Scheduled protocol programming

Dokter, K.P.C.

Chapter 8

Protocol Scheduling

The work automata developed in Section 7.1 encode all relevant scheduling information of a concurrent application and its protocol. For optimal performance of a concurrent software, the scheduler must take these dependencies into account. However, operating systems schedulers are application-independent and remain oblivious to the dependency information inherent in a protocol, even if such information is available. At best, these schedulers detect consequential effects of a protocol, such as blocking on an I/O-operation or waiting for a lock.

In this chapter, we develop a scheduling framework that extracts all scheduling information from a given work automaton and produces an ideal scheduler for the given software[1]. The most straightforward implementation of this ideal scheduler is to replace the operating system scheduler with our ideal scheduler. However, this approach is not possible if the operating system runs multiple different applications, each of which requiring its own ideal scheduler.

We offer an alternative approach that implements the ideal scheduler without changing the operating system scheduler. The main idea is to exploit the duality between schedules and protocols mentioned in Chapter 1 and transform the ideal schedule into a scheduling protocol. Then, we refine the original protocol by composing it with the scheduling protocol. The refined protocol forces the operating system scheduler to closely follows the ideal schedule, which improves the performance of the concurrent application. The scheduling protocol exploits the fact that the operating system scheduler executes only processes that are not blocked or waiting. Hence, it can enforce a custom schedule by blocking all application processes, except for those that should run according to the application's desired schedule. We block a process by prolonging existing blocking operations like I/O operations or waiting for a lock. Therefore, our approach assumes that the application's desired custom schedule is non-preemptive.

We synthesize non-preemptive schedules for concurrent applications using algorithms for games on graphs (Section 8.1). A graph game is a two-player zero-sum game played by moving a token on a directed graph, wherein each vertex is owned by one of the players. Typically, the ownership of the nodes along a path through the graph alternates between the two players. If the token is at a vertex owned

[1]The work in this chapter is based on [DA21, DJA16]

by a player, this player moves the token along one of the outgoing edges to the next vertex, after which the process repeats. The winning condition determines the player that wins based on the resulting path of the token.

We represent the scheduling problem as a graph game between the scheduler and the application (Section 8.2). The scheduler selects the processes that can execute, and the application resolves possible non-determinism within the processes of the concurrent program. Game-theoretic machinery computes a strategy for the scheduler that optimizes the execution of the concurrent program (according to an objective function that embodies some desired performance measure). Here, we consider only the objective of maximizing throughput, but similar techniques can also optimize for other scheduling performance measures, like fairness, context-switches, or energy consumption.

We view the resulting non-preemptive scheduling strategy itself as a scheduling protocol, and we compose the original protocol with this scheduling protocol to obtain a composite, scheduled protocol (Section 8.3). To evaluate the effect of the restricted protocol on a practical situation, we implement a reference version and a scheduled version of a simple cyclo-static dataflow network. This network consists of four processes, called actors, that interact asynchronously via five buffers (Figure 8.1). A buffer can handle overflows by dropping items to match its capacity. We measure the throughput (i.e., the time between consecutive productions) of both versions of the program. The throughput of the reference version varies significantly, and is on average worse than the throughput of the scheduled version, which, moreover, shows only a small variation.

Finally, we conclude and point to future work in Section 8.5.

8.1 Graph games

Our scheduling framework uses algorithms for games on graphs to construct non-preemptive schedules [DJA16]. A graph game is a two-player game of infinite duration. The possible move sequences are characterized by a safety automaton:

Definition 8.1.1. A safety automaton is a tuple $(Q, \Sigma, \delta, q_0, F)$, with Q a finite set of states, Σ a finite set of moves, $\delta : Q \times \Sigma \to Q$ a transition function, $q_0 \in Q$ an initial state, and $F \subseteq Q$ a set of accepting states, such that for every state $q \in Q$, we have $q \in F$ if and only if $\delta(q, \sigma) \in F$, for some move $\sigma \in \Sigma$.

In other words, a safety automaton is a deterministic finite automaton (DFA), wherein accepting states cannot be reached from non-accepting states, and every accepting state has an accepting successor.

As usual, we extend the transition function δ to finite move sequences by defining $\delta(q, \lambda) = q$ and $\delta(q, s\sigma) = \delta(\delta(q, s), \sigma)$, for every state $q \in Q$, finite move sequence $s \in \Sigma^*$, move $\sigma \in \Sigma$, and empty sequence $\lambda \in \Sigma^*$.

Since we are interested in infinite games only, we define the accepted language of a safety automaton as follows:

Definition 8.1.2. The accepted language $L(A)$ of a safety automaton A is the set of infinite words $\sigma_1\sigma_2 \cdots \in \Sigma^\omega$, such that $\delta(q_0, \sigma_1 \cdots \sigma_n) \in F$, for all $n \geq 0$.

