the transmission phase begins. The process performs l resource access steps by using the resource $medium$. The number of these steps is determined by the message transmission time. If another ACSR process enters the transmission phase, then the attempt to use the same resource will lead to a deadlock that signals a scheduling

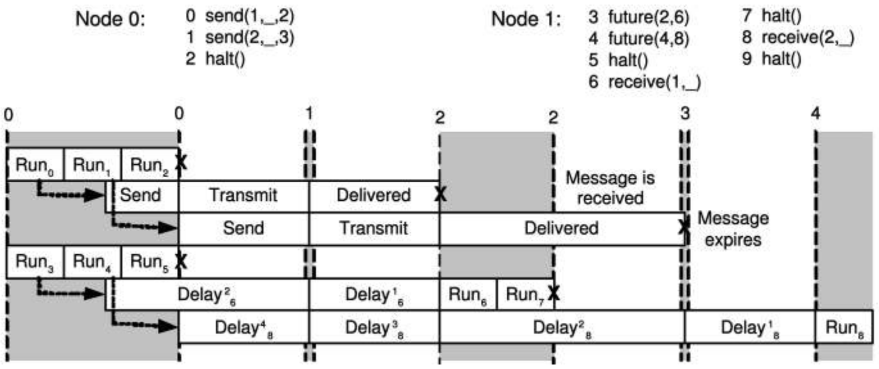


Fig. 6. The execution trace of the VERSA model running the two Network Code programs for Node 0 and Node 1.

problem. Once the transmission phase has completed, the message is delivered to the nodes in the network. This is represented by spawning a new process $\mathrm{Del}_{c,j}$ for each node n_j in the network $(j \neq i)$, which is added to the set D_j. The process $\mathrm{Send}_{c,l,lifetime}$ then becomes idle and is removed from S_i. The process $\mathrm{Del}_{c,j}$ sends the event $\mathrm{receive}_{j,c}$, which synchronizes with the receive instruction in the Network Code program of node n_j (see above). After sending the event, the process becomes idle and is removed from D_j. If the Network Code program does not execute the receive instruction within the message lifetime given by the lifetime parameter, a time-out occurs and the process becomes idle, removing itself from D_j.

6.3 Example

The translation of the Network Code into the collection of ACSR processes is automatically performed by traversing the parse tree of the Network Code in the depth-first manner. We do not present the translation procedure here to save space. Instead, we illustrate the translation by using a simple example.

We run two network nodes. The top part of Fig. 6 shows the Network Code program of both. The lower part of Fig. 6 shows how VERSA interprets the translated program. Horizontal lines represent the evolution of active ACSR processes. Shaded areas represent instantaneous executions. Arrows illustrate how new processes are spawned, whereas a cross denotes that the process has been removed.

The execution proceeds as follows: Node 0 sends two messages simultaneously, each with a transmission length of one time unit. The messages are set to expire after two and three time units, respectively. The transmission scheduler (not shown) releases the first message immediately and the second one is released after the first one is delivered. At time 1, the first broadcast is completed and the other one begins. Node 1 initially schedules two receive instructions: one after 2 time units and the other after 4. The first instruction, executed at time 2, receives the message, but the second fails. At time 4, when the second receive instruction is executed, the second message has already expired. The process Run_8 is blocked and VERSA detects the violation.

6.4 Safety Checks for Network Code

Assumptions and integrity checks. The translation sketched above assumes that the Network Code program in each node is syntactically correct and well structured. Indeed, we can expect that every reasonable Network Code program will satisfy these assumptions. The checks for these assumptions are performed by a parser for the Network Code programs in a straightforward manner. In particular, we assume the following:

1. Every guard referenced in a conditional instruction is defined in the guards section of the program.
2. Every address listed in a conditional or future instruction corresponds to a legal instruction.
3. For every instruction except halt and a conditional instruction with guard *true*, the subsequent address contains a legal instruction.
4. The delay parameter of every future instruction is a positive integer.

