In Proc. of the International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Heraclion, Crete, 2010.
Revised version.

Design Choices for High-Confidence Distributed Real-time Software

Sebastian Fischmeister and Akramul Azim

Department of Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca, aazim@uwaterloo.ca

Abstract. Safety-critical distributed real-time systems, such as networked medical devices, must operate according to their specification, because incorrect behaviour can have fatal consequences. A system's design and architecture influences how difficult it is to provide confidence that the system follows the specification. In this work, we summarize and discuss three design choices and the underlying concepts that aim at increasing predictability and analyzability. We investigate mandatory resource reservation to guarantee resource availability, separation of resource consumptions to better manage resource inter-dependency, and enumerative reconfiguration. We use the example of a distributed monitoring system for the human cardiovascular system to substantiate our arguments.

1 Introduction

Networked medical devices are good examples of distributed real-time systems with safety-critical functionality. They assist medical staff by automatically measuring physiologic parameters such as blood pressure, oxygen level, and heart rate, or actively influence the patient's parameters by means of infusion pumps for analgesia and insulin or breathing support. As a consequence, incorrect behaviour of the system can result in fatal outcomes for the patient. As such, patients must have confidence that the devices operate according to their specification. One way to establish confidence in the system is by making systems predictable and analyzable, which permits developers and certification authorities to inspect the system before deployment.

Many researchers have looked into the problem of how to make a system predictable and analyzable. By predictable, we mean that an external observer, for example the developer, can predict the system's behaviour with respect to input values and their timing without knowing the internal state. This allows the developer to build a system that implements a specification with strict constraints. By analyzable, we mean that the system can be subjected to formal analysis methods such as model checking which allows the developer to formally check the correct behaviour of the system.

In this work, we summarize and discuss three design choices made in previous and related works that aim at raising the predictability and analyzability of real-time systems. Our contribution is to abstract from these systems and provide a general description of the concept underlying the design choices. This allows developers to quickly understand the choices and adapt the concept for their own system.

The following paragraphs introduce the necessary concepts of resources, resource reservation, and resource consumption. We then discuss the three design decisions: mandatory resource reservation (in Section 2), separation of resource consumptions (in Section 3), and enumerative reconfiguration (in Section 4). We illustrate all three concepts with an example of a distributed patient monitoring system for the human cardiovascular system.

Applications require resources to execute. Classical resources include computation time (i.e., access to a processing unit to execute instructions), memory (i.e., temporary or permanent data storage), and communication bandwidth (i.e., access to a shared medium to transmit information to remote stations). One can extend this concept to logical resources such as locks or peripherals.

Before an application can use such resources, it must acquire them. A resource broker mechanism usually provides resources to applications. For some resources, the system implicitly allocates resources to applications. For example, when considering computation time, the dispatcher in the operating system decides at each scheduling point which process is ready to execute. For other resources, the application must explicitly request them. For example, programs usually make system calls to request memory during their execution or to statically request memory at their start time.

Systems can include mechanisms to reserve resources for applications. In such systems, the developer can specify that the system must provide a certain amount of resources to an application. For example, the developer might specify that a station can always receive 50kB/s of communication bandwidth to guarantee that the station can communicate the video stream of the surgical device or other patient data. Resource reservation schemes are well studied across the different resource types and come in great variety. For example, for computation time there are scheduling algorithms (e.g. [8, 23, 26]) and for communication bandwidth there is quality of service (e.g. [10, 22]).

For this work, we assume that the resources are reservable, meaning that we can build a resource reservation policy. For all examples involving networking, we assume that the system consists of a set of stations (e.g., patient monitors, biometric sensing devices, nurse workstation) and they are interconnected through a shared bus network.

2 Mandatory Resource Reservation

Using resources without an appropriate reservation scheme can make systems unpredictable. For example in networks without resource reservation, message transmission time can be unbounded. In Ethernet [28], developers are unable to predict the duration it will take to send an updated value from the sensor to the monitor or an alarm message from the monitor to the nurse station. One problem causing this is the Ethernet capture effect [32] that results in transient or short-term unfairness. This effect leads to incorrect behaviour, because Ethernet was designed to provide fair access to all stations, and during these periods of unfairness a single station can monopolize the channel. Thus, the developer is unable to predict how long it will take to transmit a message and thus is unable to know whether the system correctly implements a specification that requires a time bound on the transmission delay.

Bandwidth reservation as a means of resource reservation can solve this problem. Using bandwidth reservation, the developer can allocate bandwidth for each station and bound the transmission delay. Several different real-time protocols on top of Ethernet have demonstrated that this is technically feasible by extending the drivers in the stations [11, 15, 29, 38] or switches [9, 21, 36].

Resource reservation can thus increase the predictability of medical device software. Using resource reservation, developers can rely that sufficient resources will be available for the application whenever it needs these resources. Therefore, the application will never have to wait for the system to free up resources.

