In Proc. of the 19th Euromicro Conference on Real-Time Systems (ECRTS'07), 2007.
Revised version.

Composition Techniques for Tree Communication Schedules *

Madhukar Anand, Sebastian Fischmeister and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
anandm@cis.upenn.edu, sfischme@seas.upenn.edu, lee@cis.upenn.edu

Abstract

A critical resource in a distributed real-time system is its shared communication medium. Unrestrained concurrent access to the network can lead to collisions that reduce the system's reliability. Therefore in this area, one goal is to develop effective models for coordinating and controlling access to the shared medium and its channels.

Network Code is a verifiable, executable model for coordinating and controlling access to a shared communication medium in a distributed real-time system. In this paper, we investigate the problem of building an application by composing multiple Network Code programs. To reason about the composition, we model Network Code programs as Tree Schedules (TS) and then consider the composition of schedules that describe how the network is accessed by different applications. Specifically, we first define the notions of compatibility and composability of tree schedules, and then provide algorithms for their composition and reason about overhead of composition. We illustrate the techniques by considering the composition of two control applications.

1 Introduction

Modern real-time systems realize distributed applications with timeliness requirements. These systems typically manifest at the border between the physical and the logical world, and as such, allow us to manipulate our physical world based on programmed algorithms. Consequently, such systems must be extremely reliable because logic and timing errors no longer stop at our workstation, but can cause physical damage to the equipment, environment, and humans.

A critical resource in a distributed real-time system is its shared communication medium. Because of the system's decentralized nature in most networks, any connected node can access it anytime and can cause collisions in the network communication, which scrambles data and typically results in retransmissions. Since such collisions reduce the system's reliability, one primary research goal in this area is to provide effective and flexible coordination models for controlling access to the shared medium and its channels.

This has been the motivation to introduce the Network Code [14] as a verifiable, executable coordination model for controlling access to a shared communication medium in a distributed real-time system. In some cases, this model of programming communication enables increased throughput by skipping unnecessary communication compared to the standard offline methods [3]. The application-specific program is encoded in the domain-specific language Network Code. The programs are interpreted by a runtime environment that executes the instructions. Such a runtime environment has been implemented for RTLinuxPro [14], PIC18F2X8X [18], and as an FPGA [20]. One salient aspect of Network Code is that its programs can be translated to formal specifications, which can be model-checked to verify aspects of reliability such as absence of collisions, overhead, schedulability, and integrity (e.g., sender/receiver pairing, content typing, over/underflows) as shown in [14].

A Network Code program describes the network supply for a particular application. An important feature of Network Code is the option of specifying several different network supplies based on suitable conditions. This feature has naturally allowed us to capture the underlying structure of a Network Code program as a tree. The program encodes when a node gets a slot and when the node may transmit data. Thus, the program specifies the supply of the network resource to the application. Since the program can contain conditional branches encoded in IF instructions, a periodic supply model (c.f., [17, 9]) would have to overapproximate. Encoding the supply in a tree structure (called tree schedule) has proved to be a well suited abstraction [3], which can also encode this conditional branching without loss of generality. This differs from the recurring branching task models [6, 7] as it specifies a supply and not a demand.

Figure 1 provides an overview of the development pro-

*This research was supported in part by NSF CNS-0509143, NSF CCF-0429948, NSF CNS-0509327, ARO - DAAD19-01-1-0473, and OEAW APART-11059.

cess. Each application has its own requirements that may be specified using an arbitrary task model. From these requirements, we can generate a demand tree schedule. The demand tree schedule specifies how much resource the application requires (in our case, the network). In addition, each application also uses a resource model that describes the resource and places limits and constraints on its availability (e.g., a periodic resource model may periodically provide Θ amount of resource every Π time units). Together with the resource model, demand tree schedule leads to a supply tree schedule which describes how much resource the application will actually get at run time. If this supply tree meets the application's demands, then we have a valid tree schedule for the particular application. If it does not, then we have to generate a different supply tree or may have to alter the application requirements and repeat this process.

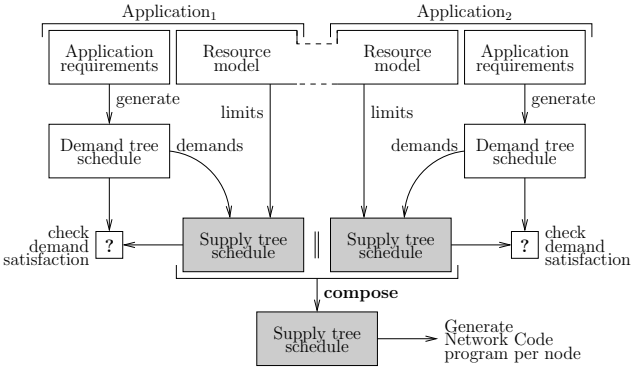


Figure 1. Composition of tree schedules.

In this paper, we investigate the problem of composing two tree schedules at the supply level. This problem occurs whenever two or more applications use this programmable coordination model and share the same resource. Consider the situation shown in Figure 1: There are two applications, each one produces its own supply tree schedule, however, since both share the same resource we need to compose them to generate one combined supply tree schedule. To provide true compositionality, we want to guarantee that the composed tree schedule still meets the demands of each individual application; therefore, there is no additional *"check demand satisfaction"* in the figure after the composition.

Overview of Composition Procedure. The tree schedules (TS), aside from encoding conditional release of messages, also have the provision to specify equivalent schedules, all of which meet the application demand. The advantage of having such equivalent schedules is apparent when composing tree schedules; given two schedules Ω_1 and Ω_2, we pick a schedule each from Ω_1 and Ω_2 such that they are mutually compatible (there are no collisions) and meet demands of the applications being composed. Figure 2 provides the overview of our procedure for composing tree schedules. Our composition operator ($\|$) requires the schedules to take transitions at the same time (time compatibility). While this condition is not always necessary for composition, it keeps the analysis simple. Our composition procedure consists of two parts: First, it tries to transform the input schedules and make them time compatible and composable. Second, it composes the schedules. The transformation again consists of two steps: In the first step, we make two input tree schedules time compatible so they take transitions at the same time. This behavior is implemented by the procedure TMC(). If the procedure succeeds then the outputs will be time compatible. In the second step, we make two tree schedules composable by removing optional parts of the tree schedule that cause collisions on the network after composition. The procedure REDUC() implements this behavior and prunes the set of equivalent schedules of all the uncomposable schedules. If the procedure succeeds, then the outputs are time compatible and composable.

If either procedure fails, then the input tree schedules cannot be composed without altering their timing behaviors. Here, we resort to composing them via temporal isolation. Temporal isolation is achieved here by using bandwidth restrictions to transform the input schedules into composable ones. Under this scheme, each application is allocated several network slots depending on their proportion of the total bandwidth. The procedure PROP_FIT() implements this behavior. Although PROP_FIT() makes the schedules composable, it may introduce additional delays that alters timing behavior. This could cause deadlines to be missed; however, it still preserves other properties such as the causal ordering of messages and tree's branching structure.

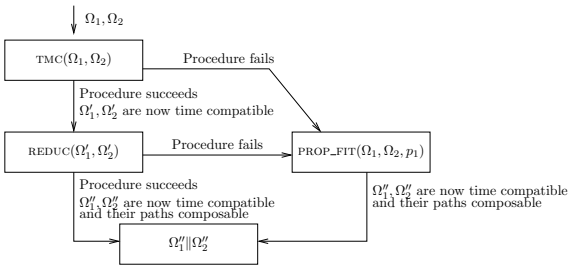


Figure 2. Flow diagram for composition of TS.

