Logical Reliability of Interacting Real-Time Tasks

Krishnendu Chatterjee
UC Berkeley
c_krish@eecs.berkeley.edu

Arkadeb Ghosal
UC Berkeley
arkadeb@eecs.berkeley.edu

Thomas A. Henzinger
EPFL
tah@epfl.ch

Daniel Iercan
"Politehnica" U. of Timisoara
daniel.iercan@aut.upt.ro

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Claudio Pinello
Cadence Research Labs
pinello@cadence.com

Alberto Sangiovanni-Vincentelli
UC Berkeley
alberto@eecs.berkeley.edu

Abstract

We propose the notion of logical reliability for real-time program tasks that interact through periodically updated program variables. We describe a reliability analysis that checks if the given short-term (e.g., single-period) reliability of a program variable update in an implementation is sufficient to meet the logical reliability requirement (of the program variable) in the long run. We then present a notion of design by refinement where a task can be refined by another task that writes to program variables with less logical reliability. The resulting analysis can be combined with an incremental schedulability analysis for interacting real-time tasks proposed earlier for the Hierarchical Timing Language (HTL), a coordination language for distributed real-time systems. We implemented a logical-reliability-enhanced prototype of the compiler and runtime infrastructure for HTL.

1 Introduction

In safety-driven embedded applications, such as automotive stability controllers and medical devices, reliability and fault tolerance are increasingly important, as regulatory bodies and customers demand robust products. Much research has been carried out on topics such as reliability analysis, fault-tolerant architectures, and fault analysis. However, we are still at the early stages for design methodologies and tools that take into consideration constraints on reliability and fault tolerance in addition to traditional design constraints such as response time and power consumption. *Platform-based design* [15] emphasizes the separation of requirements (what the system is supposed to do) from architecture (the implementation resources). A mapping from the requirements specification to the architecture can then be checked for correctness. This approach has been used in [7] for real-time software tasks, where a specification defines the functional and timing requirements of tasks. In this paper, we extend the approach to reliability requirements, thus setting the foundation for a joint schedulability/reliability analysis.

We consider a set of atomic, periodic, interacting real-time tasks. The release times and deadlines of each task are specified through the read times and write times of global variables called *communicators* [6]. An architecture consists of a set of networked hosts. An implementation must assign each task to a host so that each task invocation can be scheduled between its release time and deadline. To check schedulability, the WCET of each task on each host must be known. In this way, we separate the timing specification (release times and deadlines) of the tasks from the implementation (mapping and schedule).

We treat system reliability in a similar way, by separating reliability requirements for communicators from the reliability characteristics of hosts. With each communicator, we associate a *logical* (or *long-term*) *reliability constraint* (*LRC*), which is a real number between 0 and 1. If the LRC is 0.9, this means that in the long run, at least 0.9 fraction of all periodic writes to this communicator are required to be valid values. The LRC is a requirement on the implementation, just as release times and deadlines are requirements. The mapping of tasks to hosts must ensure the LRCs of all communicators. For this purpose, if hosts fail, it may be necessary that a task be replicated on several hosts. To check if an implementation satisfies all LRCs, the *singular* (or *short-term*) *reliability guarantee* (*SRG*) of updating a communicator with a valid value must be known. The SRG is again a real number between 0 and 1; for example, an SRG of 0.8 means that the probability that a host fails during the execution of a task invocation is 0.2. The SRG is a property of the architecture, just as WCETs are architectural properties. To achieve LRCs of 0.9 with hosts that guarantee only SRGs of 0.8, all tasks that write to communicators (with LRC 0.9) need to be replicated on two hosts. This suffices when all communication between hosts is non-faulty, because $0.2^2 < 0.1$. The above assumes that task inputs are reliable; we will later discuss scenarios for non-reliable inputs. SRGs can be computed based on networks of nodes [14, 4], fault trees [12], and reliability block diagrams (RBDs). Our approach is closest to that of RBDs [12], where systems are modeled as networks with AND/OR junctions: an OR junction works reliably when any of its inputs is reliable, and an AND junction requires that all inputs be reliable.

The main contribution of this paper is the separation of reliability requirements (LRCs) specified by a program written in a task coordination language (e.g., HTL [6]), from the reliability guarantees (SRGs) offered by hosts and communication links. The program implementation (replication mapping) must ensure that all timing requirements and all reliability requirements are satisfied. LRCs, like release times and deadlines, represent application-specific ("logical") information; SRGs, like WCETs, represent architecture-specific ("physical") data. An analysis checks whether the replication mapping and task schedule satisfies the logical requirements for timing and reliability.

We assume that the architecture consists of *fail-silent* hosts and sensors connected over a broadcast network. If a fail-silent host or sensor fails, it stops functioning (becomes silent) [3]. In [2], it is argued that fail-silence can be achieved at a reasonable cost. To keep the analysis simple, we also assume a reliable broadcast network. However, less-than-perfect reliable broadcast can be handled readily as long as the broadcast is atomic, i.e., either all hosts receive identical values or none at all.