Resource reservation can be either *mandatory* (driven by the system) or *discretionary* (driven by the applications). Mandatory means that the system guarantees the resource reservation for applications, and applications are unable to alter these reservations. In contrast to this, discretionary resource reservation allows applications to request more or less resources at run time. For example, the partitioning scheme specified by ARINC 653 [3] and cyclic executives [26] implement mandatory resource reservation. Works such as FTT-Ethernet [29] and RETHER [38] provide discretionary resource reservation, since applications can choose to request changes for their present reservations.

Mandatory resource reservation fits well for networked medical devices. Since mandatory resource reservation prohibits applications from changing their reservations, it is easier to provide evidence on the behaviour of such systems than systems with discretionary resource reservation. Mandatory resource reservation remains static and provides a complete specification of how the broker will distribute resources at run time. The resource reservation itself can then become evidence for establishing confidence in the system's correctness. Examples of this type can be a fully specified time-triggered schedule as found in TTP [22], the dispatch table of a cyclic executive [26], or the tree schedule encoded in a Network Code program [15].

We now exemplify the concept of mandatory resources reservation by looking at our previous work on tree schedules [5, 15]. We assume a set of network stations that exchange messages with each other. Stations store messages in queues before they can transmit them. A message can either contain arbitrary contents or a specific variable v. Message ordering in queues is local to each station.

We assume that time is given in discrete units, and that time is measured on a global clock. The communication medium provides an atomic broadcast service, so either all of the stations or none of them receive a message. All messaging behaviours for which developers want to give guarantees are known a-priori.

An informal description of a tree schedule is then a tree structure with a root and a set of leaves where each vertex in the structure specifies a messages to be transmitted and each edge a possible state transition. Edges contain enabling conditions. At run time, for each vertex exactly one edge is enabled at any given time. Whenever an execution reaches a leaf of the tree, it will loop back to the root. For a formal definition, we refer the reader to related work [5, 15].

Figure 1 shows a tree schedule. Labels on vertices show which variable needs to be communicated. An assignment of ϵ means that nobody will transmit. To simplify the example, we assume that each location has a duration of one time unit and that the system is already synchronized.

The system executes the tree schedule as follows: First one station transmits v_1 followed by a message containing v_2. Then, the enabling condition g_1 determines which edge to follow. If $\neg g_1$ holds, then v_3 will be transmitted; otherwise, the system will leave the medium idle for one time unit.

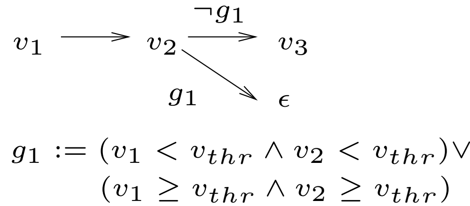


Fig. 1: Example of a tree schedule.

Tree schedules can still lead to unbounded communication delays, because the tree schedule itself may encode collisions on the medium and thus force retransmissions. Developers must choose the right type of communication to prevent this. Tree schedules can model and execute two different types of traffic: guaranteed and best effort. Also, developers can increase the level of detail by either communicating individual variables or using general message passing. The difference between these types of communication lies within the ownership of the queues, meaning which stations know the different types of queues.

For example, the communication type of *guaranteed variable updates* will occur, if only one station transmits in that state of the tree schedule, and the transmission is specifically bound to a variable. The update is guaranteed since no other station will transmit and thus the communication will be free of collisions. On the other hand, *best effort messaging* will occur, if more than one station is permitted to transmit data from their send queue in the state of the tree schedule. If more than one station has a message in its send queue, then communication problems such as collisions or packets drop might occur. These different types of communication are visible from the specification of the tree schedule, and the system also directly executes the tree schedule as it gets encoded in the Network Code language [15].

Since the system will execute the tree schedule at run time, developers can use the tree schedule itself and state-space exploration on the schedule as evidence that the system works correctly. In the later sections, we will demonstrate the advantages of hard coded enabling decisions. Here, we only argue that the schedule enables developers to, for example, provide upper bounds on the resource allocations for specific applications. For the tree schedule in Figure 1, the developer can claim that the variables v_1 and v_2 will always receive bandwidth and stations will always receive updated values every three time units.

3 Separation of Resource Consumptions

Another element reducing predictability and analyzability is the high degree of internal dependencies of resources within programs. A program requires many different

resources and uses them as the program code specifies. Consequently, consecutive lines in the program code can use different resources. This causes a dependency between the resources that is hidden in the program. While such dependencies are of no concern in traditional systems, they become a major concern for safety-critical systems, because variations in the use of resources in one line can affect subsequent lines.

```
1  thread_run() {
     float *d=NULL;
3    while ( 1 ) {
       d = malloc(sizeof(float)*10); // mem: allocating memory
5      acquireFilteredValues(d);     // cpu: computing
       msg_send(d);                  // net: communicating
7      free(d);                      // mem: deallocating memory
       milliSleep(100);              // time: controlling time
9  }}
```

Listing 1.1. Sample C program for computing a value and transmitting it.