Instantiating Definition 8.1.2 for $n = 0$, we see that $\delta(q_0, \sigma_1 \cdots \sigma_n) = \delta(q_0, \lambda) = q_0 \in F$, which means that the accepted language is non-empty if and only if the initial state q_0 is accepting.

Consider a state $q \in Q$ of a safety automaton. While all moves are possible in q (the transition function is defined for all pairs of states and moves), not every move is enabled in the sense that it leads to an accepted word. In view of Definition 8.1.2, we consider the set $\Sigma_q = \{\sigma \mid \delta(q, \sigma) \in F\}$ of enabled moves at q. By Definition 8.1.1, Σ_q is non-empty, for accepting states $q \in F$.

Based on safety automata, we construct a graph game as follows:

Definition 8.1.3. A graph game is a tuple (A, C, W), with $A = (Q, \Sigma, \delta, q_0, F)$ a safety automaton, $C \subseteq Q$ a set of controlled states, and $W \subseteq \Sigma^\omega$ a winning condition.

A graph game (A, C, W) is played by two players (say Player 1 and Player 2) who take turns to move a token from state to state. We consider Player 1 a protagonist and Player 2 an antagonist. To simplify notation, we write $C_1 = C$ for the states controlled by Player 1, and we write $C_2 = Q \setminus C$ for the states controlled by Player 2. Initially, the token is in state $q_0 \in Q$. If the token is in state $q \in C_k$, with $k \in \{1, 2\}$, then Player k selects an enabled move $\sigma \in \Sigma_q$ and moves the token to state $\delta(q, \sigma)$. As a result, the token moves along a path

$$q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} q_2 \xrightarrow{\sigma_3} \cdots$$

through the safety automaton. Player 1 wins if the sequence $\sigma_1\sigma_2\sigma_3 \cdots \in \Sigma^\omega$ of moves is contained in the winning condition W. Otherwise, Player 2 wins.

A joint strategy (for both players) is a function $\zeta : \Sigma^* \to \Sigma$ that selects a move, for every finite move sequence. A strategy for Player $k \in \{1, 2\}$ is a function $\zeta_k : P_k \to \Sigma$, with $P_k = \{\sigma_1 \cdots \sigma_n \in \Sigma^* \mid \delta(q_0, \sigma_1 \cdots \sigma_n) \in C_k\}$, that selects a move, for every finite move sequence that leads to a state controlled by Player k. If Player k follows a strategy ζ_k, then the resulting move sequence is contained in the set

$$L(\zeta_k) = \{\sigma_1\sigma_2\sigma_3 \cdots \in L(A) \mid \zeta_k(\sigma_1 \cdots \sigma_n) = \sigma_{n+1} \text{ whenever defined}\}$$

of outcomes of the game that are ensured by following strategy s_k. A strategy ζ_1 for Player 1 is winning if $L(\zeta_1) \subseteq W$. That is, strategy ζ_1 ensures that the resulting move sequence is contained in W, irrespective of the moves of Player 2. Winning strategies for Player 2 are defined similarly. A strategy ζ_k for Player k is optimal, if it is winning or if no winning strategy for Player k exists.

Of course, for a given game, it is impossible that both players have a winning strategy. However, for general winning conditions, it is possible that neither player has a winning strategy. In this case, the game is not determined. Nevertheless, Martin proved [Mar75] that graph games are determined if the winning condition is a Borel set[2]. Unfortunately, the results by Martin are descriptive and do not suggest a practical algorithm to find a winning strategy.

For simpler winning conditions, such as the ratio objective, we do have algorithms that compute a winning strategy:

[2]A Borel set is a subset of Σ^ω obtained from languages of safety automata by repeated complements and countable intersections.

Definition 8.1.4. A ratio game is a graph game with a winning condition

$$W_{s/t\geq v} = \left\{ \sigma_1\sigma_2\sigma_3\cdots \in \Sigma^\omega \;\middle|\; \liminf_{n\to\infty} \frac{\sum_{i=1}^{n} s(\sigma_i)}{\sum_{i=1}^{n} t(\sigma_i)} \geq v \text{ and } t(\sigma_1) \neq 0 \right\}$$

for some functions $s : \Sigma \to \mathbb{Z}$ and $t : \Sigma \to \mathbb{N}_0$, and value $v \in \mathbb{Q}$.

To the best of our knowledge, ratio games are introduced by Bloem et al. for the synthesis of robust systems [BGHJ09]. The winning condition $W_{s/t\geq v}$ stipulates that, for every $\epsilon \geq 0$ there exists some time $N \geq 0$, such that for all $n \geq N$ after this time, the fraction $(\sum_{i=1}^{n} s(\sigma_i))/(\sum_{i=1}^{n} t(\sigma_i))$ is at most ϵ less than v.