2 Tree Schedule and Definitions

Informally, a tree schedule is a structure consisting of locations and transitions between these locations such that its underlying graph is a directed tree. Each location of the tree schedule may specify a transmission on the shared network, and each transition is guarded by a condition that needs to

be met before proceeding to the next location. A location could also be empty, in which case no transmission but the passage of time occurs.

2.1 Definition of a Tree Schedule

Definition 1 (Tree Schedule) *A* Tree Schedule *(TS)* Ω *is a tuple* $\langle V, \mathcal{V}, sl, K, T\rangle$ *where,*

- V *is a set of locations,*
- $\mathcal{V}$ *is a set of variables,*
- $sl : V \hookrightarrow \mathcal{V} \cup \{\epsilon\} \times B$ *specifies what variable is transmitted in this location and a clock constraint,*
- K *is a set of clocks* $|K| \geq 1$*,*
- $T \subseteq V \times G \times 2^K \times 2^V$ *is a set of transitions that depend on an enabling condition* G*,*

such that the underlying graph (V, T) *is a directed tree.*

In the tree schedule, we denote the root location by $v^0 \in V$. The set of leaf locations is denoted by $V^F \subseteq V$. The mapping sl defines for each location a variable (or ϵ) and a set of clock constraints on that location. Clocks are as defined in the Timed Automata [1], and the clock constraint $b \in B$ is of the form $k = q$ where k is a clock and $q \in \mathbb{Z}^+$.

The set of transitions $T \subseteq V \times G \times 2^K \times 2^V$ connects locations to sets of descendant locations. A transition from location v to v' is guarded by an enabling condition $g \in G$, resets a set of clocks $s \subseteq 2^K$, and is denoted by $v \xrightarrow{g,s} v'$. The enabling condition is any decidable function over the variables $\mathcal{V}$. The enabling conditions $g_v^1, \ldots, g_v^m$ of transitions leaving one location v must satisfy the following conditions: (1) any two enabling conditions g_v^i and g_v^j are mutually exclusive and (2) the set of enabling conditions is exhaustive, i.e., $\bigvee_{j=1}^{m} g_v^j = \text{true}$. These conditions ensure that the schedule always makes progress, and a reset will always occur eventually. In practice, the enabling conditions are typically functions of the state of the schedule, local variables, and transmitted values.

The set T is partitioned into sets T_m and T_r. The set T_m contains all transitions that have exactly one destination (i.e., they are of the form[1]: $V \times G \times 2^K \times V$). Transitions in the set T_m are called *minimum transitions.* The set $T_r = T \setminus T_m$ contains all transitions that have more than one destination. Such transitions are called *reducible transitions.* Each node can have multiple outgoing transitions, and each transition could lead to multiple locations; here, the system can choose to continue in one of the locations. This mechanism may encode alternative, equivalent schedules which are all acceptable to the application. Two schedules are equivalent, if they both satisfy the application demand. For example, if an application requires nine slots out of ten, then the two schedules, where one provides ten out of ten the other provides nine out of ten, are equivalent.

[1]By V, we actually mean singleton sets but we write them as single nodes to simplify notations when they singleton node sets.

Definition 2 (Path) *A* path $\text{path}_\Omega(v^m, v^{m+n})$ *is a sequence of locations of a tree schedule* Ω*:* $v^m \xrightarrow{g_{m+1},s_{m+1}} \ldots \xrightarrow{g_{m+n},s_{m+n}} v^{m+n}$*, where* $\forall i, 1 \leq i \leq n : \langle v^{m+i-1}, g_{m+i}, s_{m+i}, v^{m+i}\rangle \in T$*. A* complete path *is* $\text{path}_\Omega(v^0, v)$ *with* $v \in V^F$*; the set of all complete paths is denoted by* $\mathcal{P}$*.*

If two complete paths are such that they contain a common prefix, and the first transition where they differ is a reducible transition, then they are called *equivalent* paths. We denote two equivalent paths p_1 and p_2 by $p_1 \equiv p_2$. The *duration of a path*, $\text{dur}(p)$ with $p = \text{path}(v^m, v^{m+n})$, is the time it takes to transit from v^m to v^{m+n}. If all complete paths of Ω have the same duration, then we say that Ω is a *isochronous* tree schedule. Otherwise, it is said to be *anisochronous*. Finally, we define the probability of $\text{path}(v^m, v^{m+n})$ as the probability of reaching v^{m+n} starting from v^m. This probability is useful to reason about composition of anisochronous tree schedules where there is no fixed period of recurrence of the start location.

Execution Semantics. The execution semantics of a tree schedule as it is executed in the network layer (see [14]) is as follows: The tree schedule starts execution at v^0 with all clocks set to 0. The variables in $\mathcal{V}$ are assigned to some default value by the user. The network layer then transmits variables as specified in schedule's mapping sl (or if this is ϵ, then it remains idle). The network layer remains in the current location v till it can make a suitable transition. This happens when the clock constraint of the current location evaluates to true. The transition is made to one of v's children (recall that exactly one g will be enabled by definition). The decision about which transition is taken, is made by first evaluating all the enabling conditions and then making a transition to the one that is enabled. If this transition leads to multiple locations (representing equivalent schedules), then the system chooses the first one that is stored in the list of destination locations. This execution is continued until a leaf location is reached. In the leaf location ($v \in V^F$), the schedule *resets* immediately after the clock constraint becomes true. This means that (1) the schedule starts again at the root v^0 and (2) all the clocks K are set to 0 during the reset. Variables get updated at run time through messages. If one node transmits a new value for a variable, then all nodes use this new value for evaluating enabling conditions. For more details about this, we refer the reader to Fischmeister et al [14].

Example. Assume a distributed real-time system in which we have to communicate one sensor value. Data integrity requirements specify that the system must tolerate

failures at the sensor reading hardware. We assume that the variation in the reading is bounded by δ unless a fault occurs, , and we assume that the communication medium between the units is reliable. Our approach requires the sensor and three sensor reading units. The sensor produces the value, and each unit reads it via polling. The units then need to agree on what the real sensor value is. To implement this, we need three values x_1 to x_3. The units use these values to communicate each one's sensor reading. With our assumptions, a simple majority vote is sufficient. Figure 3 shows the resulting tree schedule for this example. In the figure, (x, n) represents the variable being transmitted (x) and the time spent in that location (n). The two branches A and B only differ in the ordering of the initial two sensor readings. Note, that if two values already create a decisive vote, then it is unnecessary to communicate the third one.

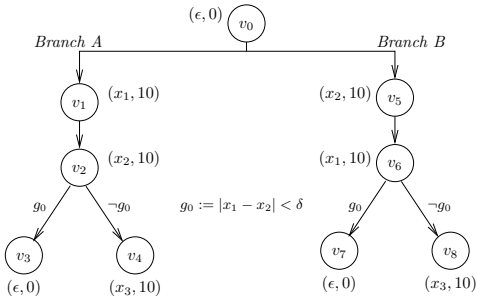


Figure 3. Example tree schedule.

We can encode the application by the TS $\langle V, \mathcal{V}, sl, K, T\rangle$:

- $V = \{v_0, v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8\}$
- $\mathcal{V} := \{x_1, x_2, \delta\}$, $K = \{k\}$,
- sl as shown in the figure for each location v,
- $T_r = \{\langle v_0, \emptyset, \{k\}, \{v_1, v_5\}\rangle\}$, and T_m contains all other transitions.

