Formal Modeling and Scheduling of Datapaths of Digital Document Printers*

Georgeta Igna[2], Venkatesh Kannan[3], Yang Yang[1], Twan Basten[1], Marc Geilen[1], Frits Vaandrager[2], Marc Voorhoeve[3], Sebastian de Smet[4], and Lou Somers[4]

[1] Fac. of Electrical Engineering
Eindhoven University of Technology, the Netherlands
{Y.Yang,A.A.Basten,M.C.W.Geilen}@tue.nl
[2] Institute for Computing and Information Sciences
Radboud University Nijmegen, the Netherlands
{g.igna,f.vaandrager}@cs.ru.nl
[3] Fac. of Mathematics and Computer Science
Eindhoven University of Technology, the Netherlands
{V.Kannan,M.Voorhoeve}@tue.nl
[4] Océ Research & Development, the Netherlands
{sebastian.desmet,lou.somers}@oce.com

Abstract. We apply three different modeling frameworks — timed automata (Uppaal), colored Petri nets and synchronous data flow — to model a challenging industrial case study that involves an existing state-of-the-art image processing pipeline. Each of the resulting models is used to derive schedules for multiple concurrent jobs in the presence of limited resources (processing units, memory, USB bandwidth,..). The three models and corresponding analysis results are compared.

1 Introduction

The Octopus project is a cooperation between Océ Technologies, the Embedded Systems Institute and several academic research groups in the Netherlands. The aim of Octopus is to define new techniques, tools and methods for the design of electromechanical systems like printers, which react in an adaptive way to changes during usage. One of the topics studied is the design of the datapath of printers/copiers. The datapath encompasses the complete trajectory of the image data from source (for example the network) to target (the imaging unit). Runtime changes in the environment (such as the observed image quality) may require the use of different algorithms in the datapath, deadlines for completion of computations may change, new jobs may suddenly arrive, and resource availability may change. To realize this type of behavior in a predictable way is a major challenge. In this paper, we report on the first phase of the project in which we studied a slightly simplified version of an existing state-of-the-art image processing pipeline that has been implemented in hardware, in particular the scheduling of multiple concurrent data flows.

* Research carried out in the context of the Octopus project, with partial support of the Netherlands Ministry of Economic Affairs under the Senter TS program.

1.1 The Case Study

Océ systems perform a variety of image processing functions on digital documents in addition to scanning, copying and printing. Apart from local use for scanning and copying, users can also remotely use the system for image processing and printing. A generic architecture of the system studied in this paper is shown in Fig. 1.

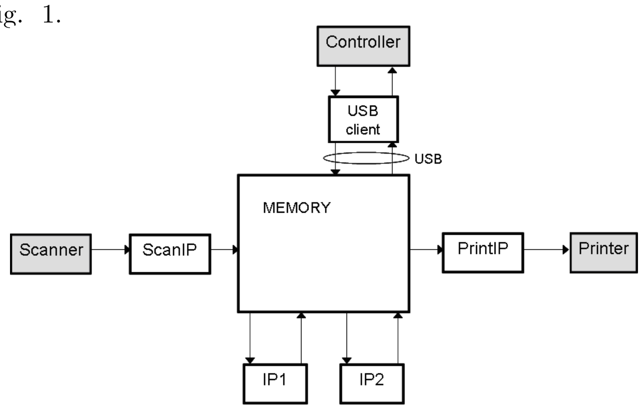


Fig. 1: Architecture of Océ system.

The system has two ports for input: Scanner and Controller. Users locally come to the system to submit jobs at the Scanner and remote jobs enter the system via the Controller. These jobs use the image processing (IP) components (ScanIP, IP1, IP2, PrintIP), and system resources such as memory and a USB bus for executing the jobs. Finally, there are two places where the jobs leave the system: Printer and Controller.

The IP components can be used in different combinations depending on how a document is requested to be processed by the user. Hence this gives rise to different use cases of the system, that is, each job may use the system in a different way. The list of components used by a job defines the *datapath* for that job. Some examples of datapaths are:

– **DirectCopy**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient, PrintIP[5]
– **ScanToStore**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ USBClient
– **ScanToEmail**: Scanner $\rightsquigarrow$ ScanIP $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
– **ProcessFromStore**: USBClient $\rightsquigarrow$ IP1 $\rightsquigarrow$ IP2 $\rightsquigarrow$ USBClient
– **SimplePrint**: USBClient $\rightsquigarrow$ PrintIP
– **PrintWithProcessing**: USBClient $\rightsquigarrow$ IP2 $\rightsquigarrow$ PrintIP

In the *DirectCopy* datapath, a job is processed in order by the components Scanner, ScanIP, IP1, and IP2, and then simultaneously sent to the Controller via the USBClient and to the printer through PrintIP. In the case of the *ProcessFromStore* datapath, a remote job is sent by the Controller to the USBClient for processing by IP1 and IP2, after which the result is returned to the remote

[5] If A $\rightsquigarrow$ B occurs in a datapath, then the start of the processing by A should precede the start of the processing by B.

user via the USBClient and the Controller. The interpretation of the remaining datapaths is similar.

