In Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06), 2006.

Network-Code Machine: Programmable Real-Time Communication Schedules

Sebastian Fischmeister, Oleg Sokolsky and Insup Lee
University of Pennsylvania
sfischme@seas.upenn.edu, {sokolsky, lee}@cis.upenn.edu

Abstract

Distributed hard real-time systems require guaranteed communication. One common approach is to restrict network access by enforcing a time-division multiple access (TDMA) schedule.The typical data representation of offline-generated TDMA schedules is table-like structures. This representation, however, does not permit applications with dynamic communication demands, because the table-like structure prevents on-the-fly changes during execution. A common approach for applications with dynamic communication behavior is dynamic TDMA schedules. However, such schedules are hard to verify, because they are usually implemented in a programming language, which does not support verification.

Network code is a behavioral model for specifying real-time communication schedules. It allows modeling arbitrary time-triggered communication schedules with on-the-fly choices, and it is also apt for formal verification. In this work, we present network code and show how we can use a model checker to verify safety properties such as collision-free communication, schedulability, and guaranteed message reception. We also discuss its implementation in RTLinux and provide performance measurements.

1. Introduction

Distributed hard real-time systems such as industrial process control, drive-by-wire systems, or hardware-in-the-loop require guaranteed communication in bounded time. Hard real-time communication provides such a guarantee and implements it by a real-time protocol.

A common strategy for media access control in real-time protocols is to utilize a collision avoidance mechanism such as time-division multiple access (TDMA). TDMA restricts access to the 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 on the medium and all communication will happen in bounded time. TDMA is typically implemented either by static table-driven schedules or a dynamic policy-driven schedule. *Static table-driven schedules* (see [18, 12, 11, 23]) consist of one table, in which one row usually describes one communication (i.e., one TDMA slot). The table's columns identify, for instance, the sender, the receivers, and the data attributes. This approach is static, because the system executes one row after the other and there are no variations. *Dynamic policy-driven schedules* (see for example [24, 4] and a survey [13]) utilize an algorithm, which implements the protocol's media-access-control policy and decides dynamically, which node gets which communication slot. The policy can implement any arbitrary schedule with on-the-fly decisions.

Network code is an executable abstraction for specifying a behavioral model for medium-access control algorithms for real-time communication, specifically TDMA. A virtual machine implemented for real-time Linux permits executing the specification and realizes a distributed real-time system by sending and receiving packets on an Ethernet network.

A major challenge with dynamic TDMA schedules for distributed hard real-time systems is to analyze the schedule and ensure safety properties. Since dynamic TDMA schedules are typically policy-driven, the developer must verify (1) the TDMA access scheme and so (2) its implementation. Verification falls into the following categories:

- *Peer-reviewed and tested implementation.* There are several published and implemented protocols with dynamic behavior such as RETHER [24], PowerLink Ethernet [4], FTT-Can [23], Byteflight [5]. Such protocols' media-access control bases its properties on the peer-review process for publishing and a tested implementation. Neither of these two provides as strong evidence as formal verification.

- *Formally verified protocols.* Some protocols' specifications have been verified using tools such as Uppaal [20] or SPIN [16]. For a list of examples see [25, 19, 1]. Such protocols' media access control

This research is supported in part by NSF CCR-0209024, NSF CNS-0410662, NSF CNS-0509327, NSF CNS-0509143 ARO DAAD19-01-1-0473, ARO W911NF-05-1-0182 and OEAW APART-11059.

provides verified guarantees on the protocol's specification level; however, the implementation might still contain bugs. So this cannot provide as strong evidence for certain properties as a formal verification of the implementation.

- *Formally verified implementation.* Theories such as model-checking [7] and abstract interpretation with tools such as CBMC [8], Blast [14] or SLAM [3] provide an initial step to verify source code (in our case the protocol's sources).

Network code is a peer-reviewed abstraction for modeling real-time communication schedules. In contrast to related approaches, its abstraction is at the level of tasks and communication. By that, it decouples task and communication scheduling, which makes it apt for formal verification of semantic and temporal aspects at that level. We will show in this work how we apply a model-checking tool to verify safety properties.

The paper's remainder is structured as follows: Section 2 provides an overview of the model and the system. Section 3 presents the network code. Section 4 shows how our abstraction maps to communication schedules. Section 5 elucidates how we use VERSA to verify the schedule's properties. Section 6 outlines our implementation in RTLinux and its performance. Section 7 discusses the used concepts and the implementation and, finally, Section 8 draws the conclusions from this work.

2. Overview

This work's goal is to create a runtime system for deterministic, dynamic, and verifiable communication schedules. Such runtime requires (R1) control of timing, (R2) control of values, (R3) control of resources (i.e., the shared medium), (R4) control of dynamic behavior, and (R5) support for verification such as model checking of the schedule. In the remainder of the paper, we will advert to these requirements with the abbreviations (R1-R5).