The guard $g_0 := |x_1 - x_2| < \delta$ checks whether the first two sensor readings are already within a bound δ. The clock k is reset on every transition.

2.2 Compatibility and Composability

In this section, we introduce notions of compatibility and composability. First, we define these for paths and then extend them for tree schedules. We assume that both tree schedules start execution at the same time and that the clock variables of two tree schedules are disjoint. For the remainder of this section, we denote two tree schedules by Ω_1 and Ω_2 where $\Omega_n = \langle V_n, \mathcal{V}_n, sl_n, K_n, T_n\rangle$, $n = 1, 2$, and $K_1 \cap K_2 = \emptyset$. We also denote paths of the tree schedule Ω_n by $p_n \equiv \text{path}_{\Omega_n}(v_n^i, v_n^j) = v_n^i \xrightarrow{g_n^{i+1}, s_n^{i+1}} \cdots \xrightarrow{g_n^j, s_n^j} v_n^j$, $n = 1, 2$.

The first notion of compatibility is *time* compatibility which means that the two paths have same timing behavior i.e., they make transitions at the same time.

Definition 3 (Time Compatibility of Paths) *Two paths p_1 of Ω_1 and p_2 of Ω_2 are said to be* time compatible*, denoted $p_1 \sim_K p_2$, if they take transitions at the same time, i.e., $\forall l, i \le l \le j : (sl_1(v_1^l).b = \text{true}) \Leftrightarrow (sl_2(v_2^l).b = \text{true})$ where '.b' denotes the projection operation onto the clock constraint.*

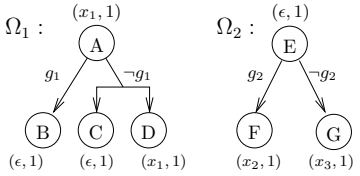


Figure 4. Example for Section 2.2

Consider for example the two schedules shown in Figure 4. For ease of explanation, we name our locations alphabetically. The enabling conditions are defined as $g_1 := x_1 < 5$ and $g_2 := g_1$. We use the following shorthand notation: a path is specified by a sequence of letters (e.g., AB is the path $A \to B$). If all the locations have a timing constraint $K = 1$ and clock K is reset on every transition, then all complete paths of Ω_1 are time compatible with those of Ω_2.

The second type of compatibility we define is called transition compatibility. If two transitions are guarded by mutually exclusive enabling conditions, then they would never be taken together. Therefore, we do not have to consider their concurrent execution. For the two schedules shown in Figure 4, paths AB and EF are transition compatible, whereas AB and EG are not. Note, that to check for transition compatibility, we assume that the two trees are time compatible. While this is not a necessary restriction, we introduce it as it keeps the analysis simple.

Definition 4 (Transition Compatibility of Paths) *Two time compatible paths p_1 of Ω_1 and p_2 of Ω_2 ($p_1 \sim_K p_2$) are said to be* transition compatible*, denoted $p_1 \sim_T p_2$, if $\forall l, i < l \le j : g_1^l \wedge g_2^l$ is true.*

Finally, we define that two paths are composable if they are transition compatible and that there are no collisions. For the example in Figure 4, paths AD and EG are transition compatible, but not composable as there is a message release at locations D and G. Transition compatible paths AC and EG are however composable.

Definition 5 (Composability of Paths) *Two transition compatible paths p_1 of Ω_1 and p_2 of Ω_2 ($p_1 \sim_T p_2$)*

are said to be composable*, denoted* $p_1 \approx p_2$*, if* $\forall l, i \leq l \leq j : (sl(v_1^l).\mathcal{V} = \epsilon) \vee (sl(v_2^l).\mathcal{V} = \epsilon)$ *where '.$\mathcal{V}$' denotes the projection operation onto the variable being transmitted.*

Given two composable paths, we can define the schedule that results from their concurrent composition.

Definition 6 (Composition of Paths) *Consider complete and composable paths* $p \equiv \text{path}_{\Omega_1}(v_1^0, v_1^m)$ *and* $q \equiv \text{path}_{\Omega_2}(v_2^0, v_2^m)$ *(*$p \approx q$*). We define the concurrent composition of p and q as the schedule* $\Omega_{p\|q} = \langle V, \mathcal{V}, sl, K, T\rangle$ *where,*

- $V = \{\langle v_1^i, v_2^i\rangle | i \in [0, m]\}, v_1^m \in V_1^F, v_2^m \in V_2^F$,
- $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, $K = K_1 \cup K_2$,
- $T = \{\langle\langle v_1^{i-1}, v_2^{i-1}\rangle, \langle g_1^i, g_2^i\rangle, \langle s_1^i, s_2^i\rangle, \langle v_1^i, v_2^i\rangle\rangle | i \in [1, m)\}$
- *for* $i \in [0, m)$,

$$sl(\langle v_1^i, v_2^i\rangle) = \begin{cases} sl(v_1^i) & sl(v_1^i).\mathcal{V} = \epsilon \wedge sl(v_2^i).\mathcal{V} = \epsilon \\ sl(v_1^i) & sl(v_1^i).\mathcal{V} \neq \epsilon \wedge sl(v_2^i).\mathcal{V} = \epsilon \\ sl(v_2^i) & sl(v_1^i).\mathcal{V} = \epsilon \wedge sl(v_2^i).\mathcal{V} \neq \epsilon \end{cases}$$

Additionally, $v^0 = \langle v_1^0, v_2^0\rangle$ *and* $V^F = \{\langle v_1^m, v_2^m\rangle\}$.

With the definitions of compatibility and composability for paths, we can define these notions for trees. Two trees are said to be *time* compatible if every path of one tree is time compatible with every path of the other tree.

Definition 7 (Time Compatibility of TS) *Two tree schedules* Ω_1 *and* Ω_2 *are said to be* time compatible*, denoted* $\Omega_1 \sim_K \Omega_2$*, if* $\forall p_i \in \mathcal{P}_1, p_j \in \mathcal{P}_2, p_i \sim_K p_j$ *where* $\mathcal{P}_n$ *denotes the set of complete paths in* $\Omega_n, n = 1, 2$.

For our example in Figure 4, Ω_1 and Ω_2 are time compatible. We define transition compatibility to mean that there exists at least one path each in Ω_1 and Ω_2 that can run concurrently and be transition compatible. Again, both Ω_1 and Ω_2 are transition compatible.

Definition 8 (Transition Compatibility of TS) *Two time compatible Tree schedules* Ω_1 *and* Ω_2 *(*$\Omega_1 \sim_K \Omega_2$*) are said to be* transition compatible*, denoted* $\Omega_1 \sim_T \Omega_2$*, if* $\exists p_i \in \mathcal{P}_1, p_j \in \mathcal{P}_2, p_i \sim_T p_j$ *where* $\mathcal{P}_n$ *denotes the set of complete paths in* $\Omega_n, n = 1, 2$*. Further, we define the set* $\mathcal{P}_{12} = \{\langle p_i, p_j\rangle | p_i \in \mathcal{P}_1, p_j \in \mathcal{P}_2, p_i \sim_T p_j\}$.

Finally, before we define the composition of two tree schedules, we need to define the union of two tree schedules. We assume that the union of two trees is defined for tree schedules that have the same start location, and that the locations in their intersection release the same message and have the same timing constraint.