If $t(\sigma) = 1$, for all moves $\sigma \in \Sigma$, then we obtain a mean-payoff game. Much research has been devoted to finding efficient algorithms that solve mean-payoff games. One of the best known algorithms for mean-payoff games is due to Brim et al. [BCD$^+$11]. Their solution for mean-payoff games generalize easily to ratio games. We provide a brief explanation of this algorithm, and refer to [BCD$^+$11] for full details.

A classical result by Ehrenfeucht and Mycielsky [EM79], called memoryless determinacy, states that there exists a positional optimal joint strategy for mean-payoff games (and ratio games).

Definition 8.1.5. A joint strategy $\zeta : \Sigma^* \to \Sigma$ is positional, if $\delta(q_0, s_1) = \delta(q_0, s_2)$ implies $\zeta(s_1) = \zeta(s_2)$, for all move sequences $s_1, s_2 \in \Sigma^*$.

A positional strategy depends only on the current state $\delta(q_0, s)$, instead of the full history $s \in \Sigma^*$. If both players follow some positional strategy, the outcome of the game is a path

$$q_0 \xrightarrow{\sigma_1} \cdots \to q_k \xrightarrow{\sigma_k} \cdots \to q_n \xrightarrow{\sigma_n} q_k \xrightarrow{\sigma_k} \cdots \tag{8.1}$$

in the safety automaton that ends with a cycle in the distinct states $q_k, \ldots, q_n$. The outcome of a ratio game is winning for a value $v = a/b \in \mathbb{Q}$ if and only if $(\sum_{i=k}^{n} s(\sigma_i))/(\sum_{i=k}^{n} t(\sigma_i)) \geq v = a/b$, which is equivalent to $\sum_{i=k}^{n} w(\sigma_i) \geq 0$, with $w(\sigma) = b \cdot s(\sigma) - a \cdot t(\sigma)$, for all moves $\sigma \in \Sigma$.

Brim et al. observed that positional winning strategies for Player 1 in a ratio game correspond with consistent valuations:

Definition 8.1.6. A valuation $f : Q \to \mathbb{N}_0 \cup \{\infty\}$ is consistent in state $q \in Q$ iff

1. $q \in C_1$ implies $f(q) + w(\sigma) \geq f(\delta(q, \sigma))$, for some move $\sigma \in \Sigma_q$, or
2. $q \in C_2$ implies $f(q) + w(\sigma) \geq f(\delta(q, \sigma))$, for every move $\sigma \in \Sigma_q$,

A valuation is consistent if it is consistent in every state.

Suppose that there exists a consistent valuation $f : Q \to \mathbb{N}_0 \cup \{\infty\}$. Repeated application of Definition 8.1.6 to the path in Equation (8.1) yields $f(q_k) + \sum_{i=1k}^{n} w(\sigma_i) \geq f(\delta(q_k, \sigma_1 \cdots \sigma_n)) = f(q_k)$. If $f(q_k) < \infty$ is finite, then the outcome is winning for Player 1.

Brim et al. suggest a value iteration method to find the smallest possible valuation (we compare valuations pointwise: $f \leq f'$ iff $f(q) \leq f'(q)$, for all states

Algorithm 3: Synthesis problem for ratio games.

Input : A ratio game $(A, C, W_{s/t \geq v})$, with $A = (Q, \Sigma, \delta, q_0, F)$, functions $s : \Sigma \to \mathbb{Z}$ and $t : \Sigma \to \mathbb{N}_0$, and a value $v = \frac{a}{b} \in \mathbb{Q}$.

Output: A the largest quasi-strategy $\zeta : Q \to 2^\Sigma$ that is winning.