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 a logical identifier. Tasks communicate necessary and optional data. Necessary data are communicated via variables, synchronized across the distributed system in bounded time. Optional data are communicated via asynchronous messages whenever there is bandwidth left on the communication medium. Task's necessary communication instances are known a priori and are static over the course of the program. To calculate the schedule offline, we forbid dynamic creation of necessary communication instances. Task's optional communication instances are considered dynamic and will not be communicated in bounded time. To have value determinism, necessary data are guaranteed to provide the latest value available in the distributed system.

Tasks run on processors, which are connected by a shared communication medium. Processors communicate with each other exclusively via the medium and our implemented middle ware; in particular, there is no shared memory. We assume that time is given in discrete units. Further, we assume the presence of a global clock and all times are measured on this clock. The communication medium provides an atomic broadcast service, therefore either all processors receive a message or none of them do.

The network-code schedule is generated offline and all necessary communication instances are known thereat. We do allow dynamic creation of tasks, but such task's communication is limited to optional data. We assume a reliable tool chain to generate the runtime.

We note that our approach is not necessarily to provide better throughput than policy-based protocols but to make them verifiable and thus predictable. It is our aim to provide a viable abstraction for real-time communication, which acts as input for the runtime system and the verification system.

2.2. System Overview

Figure 1 shows the concrete system architecture and the specific interfaces. The runtime system consists of two elements: the shared variable space and the network-code machine. The *shared variable space* manages data and provides control of the values (R2) for the real-time applications. Real-time applications access values via get and set functions. The *network-code machine* (NCM) implements the media access control. It executes a verified (R5) network-code program. The code's instructions control access to the communication medium (R3). The code's timing instructions provide control over timing (R1). Real-time applications can spontaneously transmit and receive data using the $send\_msg$ and rec_msg operations (R4). To satisfy R5, we show how we use VERSA, a model-checking tool, to analyze and verify network code programs.

3. Network Code

Network code is an executable abstraction for specifying real-time communication schedules. In the following, we present the instruction set and provide an example.

3.1. Basic Instruction Set

The basic instruction set contains the minimal set of primitives to code communication schedules. A sched-

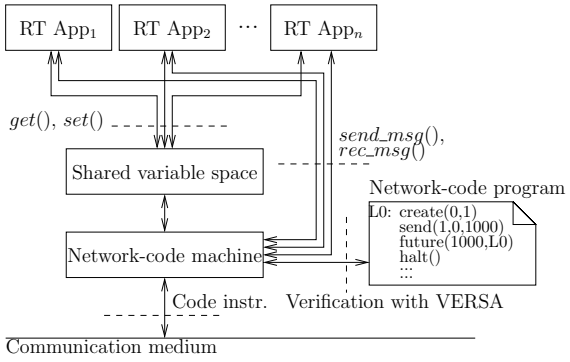

Figure 1. Concrete architecture.

ule is precomputed and does not allow for variables in the program. All instructions affect only the local node (e.g., $destroy$ frees the message's memory locally only).

Communication Control. There are four operations.

- **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 it was created and before it was destroyed.
- **Send.** The instruction $send(ch, msgid, relTime)$ schedules a message transfer. The parameter ch specifies the channel (messages are sent and received on channels). The parameter $msgid$ identifies the message to be communicated. The parameter $relTime$ specifies message's lifetime, i.e., the timespan, which the message's packets are alive and valid.
- **Receive.** The instruction $receive(ch, loc)$ retrieves a message from the message queue. The parameter ch specifies the channel, from which a message will be retrieved. The location loc specifies the memory address in the input/output layer to be written to.

Timing and State Control. There are four operations.

- **Future.** The instruction $future(dl, jmp)$ schedules a wake up call via a trigger. The parameter dl specifies, when the trigger will become enabled. The parameter jmp specifies, at which label the scheduler should start execution.
- **Halt.** The instruction $halt()$ halts the scheduler until a trigger enables it again.
- **If.** The instruction $if(g, jmp)$ implements a conditional jump. If the guard g evaluates to true, the scheduler will continue at address jmp. Otherwise, the program counter will increase by one.
- **Mode.** The instruction $mode(m)$ switches between operational modes. Currently supported modes are *sched*, *usched*, and *init*. The mode *sched* allows only to communicate scheduled data, i.e., the schedule controls access to the medium. The mode *usched* allows unrestricted access to the medium, i.e., the medium is available to all nodes for transmission of packets. The mode *init* invokes the routines to synchronize with other nodes and to start up a distributed real-time system.

Error Handling. Our failure model includes integrity errors (e.g., errors in the code or the system's software) and network errors (e.g., errors while sending and receiving values). The model excludes hardware errors that do not manifest as the above ones (e.g., memory bit flips, power failures). Errors are detected by the runtime system, specifically either the interpreter while it executes an instruction (e.g., it tries to receive a message, but the input queue is empty) or the dispatcher (e.g., it tries to send a packet, but the network card reports an error). The errors are always propagated to the interpreter.