Definition 9 (Union of TS) *We define the union of two tree schedules* Ω_1 *and* Ω_2 *if* $v_1^0 = v_2^0$ *and* $\forall v \in V_1 \cap V_2, sl_1(v) = sl_2(v)$*. The union, denoted by* $\Omega_1 \cup \Omega_2$*, is given as the tree schedule* $\langle V_1 \cup V_2, v^0, V_1^F \cup V_2^F, \mathcal{V}_1 \cup \mathcal{V}_2, sl, K_1 \cup K_2, T_1 \cup T_2\rangle$ *where* $sl(v) = sl_1(v)$ *if* $v \in V_1$ *and* $sl(v) = sl_2(v)$ *if* $v \in V_2$.

2.3 Composition of Tree Schedules

A tree schedule describes the actual communication behavior of an application. The concurrent composition of tree schedules represents the running of both applications concurrently. Therefore, the set of acceptable schedules under the composition would be the ones that are acceptable by both the underlying tree schedules and are mutually compatible.

Definition 10 (Composition of TS) *Two transition compatible Tree schedules* Ω_1 *and* Ω_2 *(*$\Omega_1 \sim_T \Omega_2$*) are said to be* composable*, denoted* $\Omega_1 \approx \Omega_2$*, if* $\forall \langle p_i, p_j\rangle \in \mathcal{P}_{12}$*, we have that* $p_i \approx p_j$*. The concurrent composition of trees* $\Omega_1 \| \Omega_2$ *is then given by,* $\bigcup_{\langle p_i, p_j\rangle \in \mathcal{P}_{12}} \Omega_{p_i \| p_j}$.

The composition of two tree schedules is determined by this three step procedure: (1) compute all complete paths for each tree schedule, group equivalent paths, and combine all paths, (2) eliminate paths that are unreachable or cause collisions and check that at least one path from each group of equivalent paths causes no collision, (3) take the union of all paths. We demonstrate the composition procedure using the schedules in Figure 4. We assume that the letters in brackets show the current location in each tree schedule with all locations having globally unique identifiers (e.g., (AE) specifies that Ω_1 resides in location A and Ω_2 in location E). Therefore (AE)(BF) specifies the concurrent paths. $A \rightarrow B$ and $E \rightarrow F$.

Step 1: The complete paths of Ω_1 are: AB, {AC, AD}. The paths AC and AD are grouped, because they stem from a reducible transition. Ω_2 has the paths EF and EG. Now paths are combined by simultaneously executing each path. This results in (AE)(BF), (AE)(BG), (AE)(CF), (AE)(CG), (AE)(DF), and (AE)(DG).

Step 2: First, we statically check for each path, whether the joint guards in each cause a contradiction. If so, we remove them. In our example since $g_1 = g_2$, $g_1 \wedge \neg g_2$ and $\neg g_1 \wedge g_2$ cause contradictions, so we remove the paths (AE)(BG), (AE)(CF), (AE)(DF). Furthermore, a path may cause collisions, if locations of both tree schedules communicate data. These paths are eliminated from the set. Given the example, the path (AE)(DG) must be removed, because of a collision in (DG). Since that path stems from a reducible transition, we check whether we can remove it. As the other path (AE)(CG) does not cause collisions, we can remove (AE)(DG) and preserve the transition in Ω_1. If

we could not remove the path or if the path stems from a minimal transition, then the composition fails at this point. The remaining paths are: (AE)(BF) and (AE)(CG).

Step 3: We now create the union of all paths using the union operation as defined for tree schedules in Section 2.2.

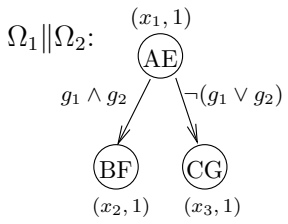


Figure 5. $\Omega_1 \| \Omega_2$ for schedules in Fig. 4.

3 Time Compatibility by Unrolling

One prerequisite for composition of tree schedules is that they are time compatible. Algorithm 1 describes how two schedules can be made time compatible by unrolling both of them to the same fixed duration and synchronizing their decision points.

The Algorithm. In the algorithm, the variable l_i represents the expected duration of compete paths in each tree schedule. We assume that exactly one transition is selected from reducible transitions to compute this length. The variable hyp represents the hyper period.

The algorithm consists of three main parts: in part A, it unrolls the schedules until the duration on all paths is equal or longer than the hyper period using the procedure APPEND(). This procedure attaches one tree schedule to the leaves of another. Specifically, it adds the root node of the new instance of Ω_i to the temporary tree schedule, adds transitions from all leaves to it, and finally takes the union of both trees. In Part A, the algorithm also computes the set $\mathcal{R}_i$ that contains all former root locations of attached trees.

In Part B, the algorithm tailors the schedules to the hyper period by pruning subtrees that are too long using the procedure PRUNE(). This procedure finds v' at which it can prune the path. This may be at a former root location or one branch of a reducible transition (hence, $v' \in \mathcal{R} \cup T_r.4$)[2]. When tailoring the tree, the optimum is to cut off as little as possible. Thus, the v' that leads to the optimum is the one that has the longest duration of the path $p(v^0, v')$ of all possible locations in question. Then, the procedure prunes the subtree rooted at that location. However, pruning may introduce paths that are shorter than hyp. If this is the case, the algorithm attaches blanks to these branches until the hyp. A blank is a slot of length one which has no scheduled communication, thus $(\epsilon, 1)$.

In Part C, the algorithm synchronizes the timing of transitions in both schedules by calling SYNC_TRANS(). This procedure performs a simple breath-first search in which it checks that for each location in all complete paths of one tree, there exists a location in all complete paths of the other tree whose clock constraint becomes true simultaneously. If such a location is missing on a particular path, then SYNC_TRANS() introduces it by duplicating the location that is closed but prior that time and adding a transition between the duplicated nodes. Specifically, if the schedules enter v_1 and v_2 concurrently and the first schedule spends time t_1 in a location v_1 while other spends $t_2 > t_1$ time in location v_2, we split the location v_2 into v_2' and v_2'' such that time spent in v_2' is t_1 and in v_2'' is $t_2 - t_1$. We also add the transition $\langle v_2', \emptyset, \emptyset, v_2'' \rangle$. For practical purposes, we limit transitions only to be a multiple of the packet size. Therefore, we can

[2] ".4" specifies a projection on the fourth element of the tuple.

Algorithm 1 Make Ω_1, Ω_2 time compatible by unrolling.

Require: $l_i = \sum_r \mathrm{dur}(r_i) \cdot p_r(p_i)$ where $p_i \in P_i$ with $i = 1, 2$
Require: $hyp \leftarrow LCM(l_1, l_2)$
Ensure: $\Omega_1 \sim_K \Omega_2$
  procedure TMC($\Omega_1, \Omega_2, hyp, \mathcal{R}_1, \mathcal{R}_2$)
    for $i = 1, 2$ **do** *// Part A*
      $\Omega^* \leftarrow \Omega_i$
      while $\exists p \in P^* : \mathrm{dur}(p) < hyp$ **do**
        APPEND(Ω^*, **new instance of** Ω_i)
      end while
      $\mathcal{R}_i \leftarrow \{v | v$ was root in $\Omega_i\}$
      $\Omega_i \leftarrow \Omega^*$
    end for
    for $i = 1, 2$ **do** *// Part B*
      for all $v \in V_i^F$ **with** $dur(p(v^0, v)) > hyp$ **do**
        PRUNE($\Omega_i, \mathcal{R}_i, v, hyp, \mathrm{true}$)
      end for
    end for
    // Part C
    SYNC_TRANS(Ω_1, Ω_2) *// bfs with node duplicating*
  end procedure

  procedure APPEND(Ω_*, Ω_+)
    $V_* \leftarrow V_* \cup v_+^0$
    for all $v \in \{V_*^F \setminus v_+^0\}$ **do**
      $T_* \leftarrow T_* \cup \langle v, \emptyset, K, v_+^0 \rangle$
    end for
    $\Omega_* \leftarrow \Omega_* \cup \Omega_+$
  end procedure

  procedure PRUNE($\Omega, \mathcal{R}, v, hyp, pad$)
    $\mathcal{L} \rightarrow (\mathcal{R} \cup T_r.4) \cap \{v'' | v'' \in p(v^0, v)\}$
    find opt. $v' \in \mathcal{L}$ with $\mathrm{dur}(p(v^0, v')) \leq hyp$
    if no v' found **then**
      return **failed**
    end if
    prune subtree rooted at v'
    if $pad = \mathrm{true} \wedge \mathrm{dur}(r(v^0, v')) < hyp$ **then**
      append blanks until hyp
    end if
  end procedure

packetize messages and thus perform this procedure. In the algorithm, the variable l_i represents the expected duration of compete paths in each tree schedule. The variable hyp represents the hyper period.