Listing 1.1 shows a short example of a program that performs a simple task, but it is hard to predict the timing behaviour. The program first allocates memory to read some sensor values. Then, it sends them to another station through the network, and frees the memory again. The program then delays for 100 milliseconds before it repeats this behaviour. Now, the interesting question is: Does the program really send a new message every 100 milliseconds?

Unfortunately, several plausible scenarios can prevent the program from sending a message every 100 milliseconds. The scenarios range from memory allocation to preemption, to collisions on the network to the clock granularity. For example, in Lines 4 and 7, the program executes memory operations. Depending on the current state of the memory manager, allocating memory might take more or less time. For example, the memory manager might need to swap out processes to free a memory frame, it might decide to flush buffered pages, or it might change the resident set sizes for processes. In Line 8, the exact time of the delay depends on the clock granularity supported by the operating system and the actual crystal. The actual duration of the milliSleep() call varies depending on these factors. Worst of all, the individual effects influence each other, so for example, the program might send the message late (see Line 6), because of a delay in the memory allocation. Individual small modification can cause ripple effects throughout the system and manifest at parts of the program. This complicates tracing the effect back to the source.

Industry and academia know this problem and provide approaches for individual effects. For example, work on synchronous languages [17] addressed the precision problem by reducing reaction intervals to reaction instants. Another work investigated jitter of conventional sleep functions in operating systems [14]. Other work addresses the problem of predictability of execution at the hardware level [25]. Industry uses static allocation of memory and other resources to minimize the dependencies. The Ravenscar profile [13], RavenSPARK [37] and work on MISRA-C [27] provide evidence for this.

With the observation that resource dependencies cause problems, we argue for factoring out the reservation and consumption parts into separate programs. While overall, the number of inter-dependencies remains the same, encapsulating them

and joining them through a well-defined mechanism makes the program predictable and more analyzable. The approach mimics divide-and-conquer in that it splits the whole program into several pieces where each piece has a minimal set of resource dependencies—in the optimal case only dependencies for one resource—we make the programs easier to understand and analyze. After the developers specify the pieces, they can join them together for example through specified timed interactions such as timed interfaces [12] and retain predictability. For medical system software, this means that the developers write independent pieces of code with little resource interdependencies and then, for example, join them through a pipe and filter architecture with well-known temporal behaviour.

Several systems have already tried to lower the resource interdependency by encapsulating resource use in separate layers. For example, Giotto [19] separates the value and execution domain. In this system, the reading and writing of values is independent of the program execution. Changing one does not necessarily require changes in the other. TTP [22], similar to other communication protocols, separates the communication from the execution domain. A schedule defines when nodes send and receive messages which are independent when tasks running on the nodes produce new values. In PEACOD [6], the authors provide a framework for specifying resource consumptions for small pieces of code to provide compositionality and predictable behaviour for multiple resources.

In the following, we show how we separate computation and communication in the Network Code framework [15] that implements tree schedules. Figure 2 provides the overview of the architecture. Computation tasks on the top implement the application logic. The communication tasks on the bottom implement the communication behaviour. Both layers interact through buffers and queues. The typical data flow is as follows: the computation tasks produce new data and write it into the buffers. The communication tasks read and encapsulate this data in messages and transmit them on the communication medium. At the remote station, the communication tasks will receive the messages and write their contents into the buffers. Finally, the computation tasks at the remote station will process the new data.

The computation tasks can only use computation and associated resources such as memory. Computation tasks never directly access the medium. Communication tasks only use the communication medium as a shared resource, all other resources need to be provided separately.

Building such a system is feasible and robust [16] as the hardware implementation shows. It isolates the computation part from the communication part in the program. Since we use tree schedules to specify the communication tasks, we can verify the communication behaviour on the shared medium and for example performing static checks for collisions, buffer underrun, buffer overruns, sender/receiver pairing, and incorrect messaging lifecycle.

For the example shown in Listing 1.1, the developer will specify a communication task on the sending station that reads the value of `d` from the buffers and transmits it precisely every 100 milliseconds. The receiving stations will run the matching tasks that receive the transmitted value and store it in the buffers. On the computation layer, tasks will now only be concerned with memory and computation resources, which the developer can easily resolve by statically allocating the memory and then performing schedulability analysis for the computation parts. We acknowledge that

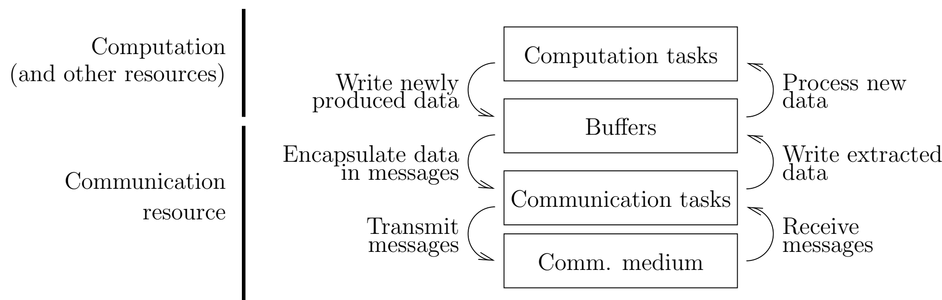


Fig. 2: Overview of the Network Code framework.

such a system still contains jitter caused by, for instance, hardware effects, however, we argue that the developers can place more confidence in the system. This increase in confidence originates from the better handling of the dependencies and using the interaction between the tasks and the buffers as well as the tree schedule as evidence.