1 **foreach** $\sigma \in \Sigma$ **do** $w(\sigma) \leftarrow b \cdot s(\sigma) - a \cdot t(\sigma)$;
2 **foreach** $q \in Q$ **do** $f(q) \leftarrow 0$;
3 **foreach** $q \in C$ **do** $c(q) \leftarrow |\{\sigma \in \Sigma_q \mid f(q) + w(\sigma) < f(\delta(q, \sigma))\}|$;
4 $B \leftarrow \sum_{q \in Q} \max(\{0\} \cup \{-w(q, \sigma) \mid \sigma \in \Sigma_q\})$;
5 $L \leftarrow \{q \in Q \mid f \text{ inconsistent in } q\}$;
6 **while** $L \neq \emptyset \neq \{q \in Q \mid f(q) = 0\}$ *and* $f(q_0) < \infty$ **do**
7 Pick $q \in L$, with $f(q)$ minimal;
8 $f_q \leftarrow f(q)$;
9 $f(q) \leftarrow \min\{n \in \{1, \ldots, B, \infty\} \mid f[q \mapsto n] \text{ consistent in } q\}$;
10 $L \leftarrow L \setminus \{q\}$;
11 **if** $q \in C$ **then** $c(q) \leftarrow |\{\sigma \in \Sigma_q \mid f(q) + w(\sigma) < f(\delta(q, \sigma))\}|$;
12 **foreach** (p, σ) *with* $\delta(p, \sigma) = q \neq p$ *and* $f(p) + w(\sigma) < f(q)$ **do**
13 **if** $p \in C$ **then**
14 **if** $f(p) + w(\sigma) \geq f_q$ **then** $c(p) \leftarrow c(p) - 1$;
15 **if** $c(p) = 0$ **then** $L \leftarrow L \cup \{p\}$;
16 **if** $p \in Q_0$ **then** $L \leftarrow L \cup \{p\}$;
17 **foreach** $q \in Q$ **do**
$\zeta(q) \leftarrow \{\sigma \in \Sigma_q \mid f(q) + w(\sigma) \geq f(\delta(q, \sigma)),$ and $f(q) < \infty\}$;

$q \in Q$). Our Algorithm 3 shows a variation of their algorithm with only a few minor, but novel, adjustments.

The first modification is on Lines 6 and 7. Let $a = \min_{q \in Q} f_*(q)$ be the smallest value of the smallest valuation f_*. If $a < \infty$, then the valuation $f' : Q \to \mathbb{N}_0$ defined as $f'(q) = f_*(q) - a$, for all $q \in Q$, is less than or equal to f_*. Minimality of f_* shows that $a = 0$, which means that there exists a state $q \in Q$ with $f_*(q) = 0$. We refer to $q \in Q$ with $f_*(q) = 0$ as a pivot state. If there are no more pivot states, we can terminate the value iteration.

The second (minor) modification is on Line 9, which becomes apparent for states $q \in Q$ with a negative self loop transition (i.e., some $\sigma \in \Sigma$ with $\delta(q, \sigma) = q$ and $w(\sigma) < 0$). While the original algorithm by Brim et al. repeatedly adds $-w(\sigma)$ to the valuation $f(q)$ at state q, Algorithm 3 immediately jumps to the smallest valuation that resolves the inconsistency at q.

Value problem For given functions s and t, we have a family $\{W_{s/t \geq v} \mid v \in \mathbb{Q}\}$ of winning conditions. Since Player 1 wishes to maximize the ratio between the cumulatives of s and t, it is natural to look for the largest value $v \in \mathbb{Q}$ for which there exists a winning strategy. This problem is known as the value-problem. The set of values that are winning for Player 1 is a half-open interval $(-\infty, v_*]$, with v_* the optimal value. Using Algorithm 3, we can test the query $v \geq v_*$ for any value $v \in \mathbb{Q}$. Hence, the value problem can be solved by a binary search. Comin and Rizzi [CR17] improved this idea by reusing results from earlier queries.

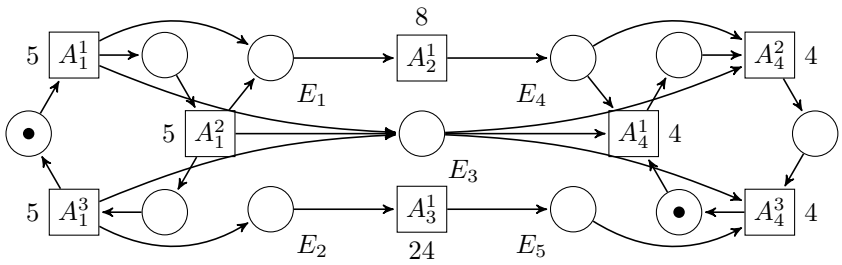


Figure 8.1: Petri net representation of a small cyclo-static dataflow graph.

8.2 Scheduling Game

We use the graph games from Section 8.1 to develop a game-theoretic framework for the synthesis of non-preemptive schedules. We assume that a concurrent program is given in the form of a *work automaton* $(Q, \Sigma, J, T, I, c_0)$ (cf., Section 7.1). Of course, we assume that the work automaton accurately models the real application. It may be nontrivial to verify whether or not the work automaton in fact models the application sufficiently accurately, but this concern is beyond the scope of our work on scheduling. In the worst case, one can ensure the application's compliance with the work automaton model by means of runtime verification.

As a running example, we formalize a simple cyclo-static dataflow network as a work automaton.