Example. Figure 6 illustrates Algorithm 1 with unrolling and pruning of one schedule. The tree schedule is simplified on purpose to show only the necessary level of detail. Consider the tree schedule Ω and a hyper period of six. In the unrolling part, the algorithm unrolls Ω until all runs are longer or equal to the hyper period. In the pruning part, the algorithm prunes all branches that are longer than hyp. In case A, it removes the subtree based on the reducible transition in v_3. In case B, it removes the subtree based on the root locations v_4 and v_5.

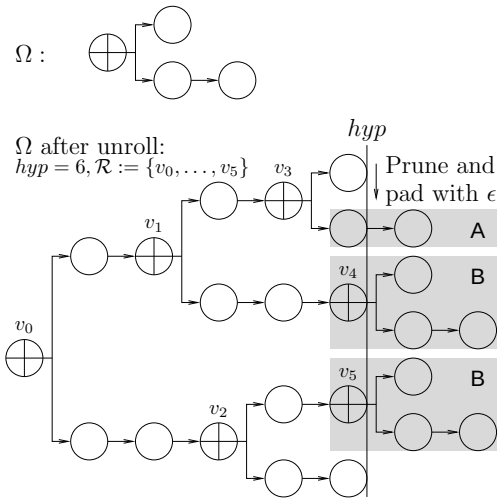


Figure 6. Example for Algorithm 1

Static Overhead. Tailoring the tree schedule to the hyper period introduces blanks. We quantify the number of all inserted blanks as static overhead. This static overhead can be computed as follows: Let w be the amount of slots allocated to a tree schedule Ω, and l_i be the expected path lengths. If the schedule is unrolled to the length of w, the trailing empty slots are inserted along paths whose length is at least $w + 1 - \max_i l_i$. The number of trailing empty slots in the tree component schedule is given by $\sum_{j=w-l+1}^{w}(w-j)h(j)$ where $h(j) = \sum_{(k_1,\ldots,k_r)} \frac{(\sum_i k_i)!}{\prod_i k_i!}$ s.t. $\sum_i l_i k_i = w - j$. To see why this is correct, observe that a path in the final schedule is made up of paths of the original schedule. With k_i subpaths of length l_i, it is possible to generate $\frac{(\sum_i k_i)!}{\prod_i k_i!}$ paths of the final schedule (the number of permutations of k_i's). We can also compute the expected number of trailing empty slots (i.e., the average overhead), given the probabilities of transition. If the probability of taking a path of length l_i is π_i, then the expected number of trailing empty slots is $\sum_{j=w-l+1}^{w}(w-j)h'(i)$ where $h'(j) = \sum_{(k_1,\ldots,k_r)} \left(\frac{(\sum_i k_i)!}{\prod_i k_i!} \cdot \prod_i \pi_i^{k_i} \right)$ s.t. $\sum_i l_i k_i = w - j$.

4 Composability by Reduction

The second prerequisite of our composition algorithm is that all paths of the input tree schedules are composable. The problem of composing tree schedules by pruning uncomposable paths containing reducible transitions, while retaining the maximum number of paths in either tree schedules, is however notoriously hard. We denote this problem as $\mathrm{MAXCOM}(\Omega_1, \Omega_2)$ and prove that it is at least NP-hard by a reduction from the constrained maximal vertex biclique problem.

Theorem 1 *Given two tree schedules Ω_1 and Ω_2, $\mathrm{MAXCOM}(\Omega_1, \Omega_2)$ is at least NP-hard.*

Proof From the tree schedules Ω_1 and Ω_2, we can generate all paths of the schedule, including those which contain reducible transitions, by performing a depth-first traversal of the tree schedule. Let us denote the generated paths of Ω_1 by $p_1, \ldots, p_r$ where $r = |V_1^F|$. These paths can be partitioned into sets $L_1, L_2, \ldots, L_n$ where $\forall p_j, p_l \in L_i, p_j \equiv p_l$, where $\equiv$ indicates equivalent paths. Similarly, the paths of Ω_2, $q_1, \ldots, q_s, s = |V_2^F|$, can be partitioned into sets $R_1, R_2, \ldots, R_m$ each containing equivalent paths. Now consider a bipartite graph $G = (L, R, E)$ where $L = \bigcup_i L_i$ and $R = \bigcup_i R_i$ and the $E = \{(p, q) | p \in L, q \in R, p \approx q\}$, i.e., there is a edge in G if paths p of Ω_1 and q of Ω_2 are composable. This graph can generated in linear time as checking whether two paths are time compatible, transition compatible, and composable can be done in a single pass from the root to the leaf location of the paths. We will show that the problem $\mathrm{MAXCOM}(\Omega_1, \Omega_2)$ is then equivalent to finding bipartite clique of G having maximum number of nodes such that the solution must consist of at least one member from each of $L_1, L_2, \ldots, L_n$ and $R_1, R_2, \ldots, R_m$. The bipartite clique condition can be seen by the following argument. Assume to the contrary that the solution is not a bipartite clique. This means that there exists a node $p \in L$ and $q \in R$ such that there is no edge between them. However, this implies that $p \not\equiv q$, which is a contradiction and the equivalence between the problems follows. MAXCOM is therefore, at least as hard as solving the constrained bipartite clique problem which has been shown to be at least NP-hard [13]. Hence, MAXCOM is at least NP-hard. □

Feige et al [13] have shown that the constrained biclique problem cannot be approximated within a factor of n^ϵ for some $\epsilon > 0$ unless $P = NP$. So, it is not possible to get a good approximation of the MAXCOM also. Therefore, we

give a heuristic algorithm that relies on pruning schedules with higher utilization.

The Algorithm. The underlying idea of the algorithm is that it checks all paths of one tree against all paths of the other tree and identifies locations that cause collisions (see Line 5 of Algorithm 2). Inside the if clause, the algorithm tries to prune the path that causes the collision by reducing a transition along the path that leads to it. Here, it reuses the PRUNE() procedure from Algorithm 1. Note that if the parameter $\mathcal{R}$ in the procedure PRUNE() is empty, then PRUNE() considers only reducible transitions. If PRUNE() fails, then it is impossible to make two tree schedules compatible by reduction. Since the algorithm prunes equivalent schedules, we do not incur any overhead.

Algorithm 2 Composability by reduction.