4 Enumerative Reconfiguration

Reconfiguration in systems has been shown to allow developers to build systems that can adapt to new use cases, increase system survivability [35], and improve efficiency in the use of system resources [7]. Any mass-produced safety-critical device benefits from these properties. Medical devices also benefit, as reconfiguration enables an integrated clinical environment [34] which improves service quality and reduces cost. However, the increase in complexity by providing a reconfiguration mechanism must not compromise the system's correctness, so an important question is how to provide the reconfiguration mechanism and still establish confidence in the system's correctness.

We want systems to be reconfigurable, but without knowing whether the system works in a different configuration, the system will be unusable for safety-critical applications. Reconfigurable systems can either be space constrained (bounded state-space) or unconstrained (unbounded state-space). In general, space unconstrained reconfiguration schemes provide more flexibility but are unable to provide evidence that the system behaves as it should, which makes it hard to provide guarantees on the behaviour of the systems. For this reason, we argue that reconfiguration must be bounded by constraints.

Bounding the reconfiguration space can either use a constraint-based or an enumerative approach. In the constraint-based approach, the developer specifies constraints at a high-level within which the system can choose its point of operation. For example, the developer can specify a range of acceptable data rates for transmitting the patient's parameters to the monitor. Then at run time depending on the actual situation the system will choose a data rate within the range. The advantage of constraint-based reconfiguration is that it provides a large state space within which the system can choose the best point of operation for the current situation.

In the enumerative approach, the developer exhaustively lists all possible configurations and the system picks one configuration at run time. In the example with the data rates, the developer will, for instance, specify three possible data rates and the system will select one of the three rates. Systems with multiple modes of operation usually provide exhaustive lists in which a mode usually realizes a system's functionality for a particular configuration.

One advantage of using the enumerative approach instead of the constrained-based approach is the guarantee that the system supports reconfiguration but remains analyzable. If system's state-space is small, then verifying the system will be tractable. However, the size depends on the constraints and the application, and a system with loose constraints can easily run into the state-space explosion problem. In the enumerative approach, the state space is automatically constrained by the requirement to list all possible reconfigurations.

Note that fast checks for safe reconfiguration are available [33, 4], however, they only apply to the resource reservation parts and developers still need to establish confidence in the functional correctness for all possible configurations once they have sufficient resources.

Several systems support reconfiguration. For example TTP/C [22] supports up to 30 modes with a safe mode-change protocol. Tree schedules [15] can encode modes in the structure that can have safe transitions. Endochronous clocked graphs [31] can encode different modes similarly to tree schedules.

Section 2 demonstrates how we can encode different configurations in tree schedules. The tree schedule shown in Figure 1 includes two modes of operation: one where values v_1 and v_2 agree and the other where the two values disagree. Similar to this configuration, we can encode the list of configurations in the tree structure and verify properties as already mentioned in the previous sections.

5 Illustrative Example

We use the following example to summarize and substantiate our point about the three concepts mentioned in the previous sections. The aim is to integrate all three concepts into one example and show how one can provide evidence using formal verification. We go through the following steps of building the system: (1) we build the resource reservation scheme using tree schedules, (2) we provide evidence that the resource reservation works by means of formal verification enabled by the separation of resource consumptions and the enumerative reconfiguration, and (3) we show the simulation framework for tree schedules in Matlab Simulink to test tree schedules before deployment.

5.1 Overview

We assume a distributed patient monitoring system in which body sensors transmit physiological parameters to the patient monitor. When the pulmonary vascular resistance (PVR) of the patient passes a given threshold, the patient monitor will send an alarm message to the nurse station within bounded time.

PVR [2] is the resistance in the pulmonary vascular bed against which the right ventricle must eject blood. To calculate the pulmonary vascular resistance, the patient monitor requires the left atrial pressure (LAP) or the pulmonary capillary wedge pressure (PCWP), the pulmonary artery pressure (PAP), and the cardiac output (CO). PCWP provides an indirect estimate of LAP. PCWP is measured by wedging a catheter into a small pulmonary artery tightly enough to block flow from behind. LAP can be measured by placing a special catheter into the right atrium and then pushing through the inter-atrial septum. Since the patient monitor only requires the LAP or the PCWP, we can create several modes for the operation of the monitor:

- Configuration 1: The patient monitor uses the PAP, CO, and LAP.
- Configuration 2: The patient monitor uses the PAP, CO, and PCWP.
- Configuration 3: The patient monitor uses the PAP, CO, and LAP. If an alarm is pending, then the monitor will make a safety check and also acquire the PCWP, before signaling the nurse alarm. This will lower the number of false alarms as it eliminates the problem of incorrect LAP measurements.
- Configuration 4: This is similar to configuration 3 but the patient first uses PAP, CO, and PCWP, and then uses LAP for the fail safe.