We also assume that each execution of a Network Code program reaches a halt instruction in a finite number of steps. This assumption ensures that the model is *non-Zeno*, that is, it cannot perform an infinite number of instantaneous steps in a zero time. This important check is performed by a depth-first search of the graph of dependencies between the $\mathrm{Run}_{a,i}$ processes representing instructions in the Network Code. There is an *immediate* dependency from $\mathrm{Run}_{a,i}$ to $\mathrm{Run}_{a,j}$ if $\mathrm{Run}_{a,j}$ is used in the definition of $\mathrm{Run}_{a,i}$. Immediate dependencies should form an acyclic graph; otherwise, the node is capable of Zeno behavior and should be rejected as invalid. There is a *delayed* dependency from $\mathrm{Run}_{a,i}$ to $\mathrm{Run}_{a,j}$ if $\mathrm{Delay}^{dl}_{j,jmp}$ is used in the definition of $\mathrm{Run}_{a,i}$. The graph of immediate and delayed dependencies is allowed to have cycles as delay instructions enforce time progress and rule out Zeno behaviors. The graph should be connected since connectivity ensures that the Network Code program does not contain a dead code.

Since VERSA uses the discrete-time modeling approach, we also need to determine the granularity of the time step used in the Network Code model. The step size is determined as the greatest common divisor of all time intervals used in the Network Code program, namely, 1) the parameter dl of the future instructions, 2) the parameter lifetime of the send instructions, and 3) message transmission times.

The VERSA model of a Network Code program assumes that all network instructions are executed instantaneously and only message transmission takes time. This assumption is conceptually similar to the *synchrony hypothesis* used in the synchronous languages [11]. This assumption allows us to have a model that is simple to construct and is amenable to an efficient analysis. At the same time, we need to demonstrate that this assumption is justified. This check is also performed by using the graph of immediate dependencies described

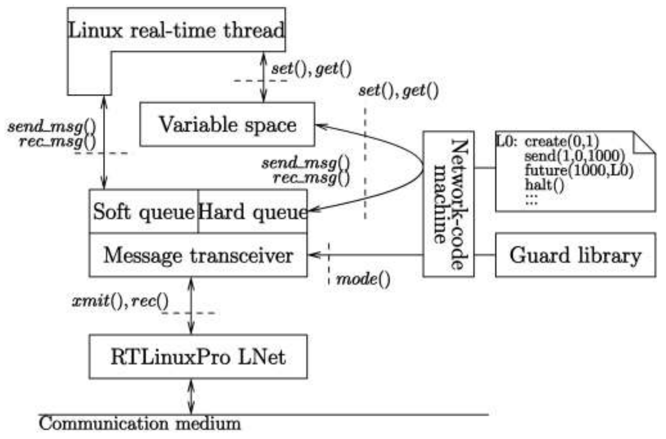


Fig. 7. Overview of the implemented runtime system for executing Network Code on top of RTLinuxPro.

above. We associate the actual runtime of the instruction with every node in the graph. For each connected component in the graph, we determine the value of the longest path. This value represents the maximum execution time for a block of network instructions and should be negligibly small compared to the time step of the VERSA model.

Behavioral checks. This group of checks ensures that the Network Code program in each node handles messages in a sensible manner. Considering a node in isolation permits these checks. Particularly, we check that every message identifier introduced by a create instruction is eventually sent on some channel and then destroyed so that the identifier can be reused. For each message identifier used in the program, we introduce an auxiliary ACSR process that receives the events corresponding to create, send, and destroy instructions and blocks if the events do not come in the expected order. The induced deadlock is detected during the state-space exploration.

Distributed checks. This last group checks whether a collection of nodes can be composed together. The main check in this category is the absence of collisions during transmission. As discussed above, collisions are detected as resource conflicts in the VERSA model which induce a deadlock in a model execution. Two other important distributed checks ensure that an attempt at receiving corresponds to a prior send and that sent messages are received by some node. The first check was also mentioned in the discussion of translation: The Run process for a receive instruction will block if there are no messages to read. For the second check, we introduce an auxiliary process R_c that is spawned when a message on the channel c is sent. The process R_c monitors the receive events sent by the processes $\mathrm{Del}_{c,i}$. If a receive event is observed, R_c becomes idle and disappears. Otherwise, R_c blocks when the validity interval of the message expires.

7 Network Code Runtime System

We implemented a runtime system for Network Code on top of the real-time system RTLinuxPro 2.2. RTLinuxPro (see www.fsmlabs.com) is a hard real-time Posix-compatible operating system. Fig. 7 shows the overall structure of the runtime system.

7.1 Overview

A Linux real-time thread accesses hard values via get() and set() functions from the variable space. The thread accesses soft values by directly sending and receiving messages from the soft queue by using the send_msg() and the rec_msg() interface.