Require: $\Omega_1 \sim_K \Omega_2$
Ensure: $\Omega_1 \approx \Omega_2$
procedure REDUC(Ω_1, Ω_2)
for all $p \in P_1$ **do**
for all $i \in p : v_i \rightarrow v_{i+1} \rightarrow \cdots \rightarrow v_{i+n}$ **do**
for all $p' \in P_2$ **do**
5: **if** $sl(v_i).1 \neq \epsilon \wedge sl(v'_i).1 \neq \epsilon$ **then**
// Collision occurred, resolve.
if ((only v prune-able) $\vee$
(UTIL(p)>UTIL(p')) **then**
PRUNE($\Omega_1, \emptyset, v_i, dur(v_1^0, v_i)$, false)
else if v' prune-able **then**
10: PRUNE($\Omega_2, \emptyset, v'_i, dur(v_2^0, v'_i)$, false)
else
return *failed*
end if
end if
15: **end for**
end for
end for
end procedure

Example. Consider the two tree schedules in Figure 4 and we now no longer assume that $g_1 = g_2$. Then, the two paths AD and EG are time compatible, but not composable, since D and G communicate a variable. Applying Algorithm 2, detects that only D is prune-able, so it removes it from Ω_1. The resulting tree schedules (c.f., Figure 7) are time compatible (as seen before) and composable.

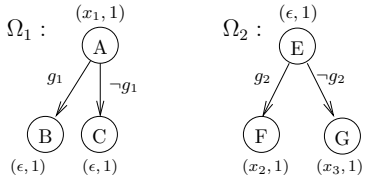


Figure 7. Example applying REDUC().

5 Proportional Fit

If either of the earlier introduced methods fails, then we resort to composing tree schedules using temporal isolation. Temporal isolation is a model in which each application gets only a fraction of the available bandwidth. In this fraction, the application has exclusive access and its network access is temporally isolated from the other applications. For example, one application might get sixty percent of the total available bandwidth while the other gets forty percent of it. Usually, network access is then proportionally distributed to each of the applications to minimize inter network-access latency. So in the example, one application would get three slots and then the other might get two slots. Many embedded networking technologies can realize this model via round-robin or priority-based schemes (see FlexRay [15], TTP [10, 16], Byteflight [8], PowerLink [11]).

Compared to the earlier shown methods, temporal isolation makes any two tree schedules time compatible and composable. However, this comes at an expense, as some of the original properties may no longer be preserved in the transformed schedule. Specifically, temporal isolation preserves the temporal ordering of network access and the structure of the tree schedule; however, it does not preserve the communication delays as they change when the other tree schedule gets its share.

Our method for applying temporal isolation and bandwidth restrictions to tree schedules is called *Proportional Fit* as it involves allocating access based on the proportion of the bandwidth each application gets. The idea is that given a bandwidth restriction for both tree schedules, the algorithm computes the least number of slots that will provide the desired level of bandwidth for each tree schedule. Using this number, the algorithm assigns slots to each tree schedule as proportionally and as evenly as possible. Furthermore, we insert blanks into one tree whenever the other has its exclusive access to the medium.

The Algorithm. Algorithm 3 shows the proportional fit algorithm. The variable p_i specifies the reserved bandwidth for Ω_i. The variable hyp is the hyper period of both schedules. The variable α computes the rate at which blanks should be introduced to Ω_2 to accommodate Ω_1 in the composition. The variable a_i specifies how often Ω_i will be executed within the hyper period. The set Cl_i contains pointers to the locations that the algorithm currently operates on. The variable Cl_2^0 denotes the first element of the set Cl_2. Note, that we assume without loss of generality that $a_1 < a_2$.

The algorithm consists of three parts: in part A, the algorithm makes both tree schedules time compatible as in Algorithm 1. Depending on whether the schedules are isochronous or anisochronous, the algorithm uses an exact

solution for hyp for the former and an approximation for the latter.

In part B, the algorithm computes the step size α and uses it to introduce blanks in one schedule to make room for the other schedule. This is achieved by advancing α steps in Ω_2 while introducing α blanks Ω_1, and then introducing 1 blank in Ω_2 and advancing 1 step in Ω_1. So, both schedules will be composable. At the end of Part B, Ω_2 has had its share of bandwidth and the remaining bandwidth belongs to Ω_1; thus, the algorithm introduces blanks to Ω_2 until hyp.

In Part C, the algorithm the cuts all branches of the tree schedules at the hyper period. This works, because in Part B, the algorithm inserted the exact number of blanks so that each schedule terminates with a former leaf location at the hyper period. The overhead introduced by this algorithm can be computed as described in Section 3 for TMC().

Algorithm 3 Proportional fit given p_1.

Require: $p_1 \leq 1, p_2 = 1 - p_1$
Require: $l_i = \mathrm{dur}(r_i)$ with r_i a run of Ω_i, with $i = 1, 2$
Require: $l'_i = \sum_r \mathrm{dur}(r_i) \cdot p_r(r_i)$ with r_i a path of Ω_i, with $i = 1, 2$
Ensure: $\Omega_1 \approx \Omega_2$
  procedure PROP_FIT(Ω_1, Ω_2, p_1)
  // Part A: unroll schedules
    if Ω_1, Ω_2 isochronous **then**
      $\beta = \frac{LCM(p_2 \cdot l_1, p_1 \cdot l_2)}{p_1 \cdot l_2}$,
      $a_2 = \beta$, $a_1 = \frac{\beta \cdot p_1 \cdot l_2}{l_1 \cdot p_2}$
    else
  // try to minimize $|a_1 \cdot l'_1 + a_2 \cdot l'_2 - \lceil a_1 \cdot l'_1 + a_2 \cdot l'_2 \rceil|$ *in:*
      solve $a_2 \cdot l'_2 \cdot p_1 = a_1 \cdot l'_1 \cdot p_2$ for $a_1, a_2 \in \mathbb{Z}_{>0}, a_1 \leq a_2$
    end if
    $hyp = \lceil a_1 \cdot l'_1 + a_2 \cdot l'_2 \rceil$
    TMC($\Omega_1, \Omega_2, hyp, \mathcal{R}_1, \mathcal{R}_2$) *// $\mathcal{R}_1, \mathcal{R}_2$ are returned*
  // Part B: make room for other schedule
    $\alpha = \lceil \frac{hyp}{a_2 \cdot l'_2} \rceil$
    $Cl_1 = \{v_1^0\}, Cl_2 = \{v_2^0\}$
    while $\mathrm{dur}(r(v_2^0, Cl_2^0)) \leq (a_2 \cdot l'_2)(1 + \alpha)$ slots **do**
      insert α blanks in Ω_1 before all locations $v \in Cl_1$
      take α transitions in Ω_2 (add entries in Cl_2 for branches)
      insert 1 blank in Ω_2 before all locations $v \in Cl_2$
      take 1 transition in Ω_1 (add entries in Cl_1 for branches)
    end while
    insert blanks to Ω_2 after all locations $v \in Cl_2$ until hyp
  // Part C: pruning the trees to hyp
    for all $\{v | v \in V_i \wedge \mathrm{dur}(p(v_i^0, v)) > hyp\}, i = 1, 2$ **do**
      prune subtree rooted at v
    end for
  end procedure

Example. Consider an example where we have two isochronous tree schedules Ω_1 and Ω_2 with $l_1 = 4$ and $l_2 = 3$. Suppose we split the bandwidth with $p_1 = 0.4$ and $p_2 = 0.6$. Using Algorithm 3, we get, $a_2 = 2a_1$ and choose $a_1 = 1$, thus $a_2 = 2$. This results in $hyp = 10, \alpha = 2$. Figure 8 shows the resulting tree schedules Ω'_1 and Ω'_2. These two tree schedules are now compatible and composable.