We extend the *Hierarchical Timing Language* (*HTL*) [6], a coordination language for distributed real-time systems, to capture the timing and reliability requirements of a set of software tasks. The HTL compiler performs a joint schedulability/reliability analysis for a given replication mapping of tasks to hosts and generates distributed code that satisfies the requirements. We also present a notion of refinement that preserves the reliability analysis and, thus, facilitates incremental reliability analysis in the design flow. In this way, the complexity of a joint schedulability/reliability analysis can be reduced significantly by progressing from the requirements to the final implementation in a sequence of steps.

To evaluate our approach, we set up an experiment with a three-tank system (3TS) controlled by an HTL program that targets a distributed, redundant implementation (consisting of two physically separated controllers) with the reliability requirement that the system functions even if one of the hosts fails. We verified that unplugging one of the two hosts from the Ethernet network has indeed no effect on the control performance.

Related work. In [1], a distributed static schedule is generated for a given periodic algorithm on a distributed architecture, trying to optimize reliability and the length of a period. Our paper, which is similar in approach, targets the joint satisfaction of timing and reliability requirements.

Reliability requirements can also be specified by assigning priorities to faults and tasks. Each failure pattern (a combination of faulty processors and channels) and tasks are assigned a priority, and a synthesis procedure determines the replication of tasks to ensure that, if a fault occurs, then all tasks with priority higher than the fault execute. In [13], this approach was combined with a mono-periodic data-flow model of computation. Our approach differs because LRCs are used instead of priorities.

Cyclic static schedules have been generated for platforms with transient faults and time-triggered communication. In [9], re-execution (time redundancy) and replication (space redundancy) are optimized automatically to improve schedulability. In [10], the approach is refined to include check-pointing, thus re-executing only the parts of a process that are affected by transient faults. A method to explore the trade-off between schedulability and transparency, using only re-execution, is proposed in [11].

2 Background

A *communicator* [6] is a typed variable that can be accessed (read from or written to) with a specified periodicity. Communicators are used to exchange data between tasks and their environment based on the logical execution time (LET) model of execution [7]. *Input communicators* are updated by physical sensors (possibly through drivers) and read by tasks. *Output communicators* are updated by tasks and read by physical actuators (possibly through drivers). All other communicators are only read and written by tasks. A task reads from certain instances of a set of communicators, computes a function and writes to certain instances of other communicators. The latest read and earliest write times implicitly specify the LET of a task. Fig. 1 shows four communicators $\mathtt{c}_1$, $\mathtt{c}_2$, $\mathtt{c}_3$, and $\mathtt{c}_4$ with periods of 2, 3, 4, and 2 time units, respectively. Task $\mathtt{t}$ reads the second instances of $\mathtt{c}_1$ and $\mathtt{c}_2$ and updates the third and sixth instances of $\mathtt{c}_3$ and $\mathtt{c}_4$, respectively. The LET of task $\mathtt{t}$ is five time units from time instant 3 to 8. Communicators avoid race conditions and therefore provide deterministic interaction behavior.

A communicator may have an unreliable value if a task fails to execute and/or the memory fails to store the value of the communicator. The task implementations are assumed

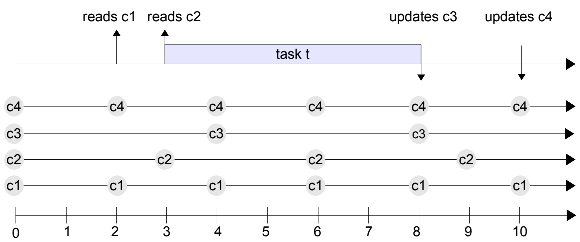

Figure 1. Communicators and tasks

to be "correct", i.e., if a task executes, then it produces the "correct" output. In this work, we present a framework to specify the logical (or long-run) reliability constraint (LRC) for a communicator, where the LRC specifies the fraction of reliable values that the communicator expects in the long run. Next, we formally define the notion of a *system* $(\mathtt{S}, \mathtt{A}, \mathtt{I})$, which consists of a specification $\mathtt{S}$, an architecture $\mathtt{A}$, and an implementation $\mathtt{I}$.

A *specification* $\mathtt{S} = (\mathtt{tset}, \mathtt{cset})$ consists of a set of tasks $\mathtt{tset}$ and a set of communicators $\mathtt{cset}$, where tasks and communicators are declared as follows.

A *communicator declaration* $(\mathtt{c}, \mathtt{type_c}, \mathtt{init_c}, \pi_\mathtt{c}, \mu_\mathtt{c})$ consists of a communicator name $\mathtt{c}$, data type $\mathtt{type_c}$, an initial value $\mathtt{init_c}$, an accessibility period $\pi_\mathtt{c} \in \mathbb{N}_{>0}$ and an LRC $\mu_\mathtt{c} \in \mathbb{R}_{(0,1]}$. All communicator names are unique. The data type includes a special symbol $\perp$ to represent unreliable communicator values; a non-$\perp$ value indicates that the communicator has a reliable value.