The variable space manages data by storing timestamped variable values. Each variable is a tuple consisting of the point in time when the value becomes valid, its numerical value, and an optional default value which is used if the value is not yet valid. One variable stores several data entries with different values and validity times. For example, the variable space can contain the value tuples $\{x_1 = \langle 5, 1, 0\rangle, x_2 = \langle 7, 2, 0\rangle\}$ for variable x. Variable x's default value is 0, which is returned by the get() function if no valid value is available (for example, in the start-up phase of the system). At time 5, get(x) will return 1 (x_1) and, at time 7, get(x) will return 2 (x_2).

The Network Code machine acts as a bridge between the hard queue and the variable space. It dequeues messages from the input hard queue via rec_msg() and updates values in the variable space via set(). It also creates new messages from values via get() and enqueues these messages via send_msg(). Additionally, the Network Code machine controls the message transceiver by setting its communication mode. The Network Code machine uses a library that stores the guard functions. This library contains all functions which are called in if instructions.

The message transceiver processes the soft and hard queues and receives incoming messages. When the message transceiver is allowed to transmit, it takes one message at a time from the specified queue and passes it to the network driver. The hard and soft queues use a FIFO policy to send messages. The Network Code program is responsible for putting the send into the right order.

To transmit and receive messages on the Ethernet medium, we use the LNet driver provided by FSMLabs. This driver tries to minimize communication input/output jitter and, for example, disables the packet framing mechanism on the network card.

The implementation of the Network Code machine, the variable space, the queues, and the transceiver has about 3,000 lines of code. The application-specific guard functions and the Network Code program add extra lines.

7.2 Performance Measurements

The runtime system introduces overhead to the running system. To quantify this, we measured the instructions' execution times and compute our maximum throughput. The tests were performed on a 100 Mbit network of Intel Pentium 4 with 1.5 GHz, 512 Mbyte RAM, RTLinuxPro 2.2, and a 3c905C-TX/TX-M [Tornado] (rev 78) with exclusive interrupt access. The measurements specify the temporal bounds, with a guarantee of 99 percent and 99.999 percent. If the runtime system violates the temporal bound, then a collision might occur on the network. In case of a collision, the runtime executes the handler registered with the handle instruction and the system continues operating.

Table 2 shows the instruction's execution times in nanoseconds. The upper part of the table shows the execution times of individual instructions plus the timing overhead. The timing overhead cannot be excluded because of the variance, as shown in the instruction nop. Instructions

TABLE 2
Measurements of the Instruction's Execution Times (in Nanoseconds) for the Implemented Runtime System

	Count	Min	Mode	% Obs	99% Obs	99.999% Obs	Max
create	999,964	419	425	14.52	419-2,065	419-7,794	47,943
future	499,981	227	377	24.01	227-471	227-5,577	27,353
handle	999,964	173	179	36.97	176-180	173-5,850	46,589
if	499,982	195	348	13.78	195-502	195-6,117	49,042
nop	499,983	171	174	85.9	173-174	171-5,416	24,993
send	999,964	371	372	39.47	371-733	371-19,090	46,552
ISR	998,248	3,432	3,560	.35	3,432-6,078	3,432-8,963	9,878
LNet	999,964	22,227	22,864	.36	22,227-32,640	22,227-70,868	86,216

not listed in the table have execution times similar to those listed (for example, the mode instruction is similar to the handle instruction). The nop instruction shows the overhead introduced by the timing measurements. The columns show the following data from right to left: *count* shows the sample size, *min* shows the observed minimum value, *mode* shows the most frequent value, *(% obs)* shows the mode's frequency in percent, *99 percent obs* shows the value range for 99 percent of the observations, *99.999 percent obs* shows the same for a larger interval, and, finally, *max* shows the maximum observed value.

The lower part of Table 2 shows the execution times of the runtime system besides executing instructions. The row *ISR* shows the interrupt service routine's (ISR) execution time. Whenever the network card receives a packet, this routine executes and stores the newly arrived packet in a correct input queue. As the statistic shows, this routing usually takes a few microseconds. However, other programs can preempt the ISR, resulting in the ISR's large range. Finally, the row *LNet* shows the execution time of sending a data packet. From similar measurements done for this hardware in [12], we know that the worst-case communication delay is described by the equation $latency = 0.18979 * datasize + 165$, which is in microseconds. This formula has been identified by an empirical latency test of about one million interchanges for data lengths of the Ethernet frame starting at size 30 up to the maximum of 1,500 bytes. The execution times of sending a packet vary according to this equation.