Example 8.2.1. Figure 8.1 shows the Petri net representation [Mur89] of a small cyclo-static dataflow (CSDF) graph from [Bam14, p. 29]. Recall that a Petri net consists of places (depicted as circles) that contain zero or more tokens, and transitions (depicted as rectangles). A transition firing consumes a single token from each of its input places and produces a token in each of its output places.

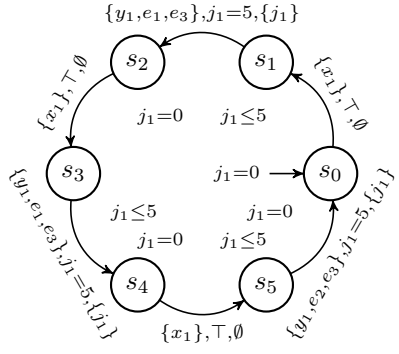
The system in Figure 8.1 consists of 4 actors (A_1 to A_4), which are connected via 5 buffers (E_1 to E_5). The original example in [Bam14] does not make any assumptions on the nature of the buffers: they can be FIFO, LIFO, lossy, or priority buffers. For our purpose, we assume that writing to a full buffer loses the written data. For simplicity, Figure 8.1 represents each buffer as a place in the Petri net. The losing behavior is made precise in the work automaton in Figure 8.2(e).

Each actor A_i, for $1 \leq i \leq 4$, cycles through a number of phases (hence the name cyclo-static). Figure 8.1 represents each phase as a transition A_i^j, for some index j, in the Petri net. Each phase of an actor requires time to execute. Although the actual times may vary, we consider only the worst-case execution times (WCET). We assume that all phases of a given actor have the same WCET. The integer value next to each transition in Figure 8.1 specifies its WCET. ◊

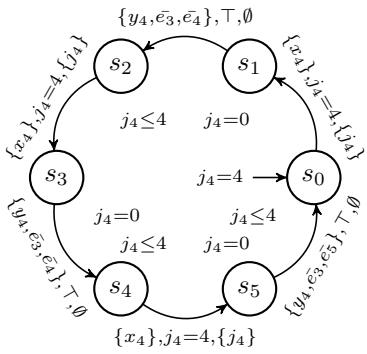
Although the functional behavior (as a Petri net) of the dataflow network in Figure 8.1 is very precise, its non-functional behavior (the timing of transitions) is still unclear. The following example makes this precise by providing a work automaton for each actor and buffer in Figure 8.1:

Example 8.2.2. Figure 8.2 shows the work automata that encode the informal

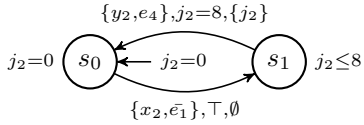
description of the behavior of the CSDF graph in Example 8.2.1. The work automaton for A_i, with $1 \leq i \leq 4$, has a real-valued job variable j_i that measures the progress of its respective actor. The initial condition $j_1 = 0$ in Figure 8.2(a) shows that actor A_1 must first perform 5 units of work on j_1 before it can produce on E_1 and E_3. In contrast, the initial condition $j_4 = 4$ in Figure 8.2(b) shows that actor A_4 can immediately consume tokens from E_3 and E_4. The work automata for A_2 and A_3 in Figures 8.2(c) and 8.2(d) are very similar. Each first consumes a datum from its input buffer, then executes for a given amount of time, and finally produces a datum in its output buffer. Figure 8.2(e) shows the work automaton for a buffer of capacity 4 (buffers of other capacity are defined similarly). The self-loop transition on state s_4 loses the data, if the buffer is full. ◊



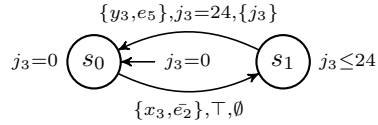
(a) A_1



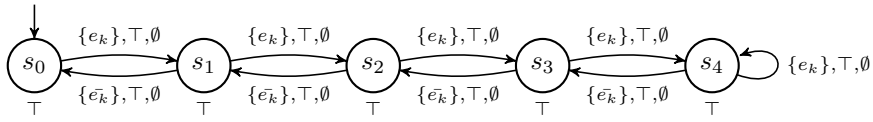
(b) A_4



(c) A_2



(d) A_3



(e) E_k

Figure 8.2: Work automata for the dataflow graph in Figure 8.1, without the idling transitions $(s_i, \emptyset, \top, \emptyset, s_i)$, for all states i. The self-loop transition in (e) loses the data, if the buffer is full.