It is not mandatory that the components in the datapath process the job sequentially: the design of the system allows for a certain degree of parallelism. Scanner and ScanIP, for instance, may process a page in parallel. This is because ScanIP works fully streaming and has the same throughput as the Scanner. However, due to the characteristics of the different components, some additional constraints are imposed. Due to the nature of the image processing function that IP2 performs, IP2 can start processing a page only after IP1 has completed processing it. The dependency between ScanIP and IP1 is different. IP1 works streaming and has a higher throughput than ScanIP. Hence IP1 may start processing the page while ScanIP is processing it, with a certain delay due to the higher throughput of IP1.

In addition to the image processing components, two other system resources that may be scarce are memory and USB bandwidth. Execution of a job is only allowed if the entire memory required for completion of the job is available (and allocated) before its execution commences. Each component requires a certain amount of memory for its task and this can be released once computation has finished and no other component needs the information. Availability of memory is a critical factor in determining the throughput and efficiency of the system. Another critical resource is the USB. This bus has limited bandwidth and serves as a bridge between the USBClient and the memory. The bus may be used both for uploading and for downloading data. At most one job may upload data at any point in time, and similarly at most one job may download data. Uploading and downloading may take place concurrently. If only one job is using the bus, transmission takes place at a rate of *high* MByte/s. If two processes use the bus then transmission takes place at a slightly lower rate of *low* MByte/s[6]. This is referred to as the *dynamic* USB behavior. The *static* USB behaviour is the one in which the transmission rate is always *high* MByte/s.

The main challenge that we addressed in this case study was to compute efficient schedules that minimize the execution time for jobs and realize a good throughput. A related problem was to determine the amount of memory and USB bandwidth required, so that these resources would not become bottlenecks in the performance of the system.

1.2 Modelling and Analysis Approaches

We have applied and compared three different modeling methods: Timed Automata (TA), Colored Petri Nets (CPN), and Synchronous Data Flow (SDF). These methods are known to serve the purpose of investigating throughput and schedulability issues. The objective of our research was to see whether the three methods can handle this industrial case study, and to compare the quality, ease of construction, analysis efficiency, and predictive power of the models.

[6] Approximately, *low* is 75% of *high*. The reason why it is not 50% is that the USB protocol also sends acknowledgment messages, and the acknowledgment for upward data can be combined with downward data, and vice versa.

Timed Automata A number of mature model checking tools, in particular Uppaal [4], are by now available and have been applied to the quantitative analysis of numerous industrial case studies [3]. In particular, timed automata technology has been applied successfully to optimal planning and scheduling problems [7,1], and performance analysis of distributed real-time systems [8,12]. A timed automaton is a finite automaton extended with clock variables, which are continuous variables with rate 1. A model consists of a network of timed automata. Each automaton has a set of nodes called *locations* connected by *edges*. A new location is reached after a condition, called *guard*, is satisfied or a synchronization with another automaton takes place via a *channel*. Another way to communicate in the network is by using shared variables.
Petri Nets are used for modeling concurrent systems. They allow to both explore the state space and to simulate the behavior of the models created. We have used CPN Tools [11,10] as the sofware tool for the present case study and for performance analysis using simulation. Petri Nets are graphs with two types of nodes: *places* that are circular, and *transitions* that are rectangular. Directed arcs are used to connect places to transitions and vice versa. Objects or resources are modelled by *tokens*, which are distributed across the places representing a state of the system. The occurrence of events corresponds to firing a transition, consuming tokens from its *input* places and producing tokens at its *output* places. CPN (Colored Petri nets) is an extension where tokens have a value (color) and a time stamp. A third extension is hierarchy, with subnets depicted as transitions in nets higher in the hierarchy.
Synchronous Data Flow Graphs (SDFG) are widely used to model concurrent streaming applications on parallel hardware. An SDFG is a directed graph in which nodes are referred to as actors and edges are referred to as channels. Actors model individual tasks in an application and channels model communicated data or other dependencies between actors. When an actor fires, it consumes a fixed number of tokens (data samples) from all of its input channels (the consumption rates) and produces a fixed number of tokens on all of its output channels (the production rates). For the purpose of timing analysis, each actor in an SDFG is also annotated with a fixed (worst-case) execution time. A timed SDF specification of an application can be analyzed efficiently for many performance metrics, such as maximum throughput [6], latency or minimum buffer sizes. Analysis tools, like the freely available SDF3 [13], allow users to formally analyze the performance of those applications.

Outline of the Paper In Sect. 2, the three models are explained. In Sect. 3, the analysis results are presented and compared. Sect. 4 gives conclusions and some directions for future research.

2 Modelling Approaches

2.1 Timed Automata

In the timed automata approach, each use case and each resource is described by an automaton, except for memory which is simply modelled as a shared variable.

All image processing components follow the same behavioral pattern, displayed in Fig. 2. Initially a component is in idle mode. As soon as the component is claimed by a job, it enters the running mode. A variable *execution_time* specifies how long the automaton stays in this mode. After this period has elapsed, the automaton jumps to the recovery mode, and stays there for *recover_time* time units. The template of Fig. 2 is parametrized by channel names *start_resource* and *end_resource*, which mark the start and termination of a component, and integer variables *execution_time* and *recover_time*, which describe the timing behavior.