A *task declaration* $(\mathtt{t}, \mathtt{ins_t}, \mathtt{outs_t}, \mathtt{fn_t}, \mathtt{model_t}, \mathtt{def_t})$ consists of a task name $\mathtt{t}$, lists of inputs $\mathtt{ins_t}$ and outputs $\mathtt{outs_t}$, a function $\mathtt{fn_t}$, an input failure model $\mathtt{model_t} \in \{1, 2, 3\}$, and a list of default values $\mathtt{def_t}$. All task names are unique. An element of the input or output list is a pair $(\mathtt{c}, \mathtt{i})$ consisting of a communicator name $\mathtt{c} \in \mathtt{cset}$ and a communicator instance number $\mathtt{i} \in \mathbb{N}_{\geq 0}$. The length of the input (resp. output) list is denoted by $|\mathtt{ins_t}|$ (resp. $|\mathtt{outs_t}|$). If $\mathtt{ins_t}(j) = (\mathtt{c}, \cdot)$, then $\mathtt{type}(\mathtt{ins_t}(j)) = \mathtt{type_c}$; similarly, if $\mathtt{outs_t}(k) = (\mathtt{c}', \cdot)$, then $\mathtt{type}(\mathtt{outs_t}(k)) = \mathtt{type_{c'}}$. The function $\mathtt{fn_t}$ computes the outputs from the inputs, i.e., $\mathtt{fn_t} : \Pi_{1 \leq j \leq |\mathtt{ins_t}|} \mathtt{type}(\mathtt{ins_t}(j)) \rightarrow \Pi_{1 \leq k \leq |\mathtt{outs_t}|} \mathtt{type}(\mathtt{outs_t}(k))$. The set of communicators read by task $\mathtt{t}$ is denoted by $\mathtt{icset_t}$.

For a task $\mathtt{t}$, its *read time* $\mathtt{read_t}$ is the latest communicator instance read by $\mathtt{t}$ and its *write time* $\mathtt{write_t}$ is the earliest communicator instance written by $\mathtt{t}$. Formally, $\mathtt{read_t} = \max_\mathtt{j}(\pi_\mathtt{c} \cdot \mathtt{i})$, where $\mathtt{ins_t}(j) = (\mathtt{c}, \mathtt{i})$ and $\mathtt{write_t} = \min_k(\pi_{\mathtt{c}'} \cdot \mathtt{i})$, where $\mathtt{outs_t}(k) = (\mathtt{c}', \mathtt{i})$. The tasks repeat with periodicity $\pi_\mathtt{S}$, where $\pi_\mathtt{S}$ is derived from the write time of the tasks. Formally, $\pi_\mathtt{S} = \mathrm{lcm}(\mathtt{cset}) \cdot \lceil (\max_{\mathtt{t} \in \mathtt{tset}} \mathtt{write_t}) / (\mathrm{lcm}(\mathtt{cset}) \rceil$, where $\mathrm{lcm}(\mathtt{cset})$ is the least common multiple of the communicator periods.

The restrictions on task declarations are as follows: (1) all tasks read from some communicators and write to some communicators, (2) for all tasks, the read time is strictly earlier than the write time, (3) no two tasks write to the same communicator, and, (4) no task can write a communicator instance multiple times. In other words, a communicator can be written by at most one task at any time instant, i.e., the specification is *race-free*.

The input failure model $\mathtt{model_t}$ denotes the action of a task if one or more inputs are unreliable. The list $\mathtt{def_t}$ is a list of default values for communicators being read by the task (if the communicator is not reliable at read time). Three failure models are considered: series ($\mathtt{model_t}$ = 1), parallel ($\mathtt{model_t}$ = 2) and independent ($\mathtt{model_t}$ = 3). For failure model *series*, if any one of the inputs fails, the task fails to execute. For failure model *parallel*, if an input is unreliable, the task may execute by using the default value of the communicator from the list $\mathtt{def_t}$. If all of the inputs are unreliable the task fails to execute. For failure model *independent*, if an input is unreliable, the task uses the corresponding default value for that input from the list $\mathtt{def_t}$. The task may execute even if all inputs are unreliable.

An *architecture* $\mathtt{A}$ is a tuple $(\mathtt{hset}, \mathtt{sset}, \mathtt{C_S})$ where $\mathtt{hset}$ is a set of hosts (connected over a reliable broadcast network), $\mathtt{sset}$ is a set of sensors and $\mathtt{C_S}$ is a set of *architectural constraints* for a given specification $\mathtt{S} = (\mathtt{tset}, \cdot)$. The constraints are: (1) reliability of hosts and sensors specified by *host reliability map* $\mathtt{hrel} : \mathtt{hset} \rightarrow \mathbb{R}_{(0,1]}$, and *sensor reliability map* $\mathtt{srel} : \mathtt{sset} \rightarrow \mathbb{R}_{(0,1]}$; and, (2) execution metrics for the tasks specified by *worst-case-execution-time (WCET) map*, $\mathtt{wemap} : \mathtt{tset} \times \mathtt{hset} \rightarrow \mathbb{N}_{>0}$ and *worst-case-transmission-time (WCTT) map*, $\mathtt{wtmap} : \mathtt{tset} \times \mathtt{hset} \rightarrow \mathbb{N}_{>0}$. The hosts are assumed to be *fail-silent* [3], i.e., if a host fails it does not produce any garbage output. Non-reliability in broadcast networks can be accounted for in our model as long as the faulty behavior is *atomic*, i.e., if the broadcast fails none of the hosts receives any input. The WCTT is measured as the broadcast time for each task from each host. Memory is assumed to be 100% reliable.