We treat calculating the pulmonary vascular resistance as a single transaction. This means that the system should always complete all data transmissions that the patient monitor requires before reconfiguring (e.g., changing configuration). This assumption is important, because we model setting the configuration with a physical button which the nurse can press with a frequency of at most once in a fixed amount of time. In addition, the patient monitor must signal the nurse alarm within a bounded time when the pulmonary vascular resistance exceeds a specific threshold.

5.2 Developing the Tree Schedule

Based on the specification in Section 5.1, we can develop the tree schedule for the resource reservation. We assume that communicating one value takes one time unit, and the inter-arrival time of button pressed events is set accordingly. Figure 3 shows the tree schedule that implements the specification (or so we claim). A vertex labeled ϵ takes zero time and we use it to encode branches with more than two choices or for early termination of the schedule. The PVR monitoring station can operate in four configurations. In any of the four configurations, the monitoring system at first receives the value of the PAP from the circulatory system to calculate the PVR. If the received value is out of the normal range for PAP values (i.e. 10-20 $mmHg$), the system will enter the *safety interlock* state. In the *safety interlock* state, the system checks the important functions of the human cardiovascular system such as the patient's pulse rate while resting (60-100 beats per minute) to determine the patient's safety. This assumption is implicit and not shown in the Figure 3. After receiving the value of CO within the normal range (4 L/min-8 L/min), the system can either receive LAP (normal range 6-12 $mmHg$) or PCWP (normal range 6-12 $mmHg$) based on the current configuration. The patient monitor will receive PCWP after PAP, if the system uses configuration 3. On the other hand, the monitor will receive LAP after the PAP, if the system runs in the default configuration (i.e.,

any configuration other than 1, 2, and 3). The system will enter into the safety interlock state for out of the normal range of CO, LAP, or PCWP. The system will generate an alarm and notify the nurse, when the PVR exceeds normal value ($> 250\ dyn.s/cm^5$). The nurse can change the configuration of the monitoring system at any point in time, but not in the middle of a transaction.

Guards g_1 and g_2 define the enabling conditions whether the PVR value of the patient exceeds the defined threshold thr. Guards g_3 to g_6 are enabling conditions depending on the configuration setting. We assume configuration 4 to be the default configuration.

For demonstration purposes, we walk through one configuration for which we assume $conf = 4$ and PVR $\geq thr$. In the root location labeled ϵ_0, only g_6 will be enabled. The tree schedule specifies that the next three messages on the bus will be PAP, CO, and PCWP. At that point PVR exceeds the threshold thr (PVR $\geq thr$), so g_1 is true and the patient monitor will also receive the LAP measurement. Finally, g_1 will again be true and the patient monitor will signal the nurse alarm before the tree schedule restarts at its root location.

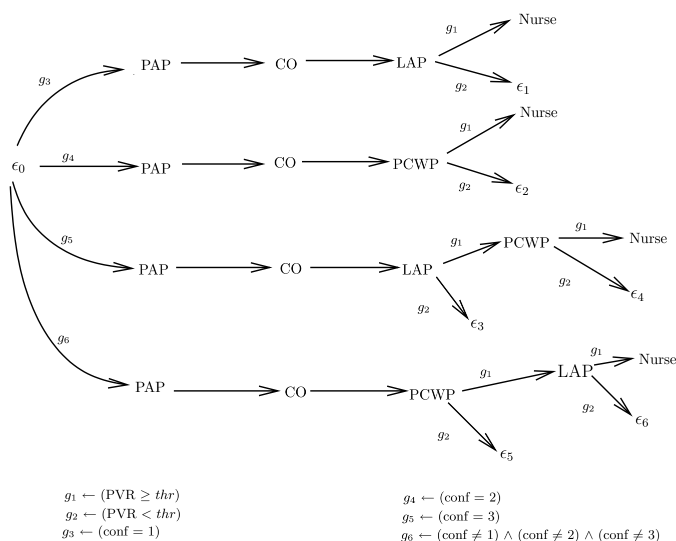


Fig. 3: The tree scheduling for the patient monitoring system.

5.3 Verifying the Tree Schedule

To provide evidence that our system meets the specification with respect to the communication requirements, we provide the following guarantees for our reservation mechanism. Note that we can verify these properties, because we separate communication from computation in our framework (see Section 3) and we can enumerate all configurations (see Section 4).

– **P1:** In every t time units and in all configurations, the PVR monitoring system will receive all data necessary to compute and display the new PVR value.
– **P2:** When PVR $\geq thr$, then the nurse will be notified no later than x time units in all configurations.
– **P3:** Calculating PVR is atomic and when the system is in a particular configuration mode, it is not possible to switch to different modes.
– **P4:** The system will always make progress and never gets stuck.