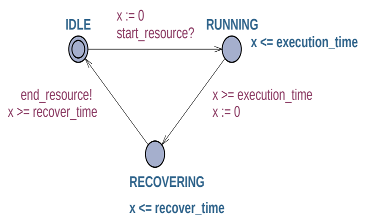


Fig. 2: Component template.

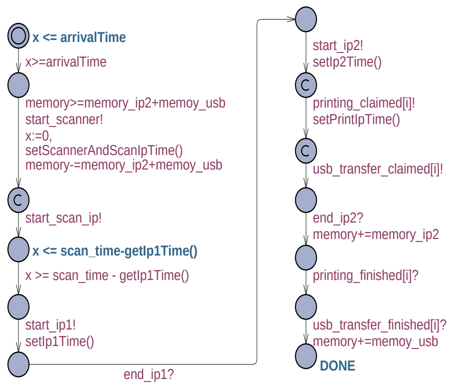


Fig. 3: Automaton for the *DirectCopy* use case.

Each use case has been modeled using a separate automaton. As an example, the automaton for the *DirectCopy* use case is depicted in Fig. 3. A job may only claim the first component from its datapath after its arrival and when enough memory is available for all the processing in the datapath. This figure shows the way memory allocation and release is modelled. At the moment a component is claimed, the use case automaton specifies its execution time. The figure illustrates, also, the way we modelled the parallel activities done by IP2, USBClient and PrintIP.

USB A challenging aspect in modelling the datapath was the USB because of its dynamic behaviour. Firstly we modelled this like a linear hybrid automaton as can be seen in Fig. 4. Linear hybrid automata [2], are a slight extension of timed automata in which besides clocks, also other continuous variables are allowed with rates that may depend on the location. In the automaton of Fig. 4, there are two continuous variables: *up* and *down*, modeling the amount of data that needs to be uploaded and downloaded, respectively. In the initial state the bus is idle (derivatives $up\ and\ \dot{down}\ are\ equal\ to$ 0) and there are no data to be transmitted ($up = down = 0$). When uploading starts (event *start_up?*), variable *up* is set to U, the number of MBytes to be transmitted, and derivative up is set to $-high$. Uploading ends (*end_up!*) when there are no more data to be transmitted, that is, *up* has reached value 0. If during uploading via the USB, a download transfer is started, the automaton jumps to a new location in which

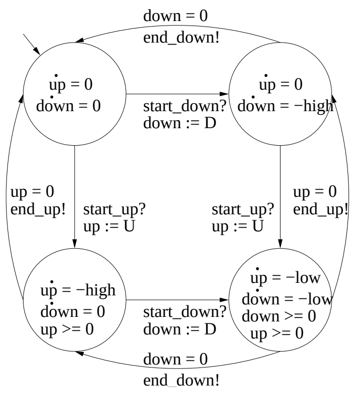


Fig. 4: Linear hybrid automaton model of USB bus.

$down$ is set to D and both $\dot{up}$ and $\dot{down}$ are set to $-low$. The problem we face is that this type of hybrid behavior cannot be modeled directly in UPPAAL. There are dedicated model checkers for linear hybrid automata, such as HyTech [9], but the modeling languages supported by these tools are rather basic and the verification engine is not sufficiently powerful to synthesize schedules for our case study.

We experimented with several timed automaton models that approximate the hybrid model. In the simplest approximation, we postulate that the data rate is high, independently of the number of users. This behavior can simply be modelled using two instances of the resource template of Fig. 2. Our second "dynamic" model, displayed in Fig. 5, overapproximates the computation times

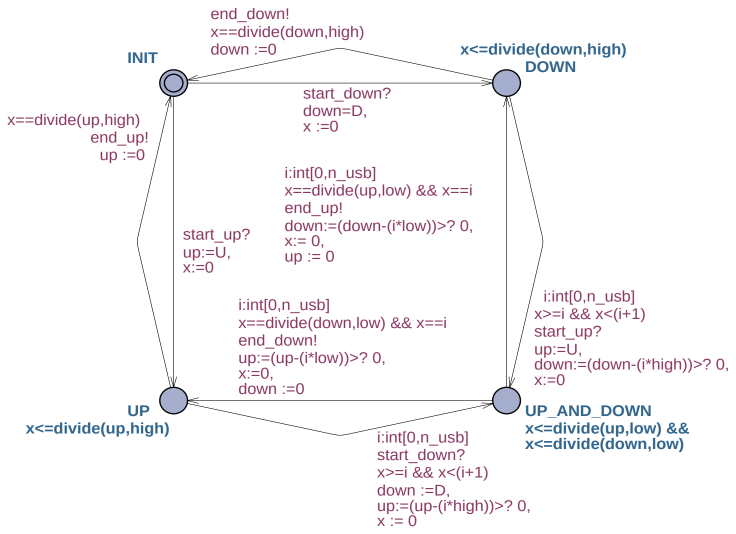


Fig. 5: Second timed automaton model of USB bus.

of the hybrid automaton with arbitrary precision. Clock x records the time since the start of the last transmission. Integer variables up and $down$ give the number of MBytes still to be transmitted. If an upward transmission starts in the initial state, up is set to U and x to 0. Without concurrent downward traffic, transmission will end at time $divide(up, high)$[7]. Now suppose that downward transmission starts somewhere in the middle of upward transmission, when clock x has value t. At this point still $up - high \cdot t$ MByte needs to be transmitted. In UPPAAL we cannot refer to the value of clocks in assignments to integer variables. However, and this is an interesting new trick, using the select statement[8] we may infer the largest integer i satisfying $i \leq t$. We update up to the maximum of $up - high \cdot i$ and 0, which is just a small overapproximation of the amount of data still to be transmitted, and reset x. The other transitions are specified in a similar style.