Given a specification $\mathtt{S} = (\mathtt{tset}, \cdot)$ and an architecture $(\mathtt{hset}, \cdot, \cdot)$, an *implementation* $\mathtt{I}$ is a function from tasks to a set of hosts, i.e., $\mathtt{I} : \mathtt{tset} \rightarrow 2^{\mathtt{hset}} \setminus \emptyset$. If a task $\mathtt{t}$ is mapped to multiple hosts, then each host $\mathtt{h}$ executes a local copy of $\mathtt{t}$; the local copy is referred to as a *task replication* $(\mathtt{t}, \mathtt{h})$. All communicators $\mathtt{c}$ are replicated on all hosts $\mathtt{h}$; each local copy of a communicator is referred to as a *communicator replication* $(\mathtt{c}, \mathtt{h})$. When a task replication completes execution, it broadcasts the output to all hosts (to update relevant communicator replications). For schedulability analysis, the end-to-end task execution times therefore include both WCETs as well as WCTTs.

Semantics. An execution of an implementation, also called an *implementation trace* (or simply *trace*), is a (possibly infinite) sequence of communicator values for every time instant. A *time instant* is a sequence of positive integers and denotes the harmonic fraction of all communicator periods. In practice, time instants are generated by the architecture through clock interrupts. We will assume the following. (1) Time instants are global, i.e., synchronized across all hosts. (2) If a sensor $\mathtt{s}$ is replicated over multiple hosts, then the environment writes identical values to all replications of $\mathtt{s}$ when the update is due. (3) At any time instant, if a communicator $\mathtt{c}$ is updated, then all replications of $\mathtt{c}$ are first updated and then read. The above constraint and exclusion of races ensure that all communicator replications have unique values when they are read. (4) When a task replication $(\mathtt{t}, \mathtt{h})$ completes execution, it broadcasts the output (to be written to a communicator $\mathtt{c}$) to all hosts $\mathtt{hset}/\{\mathtt{h}\}$. Every host receives the values from each replication of $\mathtt{t}$ and stores them in a local memory space (assigned to $\mathtt{c}$). When the update of $\mathtt{c}$ is due, voting is used to decide on the final value to be written to the communicator replication on the host. All tasks are functionally correct and given identical inputs provide identical outputs. All replications of a task have identical input failure models. At any given iteration, the replications of a task either generate $\perp$ (unreliable execution) or the correct value. If some replications generated a non-$\perp$ value, then all other replications which executed reliably generated the identical non-$\perp$ value. If there is at least one non-$\perp$ value, then the communicator replication is assigned that value.

We now formally define the semantics. For $i \geq 0$, let X_i be a function from the communicator set to the set of values, with possibly the empty set, i.e., $X_i : \mathtt{cset} \to \mathtt{type}^{\mathtt{hset}} \cup \emptyset$, where $\mathtt{type} = \cup_{\mathtt{c} \in \mathtt{cset}} \mathtt{type}_\mathtt{c}$. If $i \bmod \pi_\mathtt{c} = 0$, then $X_i(\mathtt{c}) \in {\mathtt{type}_\mathtt{c}}^{\mathtt{hset}}$, otherwise $\emptyset$. A trace is an infinite sequence $(X_i)_{i \geq 0}$ of such functions. The semantics is the set of all possible traces.