Listing 7: minimal program for sending data (the values for $I1$ and $I2$ can be taken from Table 3).

Node n_1:

```
L0: destroy (0)
    create (0, 0)
    send (1, 0, I1 + I2)
    future (I1 + I2, L0)
    halt ()
```

Listing 8: minimal program for receiving data (the values for $I1$ and $I2$ can be taken from Table 3).

Node n_2:

```
L0: future (I1, L1)
    halt ()
L1: receive (1, 0)
    future (I1 + I2, L0)
    halt ()
```

TABLE 3
Example Code for Calculating the Maximum Throughput and Its Analysis of the Worst-Case Execution Time

Category	Time [ns]	
	99%	99.999%
L0: destroy(0)	180	5,850
create(0, 0)	2,065	7,794
send(1, 0)	733	19,090
future(I1+I2, L0)	471	5,577
halt()	180	5,850
LNet driver	32,640	72,428
Comm. latency	449,000	449,000
ISR n_2	5,605	6,945
Intermediate 1 (=I1)	490,874	498,618
receive (1,0)	1,951	22,247
future(I1+I2)	416	1,928
halt()	162	14,814
Intermediate 2 (=I2)	2,529	38,989
Total cycle (=I1+I2)	493,403	537,607

We will now use the results to calculate the maximum throughput of our runtime system. Listings 7 and 8 show a simple sender and receiver. Table 3 shows the WCET of the code, with the second column displaying the values for a 99 percent guarantee and the third column for a 99.999 percent guarantee. The *Intermediate 1* row shows how long it would take until the packet is delivered to the receiver (node n_2). The *Intermediate 2* row in the table shows how long it would take for the receiver to process the packet. The total cycle time is 493,403 ns with a guarantee of 99 percent and 537,607 ns with a guarantee of 99.999 percent. Thus, the maximum throughput of 1,500 billion packets per second without clock synchronization is 2,025 packets or 2.89 Mbyte/s with a 99 percent guarantee and 1,858 packets or 2.66 Mbyte/s with a 99.999 percent guarantee, which is about 4.23 times slower than the medium's theoretic maximum itself and 3.52 times slower than our empirical average throughput on the same hardware. This calculation shows that the interpretation of Network Code takes about 1.2 percent of the total time for the 99 percent guarantee scenario.

Since we cannot assume a global clock, we implemented a clock synchronization mechanism, as described in [13]. More sophisticated algorithms can be found in [14] and [15]. We measured the precision of our clock synchronization algorithm by using a cross-link connection and a switch. When we tested using a cross-link connection, we could synchronize the clocks of the two machines to within 14 μs. When we used an off-the-shelve D-Link switch between the machines, we could synchronize the clocks to about 24 μs. The difference of about 10 μs comes from the additional jitter of the switch. Comparing these results to the Network Code machine's performance measurements in [3], our algorithm implementation's precision is sufficient to run a Network Code program in a distributed real-time system. Using the clock synchronization, we have to add the worst-case clock jitter to the final value of $I1 + I2$ and see that the synchronization is good enough that the jitter does not seriously affect the throughput. Using clock synchronization and the cross-link connection, the throughput is slowed down by about two-tenths of a percent for the 99 percent guarantee and by one-tenth of a percent for the 99.999 percent.

7.3 Raw TDMA Network

Conventional TDMA is agnostic of the data sent in the slots. It is just concerned with restricting access to a single node for a certain time period. Network Code also supports this type of raw access to the network. The mode *usched* lets anyone access the network. Using this functionality, a task has raw access to the communication medium. In the standard setup, we allow multiple nodes to be in this mode simultaneously to reuse some otherwise unused slots or to specify a slot used for flexible communication (see Fig. 5). However, to provide a collision-free raw access, at most one node must be in this state simultaneously.

The following program shows how we can code raw TDMA without clock synchronization (note that, for a working implementation, clock synchronization packets would need to be sent on a regular basis and both nodes synchronize to start executing at the same time by using the initialization mode). Nodes n_1 and n_2 split up the bandwidth evenly. First, n_2 is allowed to use the medium for 50 time units. Then, n_1 has an exclusive access also for 50 time units. After each TDMA access, we programmed a safety delay of five time units.