- **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 scheduler should immediately continue execution if an exception occurs. Available errors are: integrity, sending, receiving.

  If an error occurs, all queues are flushed and the scheduler continues at address jmp. If no handler for error err is present, then the scheduler will quit.

3.2. Composite Instructions

The basic instructions can be composed to provide high-level operations. The following presents the ones required for the example. An extended list is available in [10].

- **Goto.** The instruction $goto(jmp)$ implements an unconditional jump. It is simulated by the instruction if and a guard, which always returns true.
- **Wait.** The instruction $wait(dl)$ halts the scheduler 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.

- **Ftasyncsend.** The instruction $ftasyncsend$ $(guaranteedCh, backupCh, msgid, relTime,$ $duration)$ is a composite instruction for sending a value if the original $send$ failed. The parameter $guaranteedCh$ identifies the channel, in which a message should be present. The parameter $duration$ specifies the asynchronous period's duration, if the message needs not be sent. The guard checks whether this message has been transmitted. If it has not, then the instruction will send the message $msgid$ with lifetime $relTime$ on channel $backupCh$. If the message has been sent, then the instruction will switch into $usched$ mode.
- **Ftasyncreceive.** The instruction $ftasyncreceive($ $guaranteedCh, backupCh, loc, duration)$ will lead to one of the following: it will switch into $usched$ 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 $guaranteedCh$, and will do the second otherwise.

3.3. Example

Network-code schedules are more expressive than table-based ones [9]. This property can be exploited to conserve resources. Consider the following scenario: A control system bases its decision on inputs from two nodes. The slot size is ten 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; 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. Figure 2 shows a table-based schedule, which implements this example. Each box represents a communication slot, and the label inside specifies the message, which a node sends in that specific slot. The table structure prevents on-the-fly decisions, so all four slots are used each round.

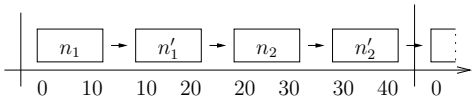

Figure 2. Example schedule (static TDMA).

In such a scenario, a table-based 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 Figure 3. After each original node's transmission, each determines, whether the backup node needs its slot. If not, the slot is available for other traffic. A scheduler, which implements such a policy, must maintain state information such as a list of transmitted packets.

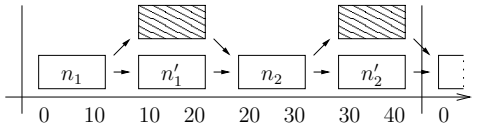

Figure 3. Example schedule (simple).

The listings below implement Figure 3's schedule using network code. Node n_1' waits for its slot, i.e., time 10, and sends the data, if necessary. Node n_r implements the receiver and follows a similar scheme. This node receives all transmissions and uses the composite instruction $ftasyncreceive$ to enable unscheduled communication, if the original transmission arrived.

Node n_1':
```
L0: wait(10)
create (A2,_)
ftasyncsend (n1,n1',A2,10,10)
destroy (A2)
wait (20)

goto (L0)
```

Listing 1. Node n_1'

Node n_r:
```
L0: wait (10)
receive (n1,_)
ftasyncreceive (n1,n1',_,10)
wait (10)
receive (n2,_)
ftasyncreceive (n2,n2',_,10)
goto (L0)
```

Listing 2. Node n_r

4. Semantics

In the following we define an abstract model for TDMA communication schedules with on-the-fly decisions. This allows us to show how we use network code to implement arbitrary time-triggered schedules.

4.1. Abstract Model for TDMA Communication Schedules

A TDMA system consists of nodes and point-to-point communication via a broadcast medium. The set N includes all system's computation nodes n, which use the medium. The set $C = \{(n_i, n_j) | n_i \in N, n_j \in N\}$ contains all the communication instances of the distributed application and the tuple $c = (n_i, n_j)$ describes a point-to-point communication instance c between n_i and n_j in which n_i is the sender and n_j is the receiver. Given a communication instance c, we refer to $c.s$ as the sender and $c.d$ as the destination. The power set $\mathcal{P}(C)$ contains all possible sets of communication instances. A subset $B \in \mathcal{P}(C)$ is a *valid* broadcast communication instance with $\forall c_i, c_j \in B : c_i.s = c_j.s$.

Example 1. *Given the schedule in shown in Figure 2, we have* $N = \{n_1, n_1', n_2, n_2', n_r\}$, $C = N \times N$, *and for*

example $B_1 = \{(n_1, n_1'), (n_1, n_r)\}$, $B_2 = \{(n_1', n_r)\}$, $B_3 = \{(n_2, n_2'), (n_2, n_r)\}$, *and* $B_2 = \{(n_2', n_r)\}$.

A TDMA schedule restricts access to the network to individual nodes using time division and slots. Such a schedule requires a definition of time, time stamp, and slot. A time stamp t specifies a point in time. A slot $sl = [t_{st}, t_{end}) \in SL$ consists of (1) the start time t_{st} and (2) the end time t_{end} with $t_{st} < t_{end}$.