The UPPAAL verification engine is able to compute the fastest schedule for completing all jobs (without any a priori assumption about the scheduler such as first come first served). However, for more than 6 jobs, the computation times increase sharply due to state space explosion. The state explosion problem can be alleviated by declaring (some of) the *start_resource* channels to be urgent. In this way we impose a "non lazy" scheduling strategy in which a resource is claimed as soon as it has become available and some job needs it. This strategy reduces UPPAAL computation times from hours to minutes, with a risk of losing sometimes the optimal schedule.

2.2 CPN

In the Octopus project, the Petri Net approach takes an architecture oriented perspective to model the Océ system. The model, in addition to the system characteristics, includes the scheduling rules (First Come First Served is used when jobs enter the system) and is used to study the performance of the system through simulation. Each component in the system is modeled as a subnet. Since the processing time for all the components, except the USB, can be calculated before they start processing a job, the subnet for these components looks like the one shown in Fig. 6. The transitions *start* and *end* model the beginning

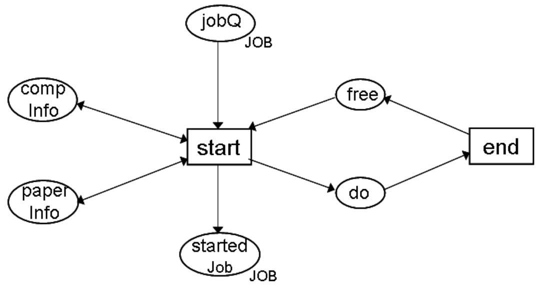


Fig. 6: Hierarchical subnet for components Scanner, ScanIP, IP1, IP2 and PrintIP.

and completion of processing a job, while the places *free* and *do* reflect the

[7] Since in timed automata we may only impose integer bounds on clock variables, we use a function $divide(a, b)$, which gives the smallest integer greater or equal than $\frac{a}{b}$.

[8] Adding a select statement $i : int[0, n\_usb]$ to a transition effectively amounts to having a different instance of the transition for each integer i in the interval $[0, n\_usb]$

occupancy of the component. In addition, there are two places that characterise the subnet to each component: *compInfo* and *paperInfo*. The place *compInfo* contains a token with information about the component, namely the component ID, processing speed and the recovery time required by the component before starting the next job. The place *paperInfo* contains information on the number of bytes the particular component processes for a specific paper size. The values of the tokens at places *compInfo* and *paperInfo* remain constant after initialisation and govern the behaviour of the component. Since the behaviour of the USB is different from the other components, its model is different from the other components; it is discussed below.

In Fig. 6, the place *jobQ* contains tokens for the jobs that are available for the components to process at any instance of time. The color of a token of type *Job* contains information about the job ID, the use case and paper size of the job. Hence, the component can calculate the time required to process this job from the information available in the *Job* token, and the tokens at the places *compInfo* and *paperInfo*. Once the processing is completed, the transition *end* places a token at the place *free* after a certain delay, governed by the recovery time specific to each component, thus determining when the component can begin processing the next available job.

Fig. 7 shows an abstract view of the model. New jobs for the system can be created using the *Job Generator* subnet, which are placed as input to the *Scheduler* subnet at the place *newJob*. The *Scheduler* subnet models the scheduling rules, memory management rules and routes each job from one component to the next based on the datapath of the job. In this model, the scheduling rules are modeled as being global to system and not local to any of the components.

To start with, the *Scheduler* picks a new job that enters the system from the place *newJob* and estimates the amount of total memory required for executing this job. If enough memory is available, the memory is allocated (the memory resource is modelled as an integer token in the place *memory*) and the job is scheduled for the first component in the datapath of this job by placing a token of type *Job* in the place *jobQ*, which will be consumed by the corresponding component for processing. When a component starts processing a job, it immediately places a token in the *startedJob* place indicating this event. The *Scheduler* consumes this token to schedule the job to the next component in its datapath, adding a delay that depends on the component that just started, the next component in the datapath and the dependency explained in Sect. 1.1. Thus the logic in the *Scheduler* includes scheduling new jobs entering the system (from place *newJob*) and routing the existing jobs through the components according to the corresponding datapaths. As mentioned above, the *Scheduler* subnet also handles the memory management. This includes memory allocation and release for jobs that are executed.

USB The USB model is different from that of the other components since the time required to transmit a job (upstream or downstream) is not constant and is influenced by other jobs that may be transmitted at the same time. The Petri

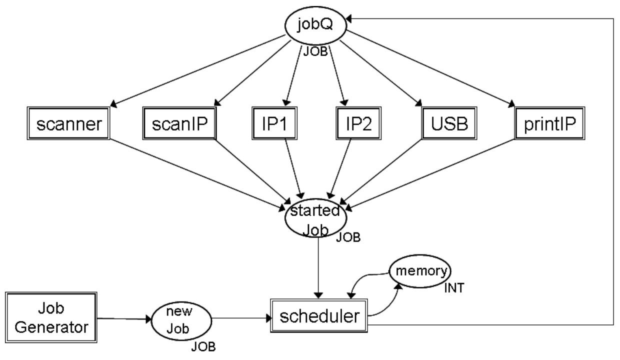


Fig. 7: Architectural view of the CPN model.