Reliability. Given a communicator $\mathtt{c}$, and set $\alpha \in \mathtt{type}_\mathtt{c}^{\mathtt{hset}}$, the value of α is reliable if α contains at least one non-$\perp$ value. A reliability based abstraction consists of only values 0 and 1, where 1 denotes a reliable value, and 0 denotes an unreliable value. Given a trace $(X_i)_{i \geq 0}$, we define the reliability-based abstraction trace $(Z_j)_{j \geq 0} = \rho((X_i)_{i \geq 0})$ as follows: $Z_j : \mathtt{cset} \to \{0, 1\}$ with $Z_j(\mathtt{c}) = 1$ if the set $X_{j \cdot \pi_\mathtt{c}}(\mathtt{c})$ is reliable, 0 otherwise. In other words, the function ρ maps a trace $(X_i)_{i \geq 0}$ to another trace $(Z_j)_{j \geq 0}$; the second trace is referred to as *reliability-based abstract trace*. We define the limit-average value of a reliability-based abstract trace for communicator $\mathtt{c}$, $\tau_\mathtt{c} = (Z_j(\mathtt{c}))_{j \geq 0}$ as the long-run average of the number of 1's in the abstract trace. Formally, the limit-average value $\mathtt{limavg}(\tau_\mathtt{c})$ of a reliability-based abstract trace for communicator $\mathtt{c}$, $\tau_\mathtt{c} = (Z_i(\mathtt{c}))_{i \geq 0}$ is defined as $\mathtt{limavg}(\tau_\mathtt{c}) = \lim_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n-1} Z_i(\mathtt{c})$. Given a communicator $\mathtt{c}$, the *set of reliable abstract traces*, denoted by $\mathit{traces}_\mathtt{c}$, is the set of reliability-based abstract traces for $\mathtt{c}$ with limit-average no less than $\mu_\mathtt{c}$, i.e., $\mathit{traces}_\mathtt{c} = \{\tau_\mathtt{c} : \mathtt{limavg}(\tau_\mathtt{c}) \geq \mu_\mathtt{c}\}$. Given the set of communicators $\mathtt{cset} = \{\mathtt{c}_1, \mathtt{c}_2, \cdots, \mathtt{c}_k\}$, the set of reliable abstract traces is $\mathit{traces}_\mathtt{cset} = \{(Z_j(\mathtt{c}_i))_{j \geq 0} : \forall i. 1 \leq i \leq k. \mathtt{limavg}((Z_j(\mathtt{c}_i))_{j \geq 0}) \geq \mu_{\mathtt{c}_i}\}$.

Analysis. Given an implementation $\mathtt{I}$ for a specification $\mathtt{S}$ on an architecture $\mathtt{A}$, we define the following analyses:

- *Schedulability analysis.* The implementation $\mathtt{I}$ is *schedulable* if (all replications of) all tasks complete execution and transmission (of the outputs) between the read and the write time of the respective task [6].
- *Reliability analysis.* The implementation $\mathtt{I}$ is *reliable* if for each communicator $\mathtt{c}$, the long-run average of the number of reliable values observed at access points of the communicator is at least $\mu_\mathtt{c}$.

An implementation $\mathtt{I}$ is *valid* for a specification $\mathtt{S}$ on an architecture $\mathtt{A}$, if $\mathtt{I}$ is *schedulable* and *reliable*.

3 Reliability Analysis and Refinement

A *specification graph* $\mathcal{G}_\mathtt{S} = (\mathtt{V}_\mathtt{S}, \mathtt{E}_\mathtt{S})$ with $\mathtt{E}_\mathtt{S} \subseteq \mathtt{V}_\mathtt{S} \times \mathtt{V}_\mathtt{S}$ is defined as follows. The set of vertices is $\mathtt{V}_\mathtt{S} = \{(\mathtt{c}, \mathtt{i}) : \mathtt{c} \in \mathtt{cset} \wedge \mathtt{i} \in \{0, \cdots, \pi_\mathtt{S}/\pi_\mathtt{c}\}\} \cup \{\mathtt{t} : \mathtt{t} \in \mathtt{tset}\}$. The set of edges is $\mathtt{E}_\mathtt{S} = \{((\mathtt{c}, \mathtt{i}), \mathtt{t}) : (\mathtt{c}, \mathtt{i}) \in \mathtt{ins}(\mathtt{t})\} \cup \{(\mathtt{t}, (\mathtt{c}, \mathtt{i})) : (\mathtt{c}, \mathtt{i}) \in \mathtt{outs}(\mathtt{t})\} \cup \{((\mathtt{c}, \mathtt{i}), (\mathtt{c}, \mathtt{i}')) : \mathtt{i} < \mathtt{i}' \wedge \forall \mathtt{t} \in \mathtt{tset}. \forall \mathtt{i}''. \mathtt{i} < \mathtt{i}'' \leq \mathtt{i}'. (\mathtt{c}, \mathtt{i}'') \notin \mathtt{outs}(\mathtt{t})\}$. A *communicator cycle* is a path δ from $(\mathtt{c}, \mathtt{i})$ to $(\mathtt{c}, \mathtt{i}')$ such that the path δ contains at least one vertex $\mathtt{t} \in \mathtt{tset}$. A specification $\mathtt{S}$ is *memory-free* if the specification graph $\mathcal{G}_\mathtt{S}$ contains no communicator cycle.

Given the constraints on tasks and assumptions on architecture, environment and semantics, replications of the same task can be assumed to be connected in parallel, and each block of such task replications is connected in series with parallel blocks of replications of other tasks. Given an implementation $\mathtt{I}$, the *reliability of a task* $\mathtt{t}$, $\lambda_\mathtt{t} = 1 - \prod_{\mathtt{h} \in \mathtt{I}(\mathtt{t})}(1 - \mathtt{hrel}(\mathtt{h}))$, i.e., $\lambda_\mathtt{t}$ is the least probability that the task $\mathtt{t}$ executes at every iteration of $\mathtt{t}$.