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

A *linear schedule* $sched = (SL, assign)$ consists of (1) a set of slots SL available for scheduling and (2) a mapping $assign : SL \rightarrow B \cup \{\epsilon\}$ with $assign(sl)$ mapping a slot to a specific broadcast instance or leaving it empty, i.e., ϵ. A schedule is *non-overlapping*, 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 non-overlapping schedule is implicitly *collision-free.*

Example 3. *Given the previous examples, Figure 2's schedule is* $sched_1 = (SL_1 = \{sl_1, sl_2, sl_3, sl_4\}, assign_1)$ *with* $assign_1(sl_1) = B_1$, $assign_1(sl_2) = B_2$, $assign_1(sl_3) = B_3$, *and* $assign(sl_4) = B_4$.

The mapping $assign$ relates at most one broadcast communication instance with a single slot. We now define tree schedules, which implement ad-hoc decisions while scheduling. The mapping $massign$ relates multiple broadcasts with a single slot. It is defined as $massign : SL \rightarrow \mathcal{P}(B)$.

Example 4. *The function* $massign_1$ *for Figure 3 is* $massign_1(sl_1) = \{B_1\}$, $massign_1(sl_2) = \{B_2, \epsilon\}$, $massign_1(sl_3) = \{B_3\}$, *and* $massign(sl_4) = \{B_4, \epsilon\}$.

To make decisions, we define a function Θ, which takes a set of broadcasts and selects one of them based on some state $state$; thus, $\Theta : \mathcal{P}(B) \times state \rightarrow B$.

Now we define a tree schedule as $tsched = (SL, massign, \Theta)$. A tree schedule $tsched$ is *collision-free*, if $\forall sl \in SL : \#\Theta(massign(sl), \_) \leq 1$. The classes of tree schedules $tsched$ and linear schedules $sched$ are equivalent, if $\forall sl \in tsched.SL : \#(msched(sl)) \leq 1$.

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.

Network code splits communication instances into a sending and a receiving part. The sending part consists of the instructions $create, send$, and $destroy$; the receiving part consists of $receive$. Each part is executed on the local node. A broadcast communication instance is realized by executing the sending part on one node and executing receiving parts on multiple nodes.

A slot is defined by a start time and an end time. The instructions $future$ and $halt$ allow for specifying such a start time. The slot's start time is computed by: the time stamp, when the last $halt$ has been executed plus the duration specified by $future$'s parameter dl. The slot's end time is computed by: the slot's start time plus the duration specified by $send$'s parameter $relTime$.

Example 5. *We will realize a broadcast* $B_1 = \{(n_1, n_1'), (n_1, n_r)\}$ *in slot* $sl_1 = [0, 10)$. *To do so, we require three network programs and will communicate this data in channel four. Node* n_1 *runs this program:* $wait(0); create(\_, \_); send(4, \_, 10); destroy(\_);$. *Node* n_1' *and* n_r *run this program:* $wait(10), receive(4, \_)$.

To implement a linear schedule $sched = (SL, assign)$ with network code, we first need to define all slots SL and then realize the mapping $assign$. We can define slots using the instructions $future$ and $halt$ as shown above. To realize the mapping $assign$, we apply a total order to the mapping's domain and iterate through. Slots are totally ordered by their start time; $sl_1 < sl_2$, iff $t_{st}(sl_1) < t_{st}(sl_2)$. Remember that slots in a linear schedule are non-overlapping. As we iterate, we generate the code for the slot (the $future$ and $halt$ instructions) and the code for the defined communication instance (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.

Example 6. *Example 5 already realizes a linear schedule* $sched_2 = (\{sl_1\}, assign_2)$ *with* $assign_2(sl_1) = B_1$. *This is a schedule consisting of only one slot.*

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

To implement a tree schedule $tsched = (SL, massign, \Theta)$ with network code, we need to define slots SL, realize the multi-slot mapping $massign$, and implement Θ. Regarding slots and the mapping $massign$, we use the same technique as described for linear schedules. However, instead of a simple concatenation, in tree schedules Θ defines successors for each slot. We realize Θ using the conditional instruction if. if executes an arbitrary function, whose output is $true$ or $false$. This arbitrary function uses the state information supplied by Θ to determine the next slot. To realize Θ, we create a chain of if instructions similar to a switch-statement and concatenate one slot with all successor slots using this construct. In [9], we provide more details about this construct.

Example 7. *Listing 1 shows the implemented tree schedule for Figure 3. The instruction $ftasyncsend$ is a composite instruction and Listing 3 shows its full source.*

```
    ftasyncsend ( guaranteedCh, backupCh,
0                 msgid, val, duration ) =
    if ( #guaranteedChNotEmpty, L0 )
    send ( #backupCh, #msgid, #val )
    goto L2
L0: mode ( usched )
5   future ( #duration, L1 )
    goto L2
L1: mode ( sched )
    halt ()
L2: nop ()
```

Listing 3. Expanded for of composite instruction $ftasyncsend$.