Nets approach models the real-time behaviour of the USB explained in Sect. 1.1.

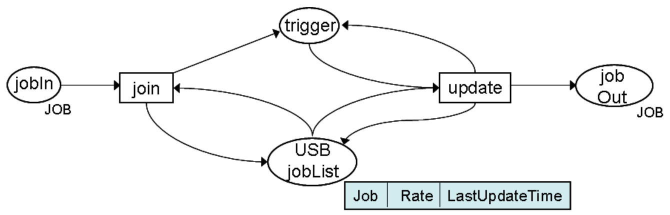
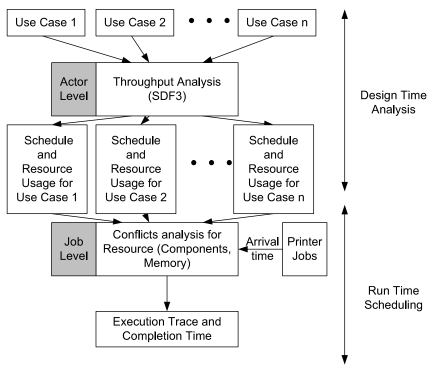


Fig. 8: CPN model for the USB.

The CPN model of the USB works by monitoring two events observable in the USB: (1) a new job joining the transmission, and (2) completion of transmission of a job. Both events influence the transmission rates for any other jobs on the USB, and hence determine the transmission times for the jobs. In the model shown in Fig. 8, there are two transitions *join* and *update*, and two places *trigger* and *USBjobList*. The place *USBjobList* contains the list of jobs that are currently being transmitted over the USB. Apart from information about each job, it also contains the transmission rate currently assigned, the number of bytes remaining to be transmitted and the last time of update for each job. The transition *join* adds a new job waiting at the place *jobIn* that requests use of the USB (if it can be accommodated) to *USBjobList*, and places a token at the place *trigger*. This enables the transition *update* that checks the list of jobs at the place *USBjobList* and reassigns the transmission rates for all the jobs according to the number of jobs transmitted over the USB. The *update* transition also recalculates the number of bytes remaining to be transmitted for each job since the last update time, estimates the job that will finish next and places a timed token at *trigger*, so that the transition *update* can remove the jobs whose transmissions have completed. The jobs whose transmission over the USB is complete are placed in the place *jobOut*. Thus the transition *join* catches the event of new jobs joining the USB and the transition *update* catches the event of jobs leaving the USB, which are critical in determining the transmission time for a single job.

2.3 Synchronous Dataflow Graphs

In the SDF approach, we choose to model the Océ system from an application oriented perspective. In contrast to the two earlier approaches, we take a compositional approach that targets analysis efficiency for application at runtime. Since SDF is particularly well suited to optimize throughput for streaming applications, we focus on the scheduling problem for job sequences consisting of jobs with many iterations per use case (i.e., 100 pages of **DirectCopy**). The scheduling problem is tackled via a 2-phase methodology (see Fig. 9). Each use case is modeled as an SDF graph. Architecture information is included by annotating graph actors with resource usage information. In the **design time analysis** phase, we apply SDF3 [13] to generate a throughput-optimal schedule per use case. In the **runtime scheduling** phase, the schedule is computed based on arrival times of jobs. The schedule takes into account constraints (number of available components, memory amount) of the system. This 2-phase scheduling approach provides schedules and guaranteed job completion times for arbitrary job sequences. It avoids the complexity of analyzing all the details of a job sequence at runtime, which is infeasible in general, sacrificing some performance that might be obtainable via global optimization. The method can be seen as an instance of the Task Concurrency Management method of [14], providing predictability by the use of SDF as a modeling formalism.



Use Case 1
Use Case 2
Use Case n
Actor Level
Throughput Analysis (SDF3)
Schedule and Resource Usage for Use Case 1
Schedule and Resource Usage for Use Case 2
Schedule and Resource Usage for Use Case n
Job Level
Conflicts analysis for Resource (Components, Memory)
Arrival time
Printer Jobs
Execution Trace and Completion Time
Design Time Analysis
Run Time Scheduling

Fig. 9: Job scheduling using SDF models.

Use Case Modeling In the SDF approach, computations performed by the components of printer are modeled as actors. Actors are annotated with execution times and resource usage information. In order to model the appropriate delays between two concurrent computations running on different components of the printer, such delays are also explicitly modeled by means of actors. The USB communication is split into two actors: USB_Download and USB_Upload. The execution times of the USB actors are approximated conservatively by always assuming low bandwidth availability. Thus, use case analysis can be decoupled from job scheduling, sacrificing some accuracy in exchange for analysis efficiency. Fig. 10 shows the SDFG of **ProcessFromStore**, actor production and consumption rates are all 1 in this simple use case.