The *SRG* $\lambda_\mathtt{c}$ *of a communicator* $\mathtt{c}$ is inductively defined as follows: (a) for an input communicator $\mathtt{c}$ we have $\lambda_\mathtt{c} = \mathtt{srel}(\mathtt{s})$, where $\mathtt{c}$ is updated by a sensor $\mathtt{s}$; and (b) for a non-input communicator $\mathtt{c}$ written by a task $\mathtt{t}$ with SRGs given for all communicators in the set $\mathtt{icset}_\mathtt{t}$, we have (1) if $\mathtt{model}_\mathtt{t} = 1$, then $\lambda_\mathtt{c} = \lambda_\mathtt{t} \cdot \prod_{\mathtt{c}' \in \mathtt{icset}_\mathtt{t}} \lambda_{\mathtt{c}'}$, (2) if $\mathtt{model}_\mathtt{t} = 2$, then $\lambda_\mathtt{c} = \lambda_\mathtt{t} \cdot (1 - \prod_{\mathtt{c}' \in \mathtt{icset}_\mathtt{t}}(1 - \lambda_{\mathtt{c}'}))$, and (3) if $\mathtt{model}_\mathtt{t} = 3$, then $\lambda_\mathtt{c} = \lambda_\mathtt{t}$.

With the constraints on task declarations, a communicator $\mathtt{c}$ can be written by a single task. Given an implementation $\mathtt{I}$, at every iteration the probability that $\mathtt{c}$ has a reliable value is at least $\lambda_{\mathtt{c}}$. Hence from the definition of local (or one-step) probabilities we obtain a probability space $Pr^{\mathtt{I}}(\cdot)$ on the set of infinite traces.

Given a memory-free specification, an implementation I is *reliable* if the probability of the set of reliable abstract traces is 1, i.e., $Pr^{\mathtt{I}}[traces_{\mathtt{cset}}] = 1$.

Proposition 1 *Given a memory-free, race-free specification, an implementation is reliable if for all communicators* c*, we have* $\lambda_{\mathtt{c}} \geq \mu_{\mathtt{c}}$.

The proposition can be proved using the strong law of large numbers (SLLN) [5].

General implementation. Consider two tasks t_1 and t_2 that write to two communicators c_1 and c_2, respectively. The LRC of both communicators is 0.9. Let h_1 and h_2 be two hosts with reliability 0.95 and 0.85, respectively. An implementation that maps t_2 to h_2 violates the reliability requirement for c_2, and an implementation that maps t_1 to h_2 violates the reliability requirement for c_1. However, consider a time-dependent implementation that maps the tasks t_1 and t_2 alternately to hosts h_1 and h_2. This implementation is reliable. Our definition of reliability is general enough to allow such time-dependent implementations. However, in this paper, we focus our analysis on implementations that are not time-dependent.

Specification with memory. If the specification graph has a cycle, then the result does not hold for all task models. Consider a task t, with $\mathtt{model}_{\mathtt{t}} = 1$, that reads and writes to a communicator c. Once $\perp$ is written, the value of c is always $\perp$ from that instant on. Hence if $\lambda_{\mathtt{t}} < 1$, then the long-run average of the number of reliable value of c is 0 with probability 1. The solution to the problem is that, for each communicator cycle, there should exists at least one task in the cycle with an independent input failure model.

Refinement. A specification can be *refined* (i.e. replaced) by another, more detailed, *refining* specification if every task in the refining specification maps to a unique task in the refined specification such that no two tasks in the refining specification map to the same task in the refined specification. We will show that, if an implementation is valid for a refined specification and all tasks in the refining specification write to communicators whose LRCs do not exceed the LRCs of the communicators being written by the tasks they map to in the refined specification, then the implementation is valid for the refining specification.