5. Verification

Network code is a viable abstraction for real-time communication schedules. First, we show how we can use the abstraction for in the model-checking tool VERSA [6]. Then we apply VERSA to the abstraction and show what correctness criteria we can check.

5.1. VERSA and NCM Encoding

VERSA [6] is a tool for modeling and analysis of systems with resource and timing constraints. It is based on the real-time process algebra ACSR [21], which allows us to specify resource requirements of a process and assign timing constraints to its executions.

The ACSR modeling approach is to represent a real-time system as a collection of concurrent processes $p_1|p_2|\ldots|p_n$. Each sequential process p_i is a state machine that can perform two kinds of steps: instantaneous sending or receiving of an event, or time-consuming resource access. As concurrent processes execute, they communicate with each other by exchanging events and contending for access to shared resources. The handshake that is involved in the synchronous exchange of events is assumed to happen instantaneously, while resource access takes time.

ACSR modeling is particularly suitable to capture the NCM abstraction. We utilize the treatment of resources built into the ACSR semantics to arrive at a model that is more concise and more transparent – thus easier to understand and maintain. Network nodes are modeled as processes, which access the communication medium. Communication is modeled by resource access, i.e., if a node broadcasts a message, then it will access the single resource communication medium. If messages overlap, e.g., two nodes are broadcasting messages at the same time, then VERSA will detect a resource conflict. We use auxiliary processes to model additional safety properties as outlined below.

5.2. Modeling Network-Code Programs

The overall approach to the translation of network programs into ACSR process is as follows. Each node in the network is represented as a collection of processes that can be grouped into three categories. The first category represents network instructions, described in detail below. The second category includes one message scheduler per network channel. Finally, the third category includes the guard processes. The latter two categories become clear as we discuss the network instruction processes. Furthermore, each message sent on the network is represented as a short-lived process whose lifetime is from the execution of the send command until the message is delivered to the receiving nodes.

An instruction $instr$ at address a is represented as a process Run_a. Processes in this category are instantaneous, that is, they do not contain time-consuming steps. The process for the $halt()$ instruction is the idle process, which is "garbage collected" by the state-space exploration engine. Processes for other instructions perform an event that correspond to the executed instruction and becomes the process for the subsequent instruction. For example, in Listing 2 Line 1, the instruction $receive(n_1, \_)$ is represented using an input event as $Run_1 = receive_{n_1,n_r}?.Run_2$. This process will block if no other process can perform the matching output event, that is to say, when the input queue of the channel n_1 in the node n_r is empty. Several instructions require special treatment, described below.

To represent conditional instructions, we introduce an auxiliary process that captures each guard's status. The guard process representing the guard g is able to 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 non-deterministic choice between receiving the two guard events. The choice is resolved by the guard process. By modeling guards in 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 $Run_a = Run_{a+1}|Delay^{dl}_{jmp}$. The new process $Delay^{dl}_{jmp}$ idles for dl time units and then behaves as Run_{jmp}. The send instruction also spawns a new concurrent process that captures the message status. It goes through three phases: scheduling of the transmission, the transmission itself, and the delivery phase. While the message is in the output queue, the process interacts with the message scheduler for the channel. By adjusting priorities of this interaction, we can experiment with various message-scheduling policies without changing the rest of the model. During the transmission, the process preforms

resource access steps, using the resource $network$. The number of these steps is determined by the message size. If another ACSR process enters the transmission phase, then the attempt to use the same resource will result in a deadlock that signals a scheduling 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 for each node in the network. The process sends the event $receive_{c,n}$, which synchronizes with the receive instruction in the network program of node n (see above). After sending the event, the process becomes idle.

5.3. 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 using a simple example.

We run two network nodes. The top part of Figure 4 shows the network-code program of both. The lower part of Figure 4 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, while a cross denotes that the process has been removed. The execution proceeds as follows: Node 0 sends two messages simultaneously, each with a 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 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 two time units and the other after four. The first instruction, executed at time 2, receives the message, but the second fails. At time 4, when the second receive happens, the second message has already expired. The process Run_6 is blocked, and VERSA detects the violation.

5.4. Safety Checks for Network Code

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

We assume that 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. We also assume that each execution of a network 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 ACSR processes representing instructions in the network code. The same check also lets us establish whether every instruction in the network program is reachable.

Behavioral checks. This group of checks ensures that the network program in each node handles the 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 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 schedulability analysis, which ensures that the message transmissions by different nodes do not conflict with each other. The transmission's deadline is defined by the parameter $relTime$ and the worst-case transmission time is provided separately from the network-code program.