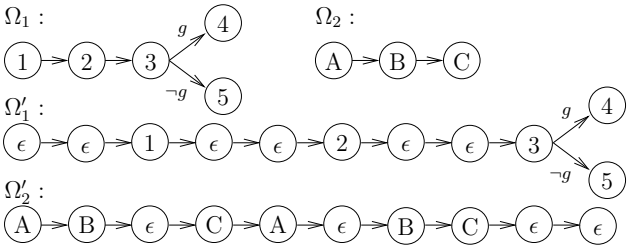


Figure 8. Proportional fit of Ω_1,Ω_2 with $p_1 = 0.4$

6 Case Study

In this case study, we consider two control systems: an inverted pendulum control and a room heater control. The inverted pendulum system [19, 5] is essentially a pole mounted on a cart. The pole is free to rotate round on one axis, and the cart can move horizontally. The objective is to maintain the inverted pendulum in the upright position. In the room heater system, we consider three rooms that share one heater. The goal is to maintain each of the rooms within a desired temperature range. This application is our 'performance' application, as depending on the update frequency, the control model becomes inevitably invalid and then the pendulum will collapse. We measure the time from the start until the collapse, and the longer that time span, the better the system. With this metric, we can measure the impact of the overhead introduced by the composition. The heater application is our 'load' application that puts additional demand on the shared medium and shows the potential of on-the-fly decisions.

6.1 Controller 1: Inverted Pendulum

The inverted pendulum system consists of five sensor units (y_0 to y_4) that communicate the sensor input (pendulum angle θ and angular acceleration ω) to the controller by broadcasting them. Each sensor is expected to report a faulty reading with a probability of 0.01. To mask such faults, the controller system performs a simple majority voting technique to pick the sensor value. As we have five sensors, we can tolerate two independent faults. The control parameter is the linear acceleration of the pivot. This parameter bases on the pendulum's angle to the car and its

angular speed. Initially, the pendulum is at the extreme left position.

We model communication between the controller and the replicated units via the sampling step size in our model. For example, in the standard TDMA system, each unit must report its value. The control unit always waits a full cycle (i.e., five steps), so the length of the TDMA cycle is the five sampling step size. In the implementation, we ignore overhead introduced by clock synchronization or computation time for reading values, adjusting values, and the voting. This can be incorporated as additional overhead to the cycle duration.

Observe that the implemented inverted pendulum system is a discretized version of the hybrid systems control model without feedback. In such a system, there is always a difference between actual and observed values of the control inputs. Because of this discrepancy, an error is introduced in the control mechanism. As this error grows large because of the lack of a feedback mechanism, the model becomes invalid, and the pendulum eventually collapses. We make use of this behavior to demonstrate the utility of tree schedules by showing that a tree schedule minimizes the control error and so stabilizes the pendulum longer than a schedule consisting of sampling of all the sensor values.

Figure 9 shows the tree schedule for this inverted pendulum application. After three sensors (y_0, y_1, y_2) communicated their values, the controller will perform the first majority vote g_1. Based on this result, more sensor values might be omitted (i.e., a reset it performed) or more sensor values are requested. The same happens for the second vote g_2. The standard TDMA schedule of this is to always transmit the longest path [3].

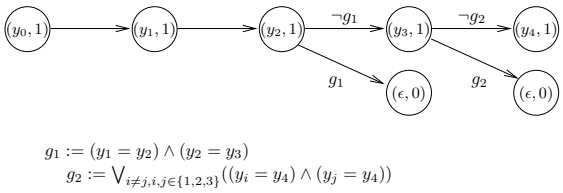


Figure 9. TS for the inverted pendulum system.

6.2 Controller 2: Room Heater

For our second application, we consider the heater benchmark controller as described in [12] with three rooms and one heater where the three rooms communicate their temperature (x_1 to x_3) to the heater. The temperature of a room depends on other rooms, the outside temperature, and on whether the heater is present in the room. The heater is controlled by a typical thermostat, i.e., it is switched on if the temperature is below a certain threshold, and off if it is beyond a higher threshold. When the temperature in any room (x_i) falls below a certain desired level, it may get a heater from the adjacent rooms, provided the temperature in that room is significantly higher. The desired objective is to maintain all the three rooms within the comfortable temperature range. A heater is moved from room j to room i if (1) room i has no heater, (2) $x_i \leq get$ and (3) $x_j - x_i \geq dif$ where get and dif are constants and can differ for each room.

To prevent faulty transitions in our model, we increase the sampling frequency of specific values as we approach the guard conditions [4]. Figure 10 shows the tree schedule for this model. If the state of the controller is such that it is very close to a guard, more values of the guard inputs are transmitted so that a better estimate of the guards can be made. This is incorporated in the schedule by having enabling conditions $x_i \leq \alpha\ get, \alpha > 1$ and $x_j - x_i \in [\beta\ dif, dif], 0 < \beta < 1$ with parameters α and β.

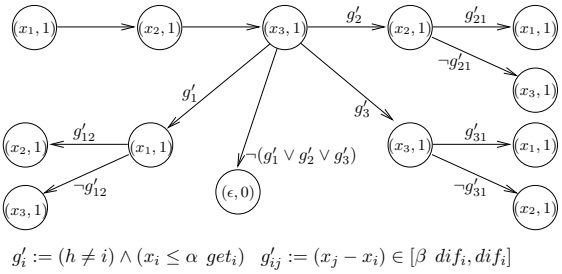


Figure 10. TS for the heater system.

6.3 Composition

As each schedule has a utilization of 100%, clearly we can only compose the two schedules using temporal isolation. Let's assume an average branch length of $l_2' = 3.8$ for the inverted pendulum and $l_1' = 4.8$ for the heater. Furthermore, we decide to divide the bandwidth with 60% for the pendulum and 40% for the heater. Following Algorithm 3, we get $a_1 = 1, a_2 = 2, hyp = 13$. Following the algorithm, this results in eight slots for the pendulum and five slots for the heater. Figure 11 shows the resulting transformed schedules after applying the algorithm. The upper part shows the pendulum schedule, and the lower one shows the heater schedule.

6.4 Simulation & Measurements

We implemented the hybrid systems control using CHARON. CHARON permits modular specification of interacting hybrid systems and supports automatic code generation [2]. Once the model is specified in CHARON, the

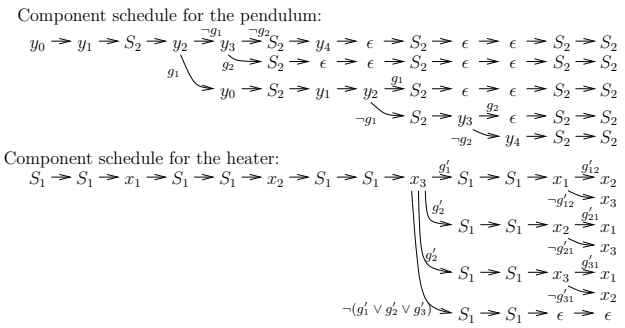


Figure 11. Case study component schedules.

code generator for an agent takes a sampling step size as an input and produces code that approximates the continuous behavior of the model.

In addition to the model, we incorporate a communication channel into the model. The communication channel implements the component tree schedule as shown in Figure 11. During the simulation, the communication channel updates the variables in the control models according to the schedule.