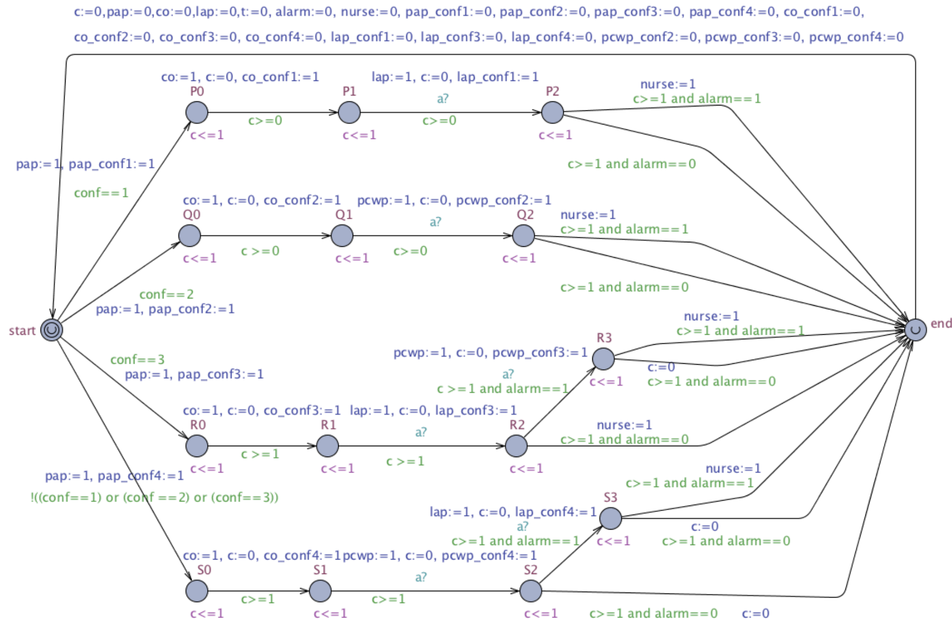


Fig. 4: Modeling PVR monitoring system in UPPAAL

To provide evidence that these properties hold in our system, we encode the tree schedule in a timed automaton and check the properties using UPPAAL. UPPAAL [1] is a timed-automata based model checker that allows formal verification of temporal logic properties in finite systems. Figure 4 shows the tree schedule part of the UPPAAL model. The whole system comprises three different processes: one modeling the tree schedule, one modeling the nurse, and one modeling the alarm condition. All three processes run in parallel. The nurse process can alter the `conf` variable at most once every time unit. The nurse alarm will sound, if the variable `nurse` is set to one. We use channel `a` to synchronize the alarm process with the tree schedule process. The alarm process implements the non-deterministic choice whether an alarm happened or not. In Figure 4, the clock `c` constraints state changes (one transmission requires one time unit), the clock `t` counts for the cycle, and the clock `tall` always increases. We use `t` and `tall` for verification purposes only. We can now check the properties defined above using UPPAAL's query language:

- **P1:** $A\Box(P.end \rightarrow (pap = 1 \wedge co = 1 \wedge lap = 1) \vee (pap = 1 \wedge co = 1 \wedge pcwp = 1)) \wedge (P.end \rightarrow (t \leq 5))$: Whenever the system reaches the end, PAP, CO, and LAP or PCWP have been transmitted. And, the system always reaches the end withing *five* time steps.
- **P2:** $A\Box((adly \geq 3) \wedge (adly \leq 4) \rightarrow ((alarm = 0) \vee (nurse = 1)))$: The system will notify the nurse within three to four time units after an alarm happened.
- **P3:** $A\Box((pap_conf1 = 1) \vee (co_conf1 = 1) \vee (lap_conf1 = 1)) \rightarrow \neg((pap_conf2 = 1) \vee (co_conf2 = 1) \vee (pcwp_conf2 = 1) \vee (pap_conf3 = 1) \vee (co_conf3 = 1) \vee (lap_conf3 = 1) \vee (pcwp_conf3 = 1) \vee (pap_conf4 = 1) \vee (co_conf4 = 1) \vee (lap_conf4 = 1) \vee (pcwp_conf4 = 1))$: When the system is in configuration 1, the system cannot enter into configuration 2, 3, or 4. Therefore, The system cannot be switched to different configurations while it is in a configuration mode.
- **P4:** $A\Box\neg deadlock$: The system will always make progress and not deadlock.

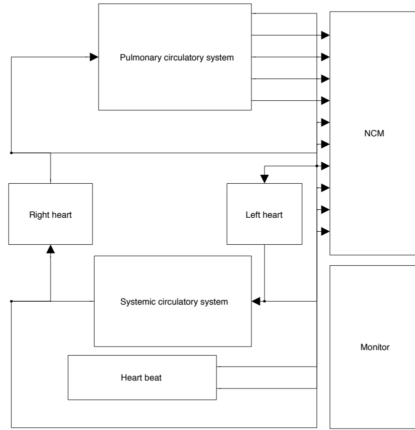
Note, compare the complexity involved in checking such properties for general programs that mix computation and communication in a programming language like C. We can easily now connect the communication layer with the computation layer through buffers and a specification when values get read and written in these buffers (as we have done in [15]). Also note that we generate the schedule from high-level specifications [30].