As discussed in Section 5.1, schedulability violations are detected as resource conflicts in the ACSR model, which induces a deadlock. Two other important checks ensure that an attempt to receive corresponds to a prior send and that sent messages are received by some node. The first of these was also mentioned in the discussion of translation: a receive instruction will block if there are no messages to read. To detect the second one, we introduce an auxiliary process that interacts with the message delivery sub-processes. Each delivery process sends an event to the auxiliary process after a synchronization on a receive event happens. If the auxiliary process does not receive any such events, it blocks when the validity interval of the message expires. We also require that the use of channels is exclusive. That is, no node can send a message on the channel until the previous message has expired or has been received by all nodes. For this purpose, we introduce one auxiliary ACSR process per channel, which accepts send events only when there are no outstanding messages on that channel.

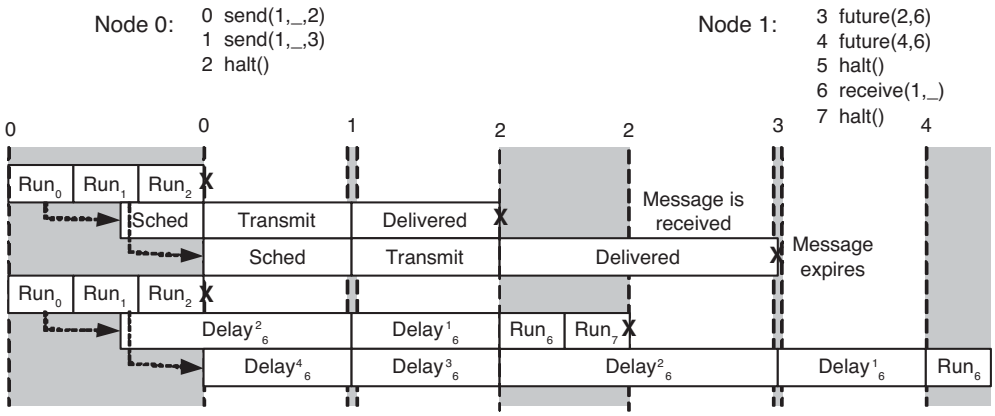

Figure 4. VERSA model execution.

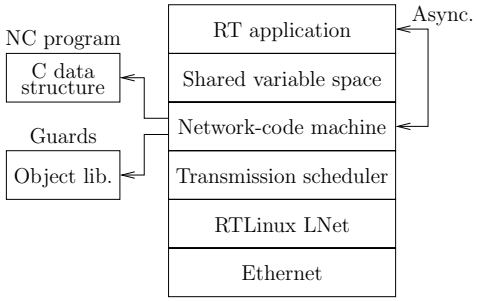

Figure 5. Network code runtime system.

6. Network-Code Runtime System

We implemented a middle-ware 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. Figure 5 shows the general structure.

The real-time application accesses values via $get()$ and $set()$ functions. They can also send messages with unbound timing constraints via the network-code machine's asynchronous messaging interface.

The shared variable space (SVS) manages data (requirement R2). Each variable stored in the SVS is a tuple consisting of: (1) the point in time the value became valid, (2) its numerical value, and (3) an optional default value. One variable stores several data entries with different values and validity times. For example, the SVS can contain the value tuples $\{v_1 = \langle 5, 1, 0\rangle, v_2 = \langle 7, 2, 0\rangle\}$ for variable x where, initially x's value is 0, at time 5 it becomes v_1, and it becomes v_2 at time 7.

The network-code machine (NCM) interprets network code and by that handles messages (requirement R3) and asynchronous communication (requirement R4). The NCM's actions are driven by the network-code program (an array of instructions representing the source). While executing the instructions, the NCM reads data values from the shared variable space and writes data updates to it. It also receives messages from a transmission scheduler and passes scheduled messages to it. To evaluate if instructions, the NCM accesses an external library containing the guard functions.

The transmission scheduler processes incoming messages and dispatches outgoing ones. The NCM can schedule multiple messages simultaneously. The transmission scheduler passes one message after the other to the network driver. The network driver notifies the transmission scheduler whenever an incoming message has arrived.

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. For example, it disables the packet framing mechanism on the network card and uses a zero-copy technique (i.e., it never copies the contents of the data packet throughout the stack implementation).

The whole system is about 3.000 lines of code. The application-specific guard functions and network-code program add extra lines.

6.1. Performance Measurements

The runtime system introduces overhead to the running system. To quantify this, we measured the instructions' execution times and compared it to non-TDMA systems and a table-driven TDMA. For sake of brevity, here we show only the results and conclusions. In [10], we provide the full data. The tests were performed on a 100Mbit network of Intel Pentium 4 with 1.5 GHz, 512 MB RAM, RTLinux-Pro 2.2, and a 3c905C-TX/TX-M [Tornado] (rev 78) with exclusive interrupt access.

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

Table 1. Code and application WCET.

Node N_1:
```
L0: destroy(0)
    create(0, 0)
    send(1, 0, I1+I2)
    future(I1+I2, L0)
    halt()
```

Listing 4. Sender.

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

Listing 5. Receiver.