Listing 9: node n_1's program for realizing a conventional TDMA schedule only with the soft queue

```
L0: wait (55)
    mode (usched)
    wait (50)
    mode (sched)
    wait (5)
    goto (L0)
```

Listing 10: node n_2's program for the conventional TDMA.

```
L0: mode (usched)
    wait (50)
    mode (sched)
    wait (60)
    goto (L0)
```

This type of access control is also the best choice if the set of active tasks may change at runtime, especially in scenarios with tasks arrivals or with sporadic tasks. The schedule will provide a collision-free raw access and all existing tasks enqueue their messages in the soft queue. However, using this approach, we can only verify the collision-free nature of the schedule and are unable to verify more sophisticated properties, as described in Section 6.

8 Related Work

A number of network protocols and their MAC methods have been published in the literature, each one with different assumptions and for different environments. However, the coding schemes for schedules have been ignored so far. The underlying data structure that is used to represent the TDMA schedule is typically a table. In such a table, one row specifies one communication (that is, one TDMA slot). The table's columns typically identify the sender, the receivers, and the data attributes.

TDMA can be implemented with an immutable communication schedule or with a mutable one. An implementation with an immutable communication schedule is usually referred to as a *static TDMA*. The communication schedule is precomputed and changes at runtime are impossible. This is particularly useful for small-footprint safety-critical systems because it allows static verification and has low computational overhead and generally low complexity. For real-time systems, a number of approaches have been implemented (c.f., [16], [17], [18], and [19]). However, a static TDMA is inflexible: It introduces communication overhead as allocated slots are wasted when a node does not need to send data. The immutable schedule prevents nodes from joining the network once the application is running. Also, it is impractical to compute a static TDMA communication schedule for applications such as real-time video whose exact communication behavior is impossible to predict. Thus, the communication schedule may be mutable and may be cyclically modified to fit new requirements. Such systems are referred to as a *dynamic TDMA*. For real-time systems, a number of approaches have been implemented (c.f., [20], [21], [22], [23], and [24]). A common implementation is that, at the end of one communication round (that is, after the table's last row), a dedicated node computes the new table and broadcasts it across the network. This marks the start of a new round and each node uses the updated communication schedule. Such a system overcomes the limitations described before; however, it adds computational overhead to the node computing the new schedule.

In our approach, we extend the capabilities of immutable schedules to allow on-the-fly changes at runtime. These schedules will be more complex and computationally intensive than the table-driven approach. However, in our design, we can still apply static verification to check the communication behavior.

In [4] and [25], the authors have presented a prior version of Network Code. In contrast to the current version, the previous version included a full-fledged tool chain with a high-level task specification from which a high-level compiler generates Network Code (see [25]). The current version extends this work as

1. its semantics are now formally specified, and the program's behavior can be verified using VERSA,
2. the instruction set is more precise, requires fewer parameters, and consists of solely atomic unambiguous actions,
3. it includes error handling, and
4. splitting the original instruction for sending into the instructions $create()$ and $send()$ provides the developer with more precise control over which values he wants to use in the message.

Several other tools, such as in [26], [27], can be used to formally model Network Code programs and verify schedules. An interesting direction of future work is to try the C interpreter of [26] or the code generation capability of [27] to generate Network Code machine interpreters. The IF toolset [10], which, like VERSA, allows us to associate priorities with processes, would also be effective in representing media access by network nodes. However, we chose VERSA as the verification platform for several reasons. On one hand, dynamic process creation allowed us to capture, in a straightforward way, messages that are pending transmission, along with their deadlines. Since the number of pending messages depends on the state of the schedule, using other means to represent messages leads to a more cumbersome translation. On the other hand, substantial expertise in using VERSA allowed us to implement the Network Code interpreter quickly and efficiently.

9 Conclusion

The goal of our work is to facilitate the design and implementation of predictable and verifiable communication for distributed real-time applications. To meet this goal, we propose Network Code, which is a domain-specific programming language for writing an executable behavioral model for scheduling policies and communication schedules for real-time communication. The programming language is powerful and allows implementing tree schedules, which are more expressive than traditional table-driven communication schedules.

In this paper, we presented the syntax and semantics of Network Code and described a few examples that show different scheduling policies, such as EDF, round-robin, and token-ring, which can be programmed. We also showed that Network Code programs can be verified and elaborated on how we use VERSA to verify safety properties such as collision-free communication, schedulability, and guaranteed message reception. To support the language, we implemented a runtime system for network code on top of a real-time operating system. In this work, we present and discuss the implementation and show that Network Code can be realized with a low system overhead.