5.4 Simulating the System

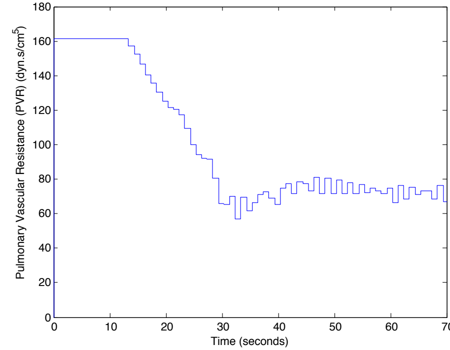
We use Simulink to simulate our patient monitoring system. We implement the patient monitor and connect it to a model of a human cardiovascular system [20] using TrueTime [18]. TrueTime supports simulating network communication for real-time control systems. The human cardiovascular system implements a heart model and produces different physiological parameters of the heart.

In our simulation, we implement the tree schedule defined in Section 5.2. Figure 5(a) provides an overview of the Simulink model and Figure 5(b) shows the resulting PVR value of a sample run of the simulation. The human cardiovascular system abstracts the implanted body sensors that report PAP, LAP, CO, and PCWP to the external PVR monitoring system. The tree schedule runs inside the network code machine (NCM) implemented on top of TrueTime. We can implement the tree schedule inside a state machine using the StateChart block.

The human cardiovascular system components connect with the TrueTime network, and the monitoring system receives the physiological parameters through the network. The basic elements of TrueTime are the TrueTime send block, the TrueTime network block, and the Network Code Machine block that implement the tree schedule. The monitoring system use TrueTime receive blocks to receive data over the network. If we enter the subsystems of the system model, we will see the detailed interactions of Matlab, Simulink, and TrueTime elements for each subsystem. Before starting the simulation, we set the parameters of different elements of the system model such as network type, number of nodes, data rate and frame size in the TrueTime network block.



(a) Human cardiovascular system model in Simulink



(b) Monitoring the PVR

Fig. 5: Simulation of tree schedules for the human cardiovascular system example.

6 Conclusion

In this work, we discussed three useful concepts following design decisions that aim at increasing the predictability and analyzability of real-time systems: resource reservation, different types of resource consumptions, and constrained reconfiguration. Our work on tree schedules created the guiding example for each of these three mechanisms. Finally, we showed an illustrative example of a distributed patient monitoring system in which we went through the phases of specifying, checking, and finally simulating the system.

The mentioned concepts open up many avenues for future work. One can explore each of the mentioned concepts in more detail for different resource types and investigate how to join them together into one framework for multiple resources. Another interesting problem is compositionality of resource consumptions and evidence. This points to the question of how can one place confidence in the whole system from having evidence that all the individual modules work as specified.

Acknowledements

We would like to thank the members of the Embedded Software group at the University of Waterloo for their feedback on an early draft as well as Hiren Patel for in-depth discussions and Steven Dain for information on the medical parts of the work. This research was supported in part by NSERC DG 357121-2008, ORF RE03-045, and ISOP IS09-06-037.