We ran a test application and took about one million timing measurements per instruction. The data and histograms are available in [10]. We will now use the results to calculate the maximum throughput of our runtime system without clock synchronization. Listings 4 and 5 show a simple sender and receiver. Table 1 shows the WCET of the code. The intermediate 1 shows how long it takes until the packet is delivered at the receiver (node n_2). The intermediate 2 shows, how long it takes for the receiver to process the packet. The total cycle time is 482,990ns with a guarantee of 99% and 539,167ns with a guarantee of 99.999%. Thus, the maximum throughput of 1500b packets per second without clock synchronization is 2,069 packets or 2.96MB/s with 99% guarantee and 1,853 packets or 2.65MB/s with 99.999% guarantee, which is about 4.22 times slower than the medium's theoretic maximum itself and 3.52 times slower than our empirical average throughput on the same hardware.

The calculation above shows, that the interpretation of network code takes about 1.2 percent of the total time for the 99% guarantee scenario. A table-driven approach will further decrease this amount, but not significantly, because the 1.2 percent includes everything above the network driver, what the table-driven approach also has to do such as create, process, and free messages, and pass values to higher layers.

7. Discussion

Network code allows the developer to code tree-like communication schedules. In the following, we discuss the network code's properties.

7.1. The Message's Life Cycle

The message life cycle on the sender side consists of the explicit instructions $create$, $send$, and $destroy$. The instruction $create$ allows the developer to explicitly control, at what time the message will be created, i.e., at what time the values will be copied into the message. The instruction $send$ merely splits the message into packets and schedules it for transmission. The instruction $destroy$ frees a created message, so the message identifier can be reused.

Alternative life cycles have drawbacks. Combining the instructions $send$ and $destroy$ limits the capabilities of the network code, because then the decisions, whether or not the message will be sent has to be known at the time, at which the instruction $create$ is executed. Since this is sometimes impossible, we require an explicit instruction $destroy$. For example, at time 5ms into the period, the message needs to be created (i.e., the values copied). However, the decision, whether the message will be sent at all happens at time 500ms into the period. If the decision is to discard the message and the instruction $destroy$ does not exist, the message will stay in memory, since it cannot be sent and there is no instruction to delete it. Another approach is to use garbage collection at the hyper period. However, the network code is hyper-period agnostic, so it would require a special instruction to mark it. The chosen life cycle with the instructions $create$, $send$, and $destroy$ is a cleaner and simpler way of implementing this behavior.

7.2. The Packet's Life Cycle

The packet's life cycle is different from the message's one. The packets' life cycle is defined solely by the instruction $send$. The last parameter $relTime$ specifies the time period, for which the message is alive. After this period, the packet is removed from the packet queue.

Alternative approaches for the life cycle are: an extra instruction, garbage collection at the hyper period, overloading the instruction $destroy$, and an overwriting policy. Using an *extra instruction*, similar to $destroy$, for packets is unnecessary, because, in contrast to messages, packets are not reused. *Garbage collection* at the hyper period is impossible, since the network code is hyper-period agnostic. *Overloading* the instruction $destroy$ and using $destroy$ to

remove messages from the input queue is not possible, because it is difficult to calculate validity in advance at the receiver's side. Conditional branches at the sender's side can lead to different validity times. The *overwriting policy* is also not advisable as alternative approach, because it limits the network code's power. With this policy, sequential packets with the same sequence count overwrite each other. But, then it is impossible to distinguish, whether a packet is from this round or from the last one. This makes instructions such as $ftasync$, $ftasyncsend$, and $ftasyncreceive$ impossible.

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 $unsched$ lets anyone access the network. By that 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 Figures 3). However, to provide collision-free raw access, at most one node must be in this state simultaneously.

The following program shows, how to code raw access to TDMA slots without clock synchronization. Node 1 and node 2 split up the bandwidth evenly. Then Node 2 is allowed to use the medium for 50 time units. Afterwards, Node 1 has exclusive access. After each TDMA access, we programmed a safety delay of five time units.

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

Listing 6. Node n_1

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

Listing 7. Node n_2

7.4. Related Work

A number of network protocols and their media access control 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. Thus, most related work uses either a table-based structure to describe the schedule or implements a policy for online scheduling (see [18, 12, 11, 24, 23]).

Regarding the interpreter, the Giotto system [15] provides a similar approach. It uses an interpreter and a comparable instruction set to control timing of tasks' execution on the local node. Network code has more explicit error handling. Giotto can use the instruction $future(guard, jmp)$ with using $guard$ as error occurrence check; however, so far none of the implementations of Giotto or related projects use the guard for other than timing aspects. It is also unclear, how it actually resolves errors, since the instruction $future()$ adds another concurrent runtime trace. Existing traces are still present and may interfere with the error recovery.

In [9, 17], the authors have presented a prior version of network code. In contrast to this 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 [17]). 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 less 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 more precise control over which values he wants to use in the message.

Several other tools, such as [22, 2] can be used to formally model network code programs and verify schedules. A tempting direction of future work is to try the C interpreter of [22] or the code generation capability of [2] to generate NCM interpreters. We chose VERSA, however, because the process spawning capability of VERSA allowed us to capture, in a plain way, messages that are pending transitions, 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.