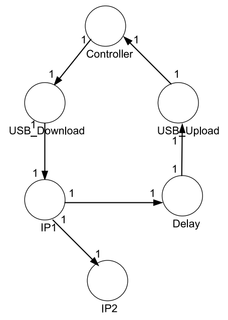


Fig. 10: ProcessFromStore.

Job Scheduling and Completion Time Analysis Using the SDF model to analyze the performance of a single job is straightforward. By ensuring composability through virtualization of the resources (every job gets its own, private share of the resources in system), multiple jobs can also be analyzed efficiently. However, as the components and memory can be shared between jobs, the existing techniques to analyze multiple jobs cannot be applied directly to the datapath analysis of a printer. How to model the behavior of concurrent jobs on the datapath of the printer in a non-virtualized way and how to calculate the completion time of those jobs is the challenge faced.

To analyze the datapath without virtualization, we make some assumptions. We conservatively assume that the resources needed by actors are claimed at the start of a firing and released at the end of firing, where the claim and release of a resource like the memory may happen in different actors. A waiting job can start if all the resources needed according to use case analysis can be reserved. The reservation of resources ensures that the execution time of all job tasks are fixed. As already mentioned, USB bandwidth is always assumed to be low. These assumptions make the system efficiently analyzable by limiting the dynamism in the datapath behavior, and allow the 2-phase scheduling approach explained above. The first phase is concerned with the actor level and uses throughput-optimal self-timed execution (data-driven, every actor fires as soon as it is enabled) as a scheduling strategy for a single SDF that represents a printer use case. Resource usage of each use case is calculated using this self-timed schedule. The second phase concerns job scheduling. Jobs are served in an FCFS (First Come First Served) way. If the resources required by a new job cannot be ensured at arrival time, the new job has to be postponed until the resources are available. Jobs can still be pipelined, overlapping in time, as illustrated below. Two types of resources, mutual exclusive and cumulative resources, are considered in a bit more detail. IP components are an example of mutual exclusive resources that can only be used by one job at a time, while shared memory is an example of a cumulative resource that can be used by many jobs as long as the total usage does not exceed the available amount.

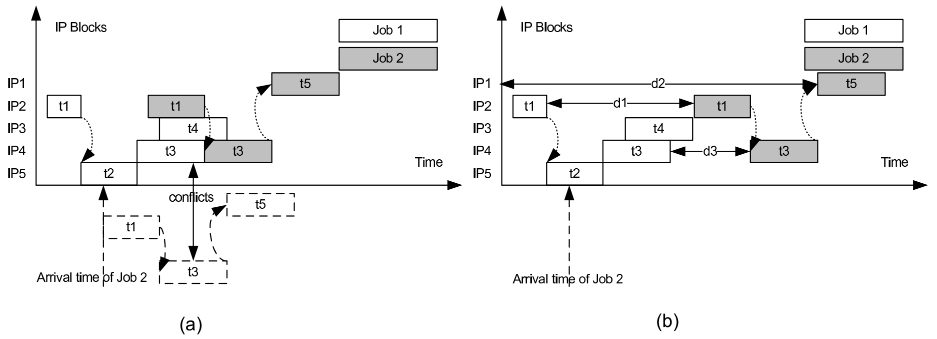


Fig. 11: Scheduling jobs with a conflict.

Fig. 11 shows the scheduling of two jobs (J1 and J2) of different use cases with a component conflict (ignoring for now memory usage). The self-timed schedule and its resource usage are computed in phase one using the throughput analysis algorithm in SDF3. The work is done off-line, avoiding computation work at runtime. In phase two, the scheduler computes the earliest start time of the second job based on the phase one results and runtime information. From Fig. 11(a), we see that J2 cannot start when it arrives at the printer due to a resource conflict on IP4. In order to maximize the resource usage, J2 has to start at a point that ensures that IP4 can be used by J2 as soon as it is released by J1. Fig.11(b) illustrates how to compute this specific point. As explained, when J1 starts its execution, all components and memory it needs are reserved. A system resource usage table is kept to store the release time of components. When J2 arrives, we initially assume it starts at the end of the last actor of J1 (t4 of J1 in Fig.11(b)). Then, we calculate the time distance between the current release time of components and the reservation of those components for J2. The start point then equals the end time of J1 minus the minimum of the computed distances (d3 in Fig.11(b)).

In order to analyze memory conflicts, we store the memory usage of each use case as a list of time interval-memory quantity pairs. Fig.12 shows how to update system memory usage when a new job starts (q_i represents the amount of memory needed at time t_i).As a single job always fits in the memory, we only need to consider the overlap between a new job starting and any running jobs using memory.

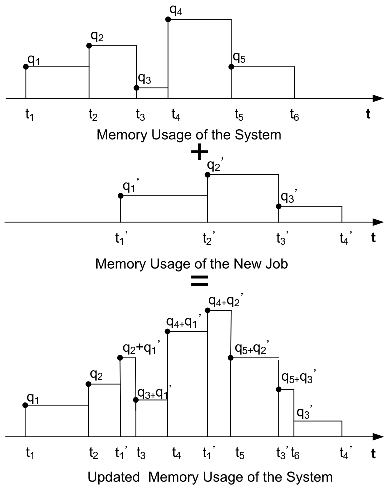


Fig. 12: Update memory usage.