References

1. UPPAAL—An Integrated Tool Environment for Modeling, Validation, and Verification of Real-Time Systems. www.uppaal.com. Visited June. 2010.
2. A. E. Abbas, F. David Fortuin, N. B. Schiller, C. P. Appleton, C. A. Moreno, and S. J. Lester. A Simple Method for Noninvasive Estimation of Pulmonary Vascular Resistance. *Journal of the American College of Cardiology*, 41(6):1021–1027, 2003.
3. Incorporated (ARINC) Aeronautical Radio. ARINC 653 (Avionics Application Standard Software Interface). ARINC Standard, 2003.
4. L. Almeida, M. Anand, S. Fischmeister, and I. Lee. A Dynamic Scheduling Approach to Designing Flexible Safety-Critical Systems. In *Proceedings of the 7th Annual ACM Conference on Embedded Software (EMSOFT)*, pages 67–75, Salzburg, Austria, October 2007.
5. M. Anand, S. Fischmeister, and I. Lee. Composition Techniques for Tree Communication Schedules. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 235–246, Pisa, Italy, July 2007.
6. M. Anand, S. Fischmeister, and I. Lee. Resource Scopes: Toward Language Support for Compositional Determinism. In *Proceedings the 12th IEEE International Symposium on Object/component/service-oriented Real-time Distributed Computing (ISORC)*, pages 295–304, Tokyo, Japan, March 2009.
7. G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic Scheduling for Flexible Workload Management. *IEEE Transactions on Computers*, 51(3):289–302, 2002.
8. G.C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, 2000.
9. G. Carvajal and S. Fischmeister. A TDMA Ethernet Switch for Dynamic Real-Time Communication. In *Proc. of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Charlotte, United States, May 2010.
10. G. Coulouris, J. Dollimore, and T. Kingberg. *Distributed Systems: Concepts and Design*. Queen Mary and Westfield College, University of London, 1996.
11. R. Court. Real-time Ethernet. *Comput. Commun.*, 15(3):198–201, 1992.
12. L. de Alfaro, T.A. Henzinger, and M. Stoelinga. Timed Interfaces. In *Proceedings of the Second International Workshop on Embedded Software (EMSOFT)*, number 2491 in LNCS, pages 108–122. Springer, 2002.
13. B. Dobbing and A. Burns. The Ravenscar Tasking Profile for High Integrity Real-time Programs. In *Proceedings of the 1998 annual ACM SIGAda international conference on Ada (SIGAda)*, pages 1–6, New York, NY, USA, 1998. ACM.
14. A. Dubey, G. Karsai, and S. Abdelwahed. Compensating for Timing Jitter in Computing Systems with General-Purpose Operating Systems. In *Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, Tokyo, Japan, March 2009.
15. S. Fischmeister, O. Sokolsky, and I. Lee. A Verifiable Language for Programming Communication Schedules. *IEEE Transactions on Computers*, 56(11):1505–1519, November 2007.
16. S. Fischmeister, R. Trausmuth, and I. Lee. Hardware Acceleration for Conditional State-Based Communication Scheduling on Real-Time Ethernet. *IEEE Transactions on Industrial Informatics*, 5:3, 2009.
17. N. Halbwachs. *Synchronous Programming of Reactive Systems*. Kluwer, 1997.
18. D. Henriksson, A. Cervin, and K. Erik rzn. TrueTime: Real-time Control System Simulation with MATLAB/Simulink. In *In Proceedings of the Nordic MATLAB Conference*, 2003.
19. T. A. Henzinger, C. M. Kirsch, and B. Horowitz. Giotto: A Time-triggered Language for Embedded Programming. In T. A. Henzinger and C. M. Kirsch, editors, *Proceedings*

of the 1st International Workshop on Embedded Software (EMSOFT), number 2211 in LNCS. Springer, October 2001.
20. Z. Hu and Y. Diao. Primary Model of Heart-systemic-pulmonary System. *Journal of Tongji University*, 30(1):61–65, 2002.
21. J. Jasperneite, P. Neumann, M. Theis, and K. Watson. Deterministic Real-Time Communication with Switched Ethernet. In *Proceedings of 4th IEEE International Workshop on Factory Communication Systems (WFCS)*, 2002.
22. H. Kopetz. *Real-time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.
23. J. Leung, editor. *Handbook on Scheduling*. CRC Press, 2004.
24. N. G. Leveson and C. S. Turner. An Investigation of the Therac-25 Accidents. *Computer*, 26(7):18–41, 1993.
25. B. Lickly, I. Liu, S. Kim, H. Patel, S. Edwards, and E. Lee. Predictable Programming on a Precision Timed Architecture. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pages 137–146, New York, NY, USA, 2008. ACM.
26. J. Liu. *Real-Time Systems*. Prentice-Hall, New Jersey, 2000.
27. G. McCall. *Misra-C: 2004*. MIRA Limited, Warwickshire, United Kingdom, 2004.
28. R. M. Metcalfe and D. R. Boggs. Ethernet: distributed packet switching for local computer networks. *Commun. ACM*, 19(7):395–404, 1976.
29. P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 134 –142. IEEE Press, June 2002.
30. D. Potop-Butucaru, A. Azim, and S. Fischmeister. Semantics-preserving Implementation of Synchronous Specifications over Dynamic TDMA Distributed Architectures. In *Proceedings of the 10th International Conference on Embedded Software (EMSOFT)*, 2010.
31. D. Potop-Butucaru, R. de Simone, Y. Sorel, and J. Talpin. Clock-driven Distributed Real-time Implementation of Endochronous Synchronous Programs. In *Proceedings of the 7th ACM International Conference on Embedded Software (EMSOFT)*, pages 147–156, New York, NY, USA, 2009. ACM.
32. K.K. Ramakrishnan and H. Yang. The Ethernet Capture Effect: Analysis and Solution. In *Proc. 19th Local Computer Networks Conference*, 1994.
33. J. Real and A. Crespo. Mode Change Protocols for Real-Time Systems: A Survey and a New Proposal. *Real-Time Systems*, 26(2):161–197, 2004.
34. R. Schrenker. Software engineering for future healthcare and clinical systems. *Computer*, 39(4):26–32, 2006.
35. C. Shelton and P. Koopman. Improving System Dependability with Functional Alternatives. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*, page 295. IEEE Computer Society, 2004.
36. K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz. A Time-Triggered Ethernet (TTE) Switch. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 794–799, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.
37. Praxis Critical Systems. SPARK 95 - The SPADE Ada 95 Kernel (including RavenSPARK). RavenSPARK S.P0468.73.62 version 4.8, January 2008.
38. C. Venkatramani and T. Chiueh. Supporting real-time traffic on Ethernet. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 282–286. IEEE Press, December 1994.