For future work, we plan to extend the concept of tree schedules to nonisochronous schedules and to fault-tolerant and multisegmented networks.

Acknowledgments

This research was supported in part by US National Science Foundation Grants CNS-0410662, CNS-0509327, and CNS-0509143, US Army Research Office Grant W911NF-05-1-0182, and OEAW APART-11059.

References

[1] T.A. Henzinger and C.M. Kirsch, "The Embedded Machine: Predictable, Portable Real-Time Code," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI)*, pp. 315-326, 2002.

[2] M. Anand, S. Fischmeister, and I. Lee, "An Analysis Framework for Network-Code Programs," *Proc. Sixth Ann. ACM Conf. Embedded Software (EmSoft '06)*, 2006.

[3] S. Fischmeister, O. Sokolsky, and I. Lee, "Appendix: Programmable Real-Time Communication Schedules," technical report, Univ. of Pennsylvania, 2005.

[4] S. Fischmeister, "Multi-Dimensional Schedules for Media-Access Control in Time-Triggered Communication," *Proc. 10th IEEE Symp. Computers and Comm. (ISCC '05)*, 2005.

[5] D. Clark, K. Pogran, and D. Reed, "An Introduction to Local Area Networks," *Proc. IEEE*, vol. 66, pp. 1497-1517, Nov. 1978.

[6] C. Venkatramani and T. Chiueh, "Design, Implementation, and Evaluation of a Software-Based Real-Time Ethernet Protocol," *Proc. Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '95)*, pp. 27-37, 1995.

[7] D. Clarke, I. Lee, and H.-L. Xie, "VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems," *J. Computer and Software Eng.*, vol. 3, pp. 185-215, Apr. 1995.

[8] I. Lee, P. Brémond-Grégoire, and R. Gerber, "A Process Algebraic Approach to the Specification and Analysis of Resource-Bound Real-Time Systems," *Proc. IEEE*, pp. 158-171, Jan. 1994.

[9] J.-Y. Choi, I. Lee, and H.-L. Xie, "The Specification and Schedulability Analysis of Real-Time Systems Using ACSR," *Proc. 16th IEEE Real-Time Systems Symp. (RTSS '95)*, Dec. 1995.

[10] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, "Tools and Applications: The IF Toolset," *Proc. Fourth Int'l School on Formal Methods for the Design of Computer, Comm., and Software Systems: Real Time (SFM-04:RT)*, M. Bernanrdo and F. Corradini, eds., 2004.

[11] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P.L. Guernic, and R. de Simone, "The Synchronous Languages Twelve Years Later," *Proc. IEEE*, special issue on embedded systems, vol. 91, pp. 64-83, Jan. 2003.

[12] M. Anand, S. Fischmeister, and J. Kim, "Distributed Code Generation from Hybrid Systems Models for Time-Delayed Multirate Systems," *Proc. Fifth ACM Conf. Embedded Software (EmSoft '05)*, 2005.

[13] H. Kopetz, "TTP/A The Fireworks Protocol," *Proc. SAE Int'l Congress and Exposition*, Feb. 1995.

[14] J. Elson, L. Girod, D. Estrin, B. Simons, J. Welch, and N. Lynch, "An Overview of Clock Synchronization," *Fault-Tolerant Distributed Computing*, B. Simons and A. Spector, eds., pp. 84-96, Springer, 1990.

[15] E. Anceaume and I. Puaut, "Performance Evaluation of Clock Synchronization Algorithms, " technical report, INRIA, Oct. 1998.

[16] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic, 1997.

[17] S. Eberle, C. Ebner, W. Elmenreich, G. Färber, P. Göhner, W. Haidinger, M. Holzmann, R. Huber, R. Schlatterbeck, H. Kopetz, and A. Stothert, *Specification of the TTP/A Protocol v2.00*, Research Report 61/2001, Institut für Technische Informatik, Technische Universität Wien, pp. 1-3/182-1, 2001.

[18] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, "Time Triggered Communications on CAN (Time Triggered CAN-TTCAN)," *Proc. Seventh Int'l CAN Conf.*, 2000.

[19] F. Consortium, *FlexRay Communications System—Protocol Specification*, version 2.0, June 2004.

[20] J. Ferreira, P. Pedreiras, L. Almeida, and J. Fonseca, "The FTT-CAN Protocol for Flexibility in Safety-Critical Systems," *IEEE Micro*, vol. 22, no. 4, pp. 46-55, July-Aug. 2002.