Given two systems $(\mathtt{S},\mathtt{A},\mathtt{I})$ and $(\mathtt{S}',\mathtt{A}',\mathtt{I}')$, let $\mathtt{S} = (\mathtt{tset},\mathtt{cset})$, $\mathtt{A} = (\mathtt{hset},\mathtt{sset},\mathtt{C}_{\mathtt{S}})$, $\mathtt{S}' = (\mathtt{tset}',\mathtt{cset}')$, and $\mathtt{A}' = (\mathtt{hset}',\mathtt{sset}',\mathtt{C}_{\mathtt{S}'})$. Let κ be a *total* and *one-to-one* function from $\mathtt{tset}'$ to tset. The system $(\mathtt{S}',\mathtt{A}',\mathtt{I}')$ *refines* system $(\mathtt{S},\mathtt{A},\mathtt{I})$ under κ, denoted as $(\mathtt{S}',\mathtt{A}',\mathtt{I}') \leq_\kappa (\mathtt{S},\mathtt{A},\mathtt{I})$, if the following set of *refinement constraints* are met: (a) $\mathtt{hset} = \mathtt{hset}'$, (b) for all tasks $\mathtt{t}' \in \mathtt{tset}'$, we require (1) $\mathtt{I}(\mathtt{t}') = \mathtt{I}(\kappa(\mathtt{t}'))$, (2) $\forall \mathtt{h} \in \mathtt{I}(\mathtt{t}') : \mathtt{wemap}(\mathtt{t}',\mathtt{h}) \leq \mathtt{wemap}(\kappa(\mathtt{t}'),\mathtt{h})$ and $\mathtt{wtmap}(\mathtt{t}',\mathtt{h}) \leq \mathtt{wtmap}(\kappa(\mathtt{t}'),\mathtt{h})$, (3) $\mathtt{read}_{\mathtt{t}'} \leq \mathtt{read}_{\kappa(\mathtt{t}')}$ and $\mathtt{write}_{\mathtt{t}'} \geq \mathtt{write}_{\kappa(\mathtt{t}')}$, (4) if $(\mathtt{c}',\cdot) \in \mathtt{outs}_{\mathtt{t}'}$, then $\mu_{\mathtt{c}'} \leq \max_{(\mathtt{c},\cdot)\in \mathtt{outs}_{\kappa(\mathtt{t}')}} \mu_{\mathtt{c}}$, (5) $\mathtt{model}_{\mathtt{t}'} = \mathtt{model}_{\kappa(\mathtt{t}')}$, and, (6) if $\mathtt{t}'$ has input failure model 1, then $\mathtt{icset}(\mathtt{t}') \subseteq \mathtt{icset}(\kappa(\mathtt{t}'))$ and if $\mathtt{t}'$ has input failure model 2, then $\mathtt{icset}(\mathtt{t}') \supseteq \mathtt{icset}(\kappa(\mathtt{t}'))$.

Note that all the constraints are local checks on $\mathtt{t}'$ and $\kappa(\mathtt{t}')$. The refinement relation is reflexive, anti-symmetric and transitive. The following results hold:

Lemma 1 *If* I *is schedulable for* S*, then* I′ *is schedulable for* S′.

Lemma 2 *If* I *is reliable for* S*, then* I′ *is reliable for* S′.

The following result follows from Lemmas 1 and 2.

Proposition 2 *If* $(\mathtt{S}',\mathtt{A}',\mathtt{I}') \leq_\kappa (\mathtt{S},\mathtt{A},\mathtt{I})$ *and* I *is valid for* S *on* A*, then* I′ *is valid for* S′ *on* A′.

4 Example

The reliability analysis is explained through the design of a three-tank-system (3TS) controller. The system consists of three tanks `tank1`, `tank2`, and `tank3`, each with an evacuation tap. Tank `tank3` is connected to both `tank1` and `tank2`. Two pumps, `pump1` and `pump2`, feed water into the tanks `tank1` and `tank2`, respectively. The controller maintains the level of water in tanks `tank1` and `tank2` in the presence and absence of perturbations [8].

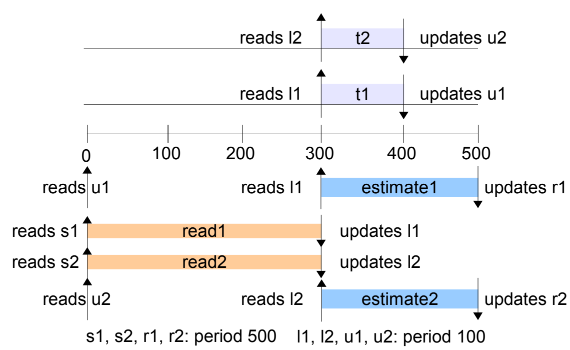

Figure 2. The control tasks

Fig. 2 shows the functionality and timing of the control tasks; the tasks repeat every 500 ms. Task `t1` reads the level `l1` (of `tank1`) and computes the motor current `u1` (for `pump1`). Task `t2` reads the level `l2` (of `tank2`) and computes the motor current `u2` (for `pump2`). Task `read1` (resp. `read2`) computes the level of water in `tank1` (resp.`tank2`) from sensor `s1` (resp. `s2`). Tasks `estimate1` and `estimate2` estimate the perturbations `r1` (for `tank1`) and `r2` (for `tank2`), respectively. Tasks `read1` and `read2` have input failure models 2; all other tasks have failure model 1.

The architecture consists of three hosts h1, h2, and h3. We do not have reliability data for our experimental platform, however, for illustration purposes, we assume all host and sensor reliabilities to be 0.999. The implementation maps task t1 (resp. t2) to host h1 (resp. h2), and the rest to host h3.