By definition, non-preemptive scheduling relies on the cooperation of the application for managing its execution. We therefore, assume that the work automaton is cooperative:

Definition 8.2.1. A work automaton A is cooperative if and only if for every job

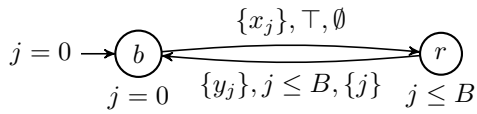


Figure 8.3: Work automaton specifying cooperative behavior of a job j. Signal x_j executes job j, and signal y_j yields job j. The bound $B \in \mathbb{N}_0$ ensures job j eventually yields.

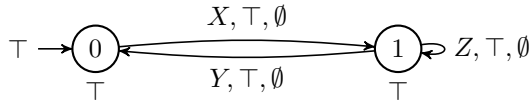


Figure 8.4: Work automaton G_A specifying the rules of the scheduling game for a work automaton A. The scheduler selects any signal $X \subseteq N$ that intersects $X_J = \{x_j \mid j \in J\}$. The application selects a signal $Y \subseteq N \setminus X_J$ that intersects $Y_J = \{y_j \mid j \in J\}$ or a signal $Z \subseteq N \setminus X_J$ that does not intersect Y_J.

$j \in J$ of A, we have that $A \bowtie A_j$ and A are identical up to renaming of states, where A_j is the work automaton in Figure 8.3.

To obtain a scheduling game from a given cooperative work automaton A, we compose A with the auxiliary work automaton G_A in Figure 8.4 that allows us to determine which player is to move and what moves are allowed by each player. In the composition $A \bowtie G_A$, every state is either controlled (if G_A is in state 0) or uncontrolled (if G_A is in state 1).

In the sequel, we assume without loss of generality that the work automaton G_A is already integrated in the specification of the cooperative work automaton:

Definition 8.2.2. A cooperative work automaton is playable iff $A \bowtie G_A$ and A are identical up to state renaming.

The semantics of a playable cooperative work automaton A from Definition 7.1.3 cannot be used directly in a graph game, because there are infinitely many ways to make progress via the d-transitions. Of course, not all progressions are equally likely to happen. In fact, if sufficiently many processors are available, we expect that all running processes make an equal amount of progress. We assume that these processes run uninterruptedly and at equal speeds. That is, we use job assignments $[S] : J \to \mathbb{R}$, for a subset of jobs $S \subseteq J$, such that $[S](j) = 1$ if $j \in X$, and $[S](j) = 0$, otherwise. We denote the expected transition in the semantics with a double arrow:

Definition 8.2.3. The expected semantics $[\![A]\!]_e$ of a work automaton A is the subgraph of $[\![A]\!]$ with the $\Rightarrow$ edges defined by the rules

$$\frac{c \xrightarrow{[S]} c' \qquad \text{if } c \xrightarrow{[S']} c' \text{ and } S' \neq S \text{ then } S \not\subseteq S'}{c \overset{[S]}{\Longrightarrow} c'} \quad \text{and} \quad \frac{c \xrightarrow{\sigma} c'}{c \overset{\sigma}{\Rightarrow} c'}$$

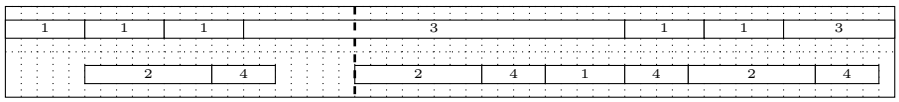


Figure 8.5: Generated schedule. The vertical dashed line indicates the start of the period.

The expected transition relation is just a subrelation of the real transition relation of the semantics of a work automaton. Note that there is no guarantee that the real execution follows the expected one. However, we show in Section 8.3 that deviations of real execution from expected execution do not cause deadlocks.

Lemma 8.2.1. *The expected semantics of a cooperative work automaton is finite.*

Proof. Since A is cooperative, the progress of every job is bounded by $B \in \mathbb{N}_0$ in every state (Figure 8.3). By a simple induction, it follows that the configurations of the expected semantics are contained in the finite set $Q \times \{0, \ldots, B\}^J$. □

The final ingredient for a scheduling game is an objective. The only restriction that we impose on a scheduling objective is that it must be expressible as a ratio objective $W_{s/t \geq v}$, in terms of some functions $s : M \to \mathbb{Z}$ and $t : M \to \mathbb{N}_0$.

Example 8.2.3. The composition of the work automata in Figures 8.2 and 8.4 yields a game graph. We maximize the throughput, which we define as the ratio between the number of productions and the number of time steps (ticks). We can count the productions by counting how often e_5 fires. Hence, we define

$$s(a_1 \cdots a_n) = |\{i \mid e_5 \in a_i \in \Sigma\}| \quad \text{and} \quad t(a_1 \cdots a_n) = |\{i \mid a_i \in \mathbb{R}_+^J\}|.$$

Algorithm 3 finds a subgame for which every play is of optimal throughput.