[21] P. Pedreiras, L. Almeida, and P. Gai, "The FTT-Ethernet Protocol: Merging Flexibility, Timeliness and Efficiency," *Proc. 14th Euromicro Conf. Real-Time Systems (ECRTS '02)*, pp. 134-142, June 2002.

[22] "Ethernet Powerlink: Data Transport Services,"white paper, Bernecker + Rainer Industrie Elektronik, fifth ed., Sept. 2002.

[23] F. Hanssen and P. Jansen, "Real-Time Communication Protocols: An Overview," technical report, Centre for Telematics and Information Technology, 2003.

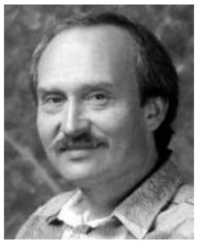
[24] J. Kiszka, B. Wagner, Y. Zhang, and J. Broenink, "RTnet—A Flexible Hard Real-Time Networking Framework," *Proc. 10th IEEE Int'l Conf. Emerging Technologies and Factory Automation (ETFA '05)*, 2005.

[25] G. König, "Using Interpreters for Scheduling Network Communication in Distributed Real-Time Systems," master's thesis, Salzburg Univ., Mar. 2005.

[26] P.C. Ölveczky and J. Meseguer, "Specification and Analysis of Real-Time Systems Using Real-Time Maude," *Proc. Fundamental Aspects of Software Eng. (FASE '04)*, 2004.

[27] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi, "Times: A Tool for Schedulability Analysis and Code Generation of Real-Time Systems," *Proc. First Int'l Workshop Formal Modeling and Analysis of Timed Systems (FORMATS '03)*, P. Niebert and K.G. Larsen, eds., pp. 60-72, 2004.

Sebastian Fischmeister received the Dipl-Ing degree in computer science from the Vienna University of Technology, Vienna, Austria, in 2000 and the PhD degree in computer science from the University of Salzburg, Austria, in December 2002. He is a research associate in the Department of Computer and Information Science at the University of Pennsylvania. His primary research interests include software technology and distributed systems for real-time embedded systems. He is a member of the IEEE and the IEEE Somputer Society.



Oleg Sokolsky received the MSc degree in computer science from St. Petersburg Technical University, Russia, in 1988 and the PhD degree in computer science from the State University of New York at Stony Brook, in 1996. He is a research assistant professor in the Department of Computer and Information Science at the University of Pennsylvania, where he has occupied various research staff positions since 1998. His primary research interests include formal methods for the analysis of real-time and hybrid systems, architectural modeling for embedded systems, and runtime verification. He is a member of the IEEE, the IEEE Computer Society, and the Technical Committee on Real-Time Systems of the IEEE Computer Society.

Insup Lee received the BS degree in mathematics from the University of North Carolina, Chapel Hill, in 1977 and the PhD degree in computer science from the University of Wisconsin, Madison, in 1983. He is the Cecilia Fitler Moore Professor of Computer and Information Science. He joined the faculty of the Department of Computer and Information Science at Pennsylvania State University in 1983 and was the CSE Undergraduate Curriculum Chair from 1994 to 1997. He holds a secondary appointment in the Department of Electrical and Systems Engineering. He was the chair of the IEEE Computer Society Technical Committee on Real-Time Systems and an IEEE CS Distinguished Visitor Speaker. He is a member of the Technical Advisory Group, President's Council of Advisors on Science and Technology, Networking and Information Technology. He has served on many program committees and chaired several international conferences and workshops, including RTSS, RTCSA, ISORC, CONCUR, EMSOFT, and HCMDSS/MD PnP Interoperability. He has also served on various steering and advisory committees of technical societies, including the Steering Committee of ACM SIGBED and was a cochair of the IEEE CS Technical Steering Committee on Embedded Systems. He has served on the editorial boards of several scientific journals, including the *IEEE Transactions on Computers*, *Formal Methods in System Design*, and *Real-Time Systems Journal*. His research interests include real-time systems, embedded and hybrid systems, formal methods and tools, medical device systems, wireless sensor networks, and software engineering. The major theme of his research activities has been to assure and improve the correctness, safety, and timeliness of embedded real-time systems. He has been transitioning research results to practice by applying them to safety-critical real-time systems and high-confidence medical devices. He has published widely and received the Best Paper Award from RTSS 2003. He is a fellow of the IEEE and a member of the IEEE Computer Society.