Each task is mapped to one host; thus the reliability of each task is the same as the reliability of its host. The SRGs of the communicators are computed as follows. The SRGs $\lambda_{\mathtt{s1}}$ and $\lambda_{\mathtt{s2}}$ are the same as the sensor reliability, i.e., 0.999. From reliability analysis it follows that $\lambda_{\mathtt{l1}} = \lambda_{\mathtt{read1}} \cdot \lambda_{\mathtt{s1}} = 0.998$ and $\lambda_{\mathtt{u1}} = \lambda_{\mathtt{l1}} \cdot \lambda_{\mathtt{t1}} = 0.997$ and, similarly, $\lambda_{\mathtt{l2}} = 0.998$ and $\lambda_{\mathtt{u2}} = 0.997$. If the LRCs $\mu_{\mathtt{u1}}$ and $\mu_{\mathtt{u2}}$ are 0.99, then the above implementation is reliable with respect to the requirements.

By contrast, if the desired LRCs $\mu_{\mathtt{u1}}$ and $\mu_{\mathtt{u2}}$ are set to 0.9975, then the above implementation is not reliable. We will analyze two scenarios for meeting the new requirements. In the first scenario, the tasks t1 and t2 are mapped to both hosts h1 and h2, respectively. The reliability of the task t1, as well as t2, is modified to $1 - (1 - 0.999)^2 = 0.999999$. In turn, this changes the SRGs of u1 and u2 to 0.997999, which meet the LRCs. In the second scenario, the sensors are replicated, i.e., tasks read1 and read2 read from two sensors each; the reliability of each sensor is 0.999. The SRGs of the communicators l1 and l2 are now $\lambda_{\mathtt{l1}} = \lambda_{\mathtt{read1}} \cdot (1 - (1 - 0.999)^2) = 0.998999$ ($\mathtt{model}_{\mathtt{read1}} = 2$); and $\lambda_{\mathtt{l2}} = 0.998999$. This changes the SRGs of u1 and u2 to 0.998, which again meet the LRCs.

Implementation in HTL. We ran experiments with a real version of the 3TS controller written in the Hierarchical Timing Language (HTL) [6]. The control tasks were distributed over multiple hosts. To validate the fault tolerance assumptions used in the reliability analysis, we unplugged one of the two hosts from the network and verified that there was no change in the control performance of the system. To account for replication, the code generation technique [6] is modified as follows. The output of each (replication of a) task is sent to all other hosts. Each host then performs a voting routine on the received data to determine, if possible, the correct value, which is then stored in the local communicators. Refer to htl.cs.uni-salzburg.at for more details. In the example, there are mode switches between tasks, but the switch is always to tasks with identical reliability constraints, and the reliability analysis of Section 3 applies.

5 Conclusion

We proposed a separation-of-concerns approach for the joint schedulability and reliability analysis of safety-critical real-time embedded applications. The main contribution is the separation of reliability requirements in a specification (possibly written in a task coordination language), from the reliability guarantees offered by hosts and communication links. The implementation (replication mapping and scheduling) must ensure that all timing and reliability requirements of the specification are met.

Acknowledgments. This work was supported in part by the GSRC grant 2003-DT-660, the NSF grant CCR-0208875, the HYCON and Artist II European Networks of Excellence, the European Integrated Project SPEEDS, the SNSF NCCR MICS, the Austrian Science Fund Project P18913-N15, General Motors, United Technologies Corporation, and the CHESS at UC Berkeley, which is supported by the NSF grant CCR-0225610, the State of California Micro Program, and Agilent, DGIST, Hewlett Packard, Infineon, Microsoft, and Toyota.

References

[1] I. Assayad, A. Girault, and H. Kalla. A bi-criteria scheduling heuristics for distributed embedded systems under reliability and real-time constraints. In *DSN*. IEEE, 2004.

[2] M. Baleani, A. Ferrari, L. Mangeruca, M. Peri, S. Pezzini, and A. Sangiovanni-Vincentelli. Fault-tolerant platforms for automotive safety-critical applications. In *CASES*. ACM, 2003.

[3] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), 1991.

[4] W. P. Dotson and J. O. Gobien. A new analysis technique for probabilistic graphs. *IEEE Trans. Circuits and systems*, 10, 1979.

[5] R. Durrett. *Probability: Theory and Examples*. Duxbury Press, 1995.

[6] A. Ghosal, T. A. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli. A hierarchical coordination language for interacting real-time tasks. In *EMSOFT*. ACM, 2006.

[7] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. GIOTTO: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91, 2003.

[8] D. Iercan. Tsl compiler. Technical report, 'Politehnica' University of Timisoara, 2005.

[9] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *DATE*. ACM, 2005.

[10] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant embedded systems with checkpointing and replication. In *Intl. Work. on Electronic Design, Test and Applications*. IEEE, 2006.

[11] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Synthesis of fault-tolerant schedules with transparency/performance trade-offs for distributed embedded systems. In *DATE*, 2006.

[12] D. Kececioglu. *Reliability Engineering Handbook*, volume 2. Prentice Hall, 1991.

[13] C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *DATE*. ACM, 2004.

[14] S. Rai and A. Kumar. Recursive technique for computing system reliability. *IEEE Trans. on Reliability*, 36, 1987.

[15] A. Sangiovanni-Vincentelli and et al. Benefits and challenges for platform-based design. In *Proc. DAC*, volume 91. ACM, 2004.