As the CSDF graph is a deterministic program, all non-determinism is controlled by the scheduler. Since all options ensure optimal throughput, we can resolve the non-determinism arbitrarily. We resolve non-determinism by preferring idling. The deterministic scheduling strategy for this example can be presented as a Gantt chart, shown in Figure 8.5. ♢

8.3 Protocol restriction

Consider a playable cooperative work automaton A from Section 7.1 and a non-preemptive scheduling strategy $\zeta : C \to \Sigma$ from Section 8.2, with C the set of configurations of the expected semantics $[\![A]\!]_e$ of A. Now we intend to implement the schedule ζ as a protocol $A(\zeta)$ that blocks precisely those jobs that are not supposed to make progress, such that the operating system scheduler has to closely follow the desired scheduling strategy.

The solution to ratio games in Algorithm 3 returns a subgraph of the original game graph. For a scheduling game, vertices are configurations of A, and the edge come from the expected transition relation. Hence, the resulting scheduling strategy can be easily transformed back into a work automaton.

Definition 8.3.1. The scheduling work automaton $A(\zeta)$ of a scheduling strategy ζ is defined as the tuple $(C, \Sigma, \emptyset, \to, I, c_0)$ with states $C = Q \times \mathbb{N}_0^J$, trivial invariant $I(c) = \top$ for all states $c \in C$, and transition relation defined by the rule

$$\frac{c \in C \text{ controlled} \qquad c \overset{a}{\Rightarrow} c' \qquad \zeta(c) \text{ defined implies } a = \zeta(c)}{c \xrightarrow{a,\top,\emptyset} c'}$$

We enforce the scheduling strategy by considering the composition $A \bowtie A(\zeta)$. The composition $A \bowtie A(\zeta)$ does not introduce any new, undesired executions, but merely restricts the composition to a subset of desired executions of A. By construction, A and $A(\zeta)$ synchronize only on signals, which agrees with the fact that the schedule ζ is non-preemptive.

For the construction of the scheduling game in Section 8.2, we assume that scheduled jobs run as expected, i.e., they run continuously and at constant speed. However, it is actually very likely that the actual execution deviates from the expected execution. A natural question is whether such deviations may confuse the scheduler enough to introduce deadlocks.

Theorem 8.3.1. *If A is a composition of simple work automata, then $A \bowtie A(\zeta)$ is deadlock free.*

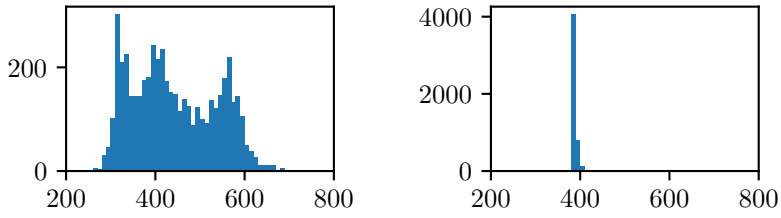
A simple work automaton is a work automaton with at most one job and no silent transitions that in every configuration can either make progress or fire a transition, but cannot do both. The work automata in Figure 8.2 are examples of simple work automata.

of Theorem 8.3.1. The state-space of $A \bowtie A(\zeta)$ is defined by tuples consisting of a state $q \in Q$ and a configuration (q', p') of $A(\zeta)$. Let $c = ((q, (q', p')), p) \in (Q \times (Q \times \mathbb{N}_0^J)) \times \mathbb{R}^J$ be a reachable configuration of $A \bowtie A(\zeta)$. The absence of silent transitions ensures that the scheduler $A(\zeta)$ knows the state of A (i.e, we have $q = q'$). The progress $p : J \to \mathbb{R}$ of the jobs may still be unknown (i.e., $p \neq p'$ is possible). By construction, there exists some expected execution of $A \bowtie A(\zeta)$ that passes through the same state q (although the progress may be different from p) and enables a transition $t = (q, \sigma, g, R, q')$. Since the work automaton for each job is simple, the guard g of transition t states that the progress of a subset of jobs is maximal (while the progress of other jobs is irrelevant). Hence, from c we can make sufficient progress to enable transition t, which implies that c is not a deadlock. □

Example 8.3.1. To evaluate the schedule in Figure 8.5, we implement the CSDF graph in Figure 8.1 in Python. We performed the experiment on a 64-bit Windows 10 Home Edition with a Intel® Core™ i7-7700HQ CPU at 2.80 GHz and 16 GB RAM. We executed the source code with a 64-bit Python 3.9.0 interpreter.