8. Conclusion

Network code is an executable abstraction for specifying a behavioral model for medium-access control algorithms for real-time communication, specifically TDMA. It is more expressive than table-driven communication schedules [9], and allows implementing tree schedules, i.e., static schedules with on-the-fly decisions during the communication cycle.

In this work, we showed that network code provides a verifiable abstraction for communication schedules. The abstraction is apt for verification, because of its limited expressiveness (e.g., no loops, no pointers). However, it is powerful enough to express arbitrary time-triggered communication schedules with on-the-fly decisions. We showed how we use VERSA to verify safety properties such as collision-free communication, schedulability, and guaranteed message reception.

We also showed that we could provide a viable imple-

mentation for the network-code abstraction. In our implementation, the runtime system adds about 1.2 percent overhead to the application with a guarantee of 99%. If this is acceptable for the real-time application, then network-code provides more flexibility and power than table-driven approaches.

For future work, we plan to extend the language to support multiple parallel media for fault tolerance reasons, verify the runtime system using CBMC [8], and investigate how we can simulate other protocol's media access control with network code.

9. Acknowledgments

We like to thank the reviewers and especially our shepherd Marco Caccamo for their time and efforts invested in scrutinizing our work and providing detailed comments and suggestions to improve the presentation of the paper. We also like to thank Gregor König, who assisted in coding the initial version of network-code machine.

References

[1] Uppaal examples. WWW. `http://www.it.uu.se/research/group/darts/uppaal/examples.shtml`.

[2] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Times: a tool for schedulability analysis and code generation of real-time systems. In P. Niebert and K. G. Larsen, editors, *Proc. of FORMATS'03*, number 2791 in Lecture Notes in Computer Science, pages 60–72. Springer–Verlag, 2004.

[3] T. Ball and S. Rajamani. The SLAM toolkit. In *Proc. of 13th Conference on Computer Aided Verification (CAV)*, number 2102 in LNCS. Springer, 2001.

[4] BERNECKER + RAINER Industrie-Elektronik Ges.m.b.H. *Ethernet Powerlink: Data Transport Services*, 5 edition, Sept. 2002. White-Paper.

[5] J. Berwanger, M. Peller, and R. Griessbach. Byteflight – a new high-performance data bus system for safety-related applications. Technical Report EE-211, BMW AG, 2000.

[6] D. Clarke, I. Lee, and H.-L. Xie. VERSA: A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2):185–215, April 1995.

[7] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. Number 0262032708. The MIT Press, 2000.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[9] S. Fischmeister. Multi-dimensional schedules for media-access control in time-triggered communication. In *Proc. of the IEEE Symposium on Computers and Communications (ISCC'05)*. IEEE Press, 2005.

[10] S. Fischmeister, O. Sokolsky, and I. Lee. Network-code machine: Programmable safe real-time communication schedules (supplementary materials). Technical Report MS-CIS-06-01, University of Pennsylvania, 2005.

[11] FlexRay Consortium. *FlexRay Communications System — Protocol Specification*, June 2004. Version 2.0.

[12] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time Triggered Communications on CAN (Time Triggered CAN - TTCAN). In *Proceedings 7th International CAN Conference*, Amsterdam, Netherlands, 2000.

[13] F. Hanssen and P. Jansen. Real-time communication protocols: an overview. Technical report, Centre for Telematics and Information Technology, 2003.

[14] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proc. of the 10th SPIN Workshop on Model Checking Software (SPIN)*, volume 2648 of *LNCS*, pages 235–239. Springer, 2003.

[15] T. A. Henzinger and C. M. Kirsch. The Embedded Machine: predictable, portable real-time code. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 315–326, 2002.

[16] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering (TSE)*, 23(5):279–295, 1997.

[17] G. König. Using Interpreters for Scheduling Network Communication in Distributed Real-Time Systems. Master's thesis, Salzburg University, Jakob-Haringer-Str. 2, 5020 Salzburg, Austria, Mar. 2005.

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

[19] J. Krakora, L. Waszniowski, P. Pisa, and Z. Hanzalek. Timed automata approach to real time distributed system verification. In *Proc. of IEEE International Workshop on Factory Communication Systems (WFCS)*, pages 407 – 410, Sept. 2004.

[20] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[21] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of the IEEE*, pages 158–171, Jan 1994.

[22] P. C. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using real-time maude. In *In Proc. Fundamental Aspects of Software Engineering (FASE'04)*, number 2984 in LNCS. Springer, 2004.

[23] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pages 134 –142. IEEE Press, June 2002.

[24] C. Venkatramani and T. Chiueh. Design, implementation, and evaluation of a software-based real-time ethernet protocol. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 27–37. ACM Press, 1995.

[25] T. Watteyne and I. Auge-Blum. Proposition of a hard real-time mac protocol for wireless sensor networks. In *Proc. of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 533 – 532, 2005.