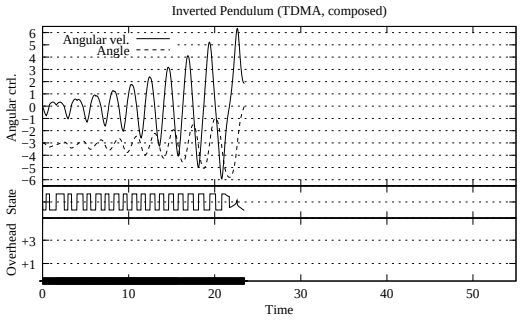
Configuration	Type	Duration	Overhead
TDMA	standalone	51.5	0
Tree schedule	standalone	≈141.2	0
TDMA	composed	23.44	0
Tree schedule	composed	≈54.20	≈ 2.10

Table 1. Measured results of the simulation for the inverted pendulum.

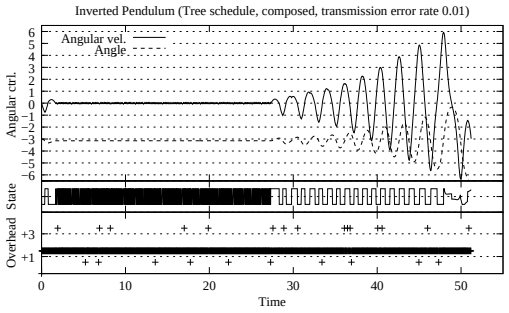
Table 1 shows the measured results of the simulation with a slot size of 0.0035 time units. The configuration column explains what schedule has been used, the type specifies whether the application ran standalone or composed with the heater, the duration column lists how long the model stayed valid, and the overhead specifies the average number of empty slots added in the end of the schedule.

When comparing the first two rows, the tree schedule outperforms the TDMA schedule in standalone. The next two rows show that this advantage is kept during composition, although additional overhead has been added to the tree schedule during the composition operation. Finally, the calculated average overhead (e.g., 2.1032 time units for the inverted pendulum) and the average waiting time (e.g., 5.33 time units for slot x_1 in the inverted pendulum) go in line with the ones observed during the simulation.

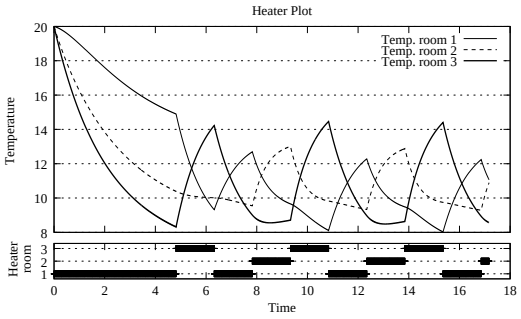
Figure 12 shows the simulation results for the inverted pendulum and the heater. In Figures 12(a) and 12(b), the top part shows the angular control with the angular velocity ω and the angle θ. Right below that part, it shows the



(a) The pendulum using TDMA under composition.



(b) The pendulum using tree schedules under composition.



(c) Overview of the heater model under composition.

Figure 12. Simulation results and plots for the inverted pendulum and the heater.

cart acceleration v. The bottom part shows the overhead due to composition. Figure 12(a) shows the TDMA schedule under composition, which results in no overhead and the model stays valid for about 23 time units. Figure 12(b) shows one path of the tree schedule under composition, where the model stays valid for 51.2 time units. The bottom part shows the overhead which depends on which branch has been taken by the tree schedule. Different branches have different overhead (see Figure 11).

Figure 12(c) provides an overview of the first eighteen time units of the heater. The top part shows the temperature in all three rooms, and the bottom part shows the current location of the heater.

7 Conclusion

Network Code has been proposed as a verifiable, executable model for coordinating and controlling access to a shared communication medium in a distributed real-time system. Previous work in this area concentrated on the conversion of such programs into tree schedules, verification of network properties such as absence of collisions, and providing metrics for their evaluation. In the current work, we have investigated the problem of composing different tree schedules. Specifically, we have extended the definition of tree schedule to include equivalent schedules, defined various notions of compatibility and composability, and proposed algorithms for making them compatible and composable. As a case study, we have considered the composition of two applications: an inverted pendulum system and a room heater system where each application is described by a control model and a slotted transmission schedule. These applications were composed using the algorithm with temporal isolation and the effect of the composition studied by simulating them in CHARON. The results of the simulation show that the tree schedules for the applications can lead to better performance (marked by smaller error) than the standard TDMA schedules when they are run stand alone and composed with another application. The utility of tree schedules is also clearly demonstrated in the room heater application where there was more communication near a guarded transition which resulted in higher switching accuracy. Future work includes applying the tree schedules to hybrid control systems, where they can be used to lower switching errors and facilitate adaptive control. Another is to apply the developed theory to a physical system and see how the simulation results translate to it.

References

[1] R. Alur. Timed automata. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 8–22, London, UK, 1999. Springer-Verlag.

[2] R. Alur, F. Ivancic, J. Kim, I. Lee, and O. Sokolsky. Generating embedded software from hierarchical hybrid models. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 171–182, New York, NY, USA, 2003. ACM Press.

[3] M. Anand, S. Fischmeister, and I. Lee. An analysis framework for network-code programs. In *Proc. of the 6th Annual ACM Conference on Embedded Software (EmSoft'06)*, 2006.

[4] M. Anand, J. Kim, and I. Lee. Code generation from hybrid systems models for distributed embedded systems. In *ISORC*, pages 166–173, 2005.

[5] K. J. Astrom and K. Furuta. Swinging up a pendulum by energy control. *Automatica*, 1999.

[6] S. Baruah. A general model for recurring real-time tasks. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 114–122, 2-4 Dec. 1998.

[7] S. K. Baruah. Feasibility analysis of recurring branching tasks. In *ECRTS*, pages 138–145, 1998.

[8] 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.

[9] A. Easwaran, I. Shin, O. Sokolsky, and I. Lee. Incremental schedulability analysis of hierarchical real-time components. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 272–281, New York, NY, USA, 2006. ACM Press.

[10] S. Eberle, C. Ebner, W. Elmenreich, G. Färber, P. Göhner, W. Haidinger, M. Holzmann, R. Huber, R. Schlatterbeck, H. Kopetz, and A. Stothert. Specification of the TTP/A protocol v2.00. Research Report 61/2001, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2001.

[11] Ethernet Powerlink Standadisation Group (EPSG). *Ethernet Powerlink V2.0 – Communication Profile Specification*, version 0.1.0 edition, 2003.

[12] A. Fehnker and F. Ivancic. Benchmarks for hybrid systems verification. In *HSCC*, pages 326–341, 2004.

[13] U. Feige and S. Kogan. The hardness of approximating hereditary properties. `http://research.microsoft.com/research/theory/feige/homepagefiles/hereditary.pdf`, 2005.

[14] S. Fischmeister, O. Sokolsky, and I. Lee. Network-Code Machine: Programmable Real-Time Communication Schedules. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, 2006.

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

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

[17] I. Shin and I. Lee. Periodic resource model for compositional real-time guarantees. In *Proc. of the 24th IEEE Real-Time Systems Symposium (RTSS)*, 2003.

[18] Y. Takashima. Towards Improving Time and Value Determinism in Distributed Real-Time Systems. Master's thesis, University of Pennsylvania, 2006.

[19] C. J. Tomlin. Hybrid Systems: Modeling, Analysis, and Control, Course in Stanford University. `http://www.stanford.edu/class/aa278a/lecture1.pdf`.

[20] R. Trausmuth and S. Fischmeister. A Soft Processor for Verifiable Real-Time Communication. Technical report, University of Applied Sciences Wiener Neustadt, 2007.