Appendix B shows the source code of the scheduled application. It is obtained manually, but mechanically, from the original source code in Appendix A by adding barrier synchronizations between the actors and a scheduler process. This scheduler process implements the non-preemptive schedule in Figure 8.5.

Figure 8.6 shows the histogram (with a bin-size of 10 ms) of the output of both versions of the program. We measure the throughput (i.e., the time between successive productions) of each version. Both the expected value and the standard



(a) Original (b) Scheduled

Figure 8.6: Throughput of original program (a) and scheduled program (b). The horizontal axis is the time in milliseconds between successive firings of A_4^3 (grouped in bins of 10 milliseconds), and the vertical axis is the frequency of each bin.

deviation of the two versions differ. The original version has an expected throughput of 441 ms with a standard deviation of 95 ms. The scheduled version has an expected throughput of 386 ms and a standard deviation of 6 ms.

The quality of the schedule alone does not explain the improvement of the expected throughput; the characteristics of the original protocol are the most important factor in this example. Recall that we use overflow buffers, which means that an actor loses its datum, if its respective buffer is full. If a datum is lost, all effort invested in its production is also lost. The general-purpose scheduler of the operating system is unaware of these losses. ◊

Although the game-theoretic framework in Section 8.1 aims to optimize the expected throughput, Example 8.3.1 shows that the most significant improvement in the scheduled protocol is in its impact on the standard deviation of the throughput. The time between successive productions is much more predictable for the scheduled version compared to the original version. Since we force the operating system scheduler to closely follow a fixed, deterministic schedule, predictability of the throughput is not surprising.

Predictable timing is a requirement for many systems. For example, a pacemaker must assist a patient's heart to beat at a regular and predictable rate, and a self-driving car must sense and analyze its environment at a predictable rate to avoid collisions. The results of Example 8.3.1 show that the unpredictable behavior of the scheduler of the operating systems can be made tightly predictable by restricting the protocol through composition with a scheduling protocol.

8.4 Related work

There is a wealth of literature on different models for concurrent software, ranging from the well-known Petri nets [Mur89], to the lesser-known higher-dimensional automata [Pra91, vG06].

Our implementation of the scheduler as a composition is very similar to the definition of a scheduler defined by Goubault [Gou95]. In his work, Goubault specifies the application as a higher dimensional automaton and views the scheduler

as a subautomaton. While our scheduling framework builds on graph games, his work depends on the solution of a particular problem on huge sparse matrices.

The work on (variants of) timed automata [AD90] is closely related to our work in Chapters 7 and 8. In fact, syntax of work automata is identical that of timed automata, and the clock variables in timed automata correspond naturally with the jobs in work automata. However, their semantics differs significantly: in a timed automaton all clocks progress at the same speed, while work automata do not make assumptions on the relative speeds of job progress.

Stopwatch automata [CL00] are one step closer to work automata, as in each state a clock is either running or paused. This feature allows stopwatch automata to use the clocks to measure the progress of jobs [AM02].

Uppaal Stratego [DJL+15] is a tool the analysis of stochastic priced timed games. Similar to our work, this tool uses strategies to achieve safety and performance. Uppaal Stratego enforces these strategies via parallel composition with the original automaton. In contrast with our work, Uppaal Stratego can handle stochastic environments by means of simulation and reinforcement learning. Uppaal Stratego's ideas complement our work. Currently, we use the algorithm for ratio games from Section 8.1 to solve the scheduling game in this chapter. However, replacing this algorithm with Uppaal Stratego's algorithms would provide the benefits of both approaches.

8.5 Discussion

Protocols contain valuable information indispensable for construction of optimal (non-preemptive) schedules for allocation of resources to execute a concurrent application. Exogenous languages like Reo express a protocol as an explicit software construct, which makes this scheduling information accessible. We express protocols together with their scheduling information in terms of the work automaton semantics of Reo. We construct a generic scheduling framework based on ratio games to find optimal non-preemptive schedules for an application defined as a work automaton. By composing such a scheduling protocol with the original protocol of an application, we obtain a composite scheduled protocol that forces generic operating system preemptive schedulers to closely follow the desired optimal schedule. The exogenous nature of Reo guarantees that the application code remains oblivious to the substitution of the composite scheduled protocol for its original protocol. An experiment shows that a scheduled version of a cyclo-static dataflow network (with the composite protocol) has higher and more predictable throughput compared to its original version.

Future work The algorithm by Brim et al. to solve ratio games requires the full state-space of an application. In view of the state-space explosion problem, we can use other schemes (like Monte-Carlo tree search) to find good schedules.

Although the work presented in our current chapter focuses on maximizing throughput, the ratio objective can express many other scheduling performance measures, like fairness, context-switches, or energy consumption. We intend to express these performance measures in terms of ratio objectives.