We define a memory usage interval $MI_i = (q_i, [t_i, t_{i+1}))$ to represent that from t_i to t_{i+1} the used memory equals q_i. We can check those intersecting intervals iteratively to verify the memory constraint and calculate the time the job may need to be postponed. Assume that the memory constraint is q_c, and $MI_1 = (q_1, [t_1, t_3))$ belongs to the system and $MI_2 = (q_2, [t_2, t_4))$ to the new job. If $q_1 + q_2 < q_c$, we can check the next pair of overlapping intervals; else, we have to postpone the new job $t_3 - t_2$ and recheck all overlapping intervals. We can repeat these checks until the memory constraint is satisfied.

Observe that other exclusive or cumulative resources, like the USB connection, can be treated in the same way as outlined here.

3 Comparison

This section presents the results from comparing the analysis results for the three approaches. For this purpose, we have chosen a common arrival sequence of 7 one-page jobs shown below[9]:

[9] Due to space limitations we only present one benchmark here. In fact, we studied several other benchmarks, for which we obtained similar results. The benchmark presented in this paper is the most challenging one that we studied thus far.

JobID	Use case	Arrival time	Memory required
a6	PrintWithProcessing	0	12Y
a7	ProcessFromStore	0	24Y
a2	Scan2Email	1X	48Y
a3	Scan2Store	1X	36Y
a5	PrintWithProcessing	1X	12Y
a1	ProcessFromStore	2X	24Y
a4	ProcessFromStore	3X	24Y

The remainder of this section explains the results for both static and dynamic USB behaviour obtained via the three approaches.

Static USB behavior

CPN approach Fig. 13 shows the execution of the jobs by the components with a total completion time of 24X. Even though jobs a6 and a7 arrive at the same time and request *USBdown* simultaneously, job a6 is chosen non-deterministically to use *USBdown* first at time 0. Such resource contentions can be observed for the IP components as well where job a5 waits for the *PrintIP* to become available as it is processing job a6, even though job a5 can be processed by *PrintIP* as soon as the *USBdown* is completed at 6X. The execution of the jobs is also influenced by the memory available in the system. Even though job a2 arrives before jobs a3, a5, a1 and a4, its execution commences only at time 15X, as shown in Fig. 13, because until 15X the memory available is less than that is required for job a2 of use case Scan2Email. This can be observed from Fig. 13 and Fig. 16, where at time 15X execution of job a7 is completed, the memory required for job a2 is available and it is scheduled for processing by the scanner.

SDF approach In the SDF approach, jobs are served in an FCFS way and are scheduled based on the strategy described before. The order of jobs is determined by the arrival time of each job. If jobs arrive at the same time, the order is determined non-deterministically. For the given arrival sequence, there are 12 possible orders for the 7 jobs ($2! \cdot 3! \cdot 1 \cdot 1 = 12$) and the best schedule has the shortest completion time, 27X, under 96Y of memory (see Fig. 14). From the figure, we can see that job a2 is postponed due to the memory limit. Since the static USB model optimistically assumes that USB bandwidth is high, in this experiment, we align with this assumption.

TA approach UPPAAL computed an optimal schedule with completion time 22X, displayed in Fig. 15. The memory chart of Fig. 16 illustrates the use of the available memory in this schedule. At 10X memory is released when a2 is completed, and immediately afterwards, two jobs (a3 and a5) are started. The execution chart shows that IP2 is the critical resource, which is used optimally.

We conclude that UPPAAL managed to come up with the optimal schedule of 22X, CPN found a schedule of 24X, and SDF came up with a schedule of 27X. For the simulation based approach followed by the CPNTool, of course the result depends on the simulation time, and longer simulations lead to better schedules. The SDF approach follows a strict FIFO scheduling and hence the total completion time for the jobs is higher than for the optimal schedule. However, it is the

only approach that is compositional, and hence it is expected to scale better to larger job sequences.
Dynamic USB behavior
CPN approach As shown in Fig. 17, the total execution time is 25.5X as against 24X for the static USB behavior. Analysing the simulation results of the static and dynamic USB behavior, the difference in completion time is caused by the change in transmission rates of the USB.
TA approach The result for the dynamic USB model is depicted in Fig. 18. The total completion time is 25X. It is easy to see that the only difference between this result and the one for the static model is caused by the changes in transmission rate when an upload and a download transmission occurs simultaneously. We claim that this is the optimal schedule for this dynamic behavior.

4 Conclusions and Future Work

We have applied three prominent state based modelling frameworks —Uppaal, Colored Petri Nets, and Synchronous Data Flow— to a realistic industrial case study, and managed to compute schedules for a representative benchmark. Our preliminary conclusion is that Colored Petri Nets provide the most expressive modeling framework, whereas Uppaal currently appears to be the most powerful tool for finding (optimal) schedules. However, this case study pushes Uppaal to its limits and since the SDF approach, which is the only compositional one, is more scalable it is certainly possible that it will outperform Uppaal on larger benchmarks. Such benchmarks can possibly also be tackled using Uppaal Cora [5], a variant of Uppaal that has been constructed to solve scheduling problems. We consider it too early for a definite comparison of modeling frameworks.

We have not embarked on the enterprise to formally relate the three different models. However, we can confirm the result of [8] that the construction of models of the same system using different tools helps to find bugs in the models, and thus contributes to improving the quality of the models.

From a modelling perspective, a very interesting feature in the Océ case study is definitely the USB bus. We consider it surprising that timed automata are able to deal so well with what at first sight appears to be a hybrid phenomenon. The select statement from Uppaal is crucial in defining this model.

Future work includes the construction of more refined models of the same system. We want to develop better sense for what is the right level of modelling. Notable features that we want to model in more detail are the memory bus and memory fragmentation. We also want to study more realistic requirements on system performance such as larger number of use cases (involving for instance batches of several hundred pages), priorities between use cases, hard constraints on throughput, and runtime decision making whether new jobs can be accepted and how they should be scheduled.

References

1. Y. Abdeddaïm, E. Asarin, and O. Maler. Scheduling with timed automata. *TCS*, 354(2):272–300, 2006.

2. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J.Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *TCS*, 138:3–34, 1995.
3. G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal. In *Formal Methods for the Design of Real-Time Systems*, LNCS 3185, pages 200–236. Springer, 2004.
4. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. Uppaal 4.0. In *QEST 2006*, pages 125–126. IEEE CS Press, 2006.
5. G. Behrmann, K. G. Larsen, and J. I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.
6. A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi. Throughput analysis of synchronous data flow graphs. In *ACSD 2006*, pages 25–34. IEEE CS Press, June 2006.
7. M. Hendriks, N. J. M. van den Nieuwelaar, and F. W. Vaandrager. Model checker aided design of a controller for a wafer scanner. *STTT*, 8(6):633–647, 2006.
8. M. Hendriks and M. Verhoef. Timed automata based analysis of embedded system architectures. In *WPDRS 2006*. IEEE, 2006.
9. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A Model Checker for Hybrid Systems. *STTT*, 1:110–122, 1997.
10. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1992.
11. K. Jensen, L. M. Kristensen, and L. Wells. Coloured Petri nets and CPN tools for modelling and validation of concurrent systems. In *STTT*, 9(3-4), June 2007.
12. S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *EMSOFT*, pages 193–202, New York, NY, USA, 2007.
13. S. Stuijk, M. Geilen, and T. Basten. SDF3: SDF For Free. In *ACSD 2006*, pages 276–278. IEEE CS Press, June 2006.
14. C. Wong, P. Marchal, P. Yang, A. Prayati, F. Catthoor, R. Lauwereins, D. Verkest, and H. D. Man. Task concurrency management methodology to schedule the mpeg4 1m1 player on a highly parallel processor platform. In *CODES*, pages 170–177. ACM, 2001.

A Charts

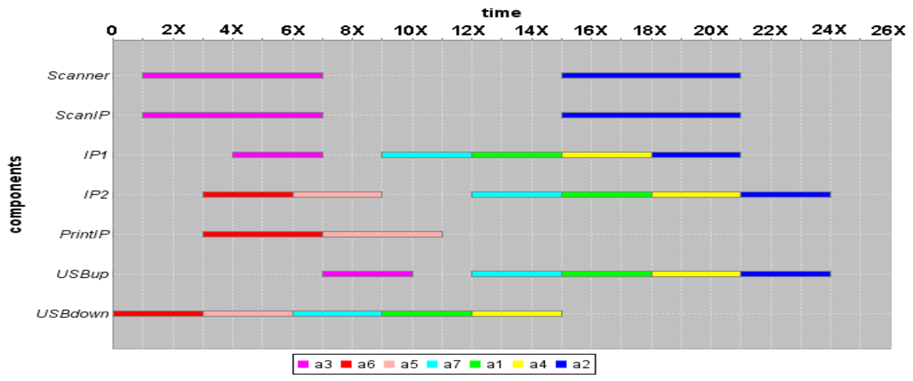


Fig. 13: Execution chart for CPN model with static USB behavior.

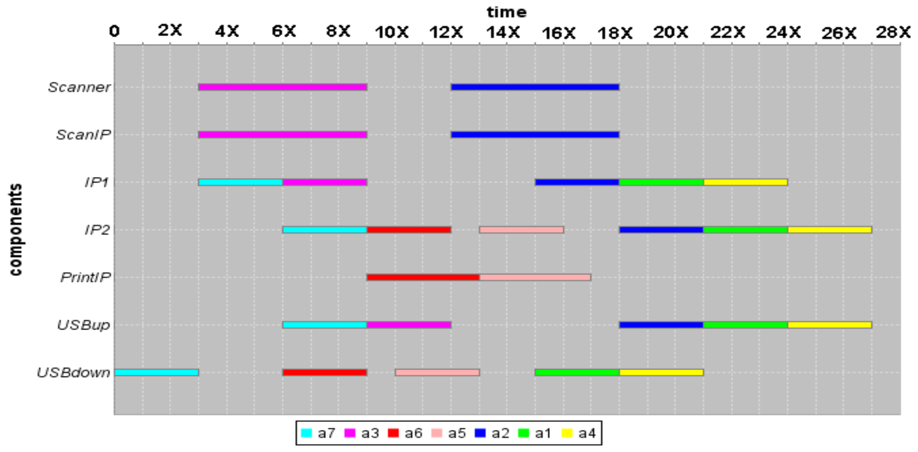


Fig. 14: Execution chart for SDF model with static USB behavior.

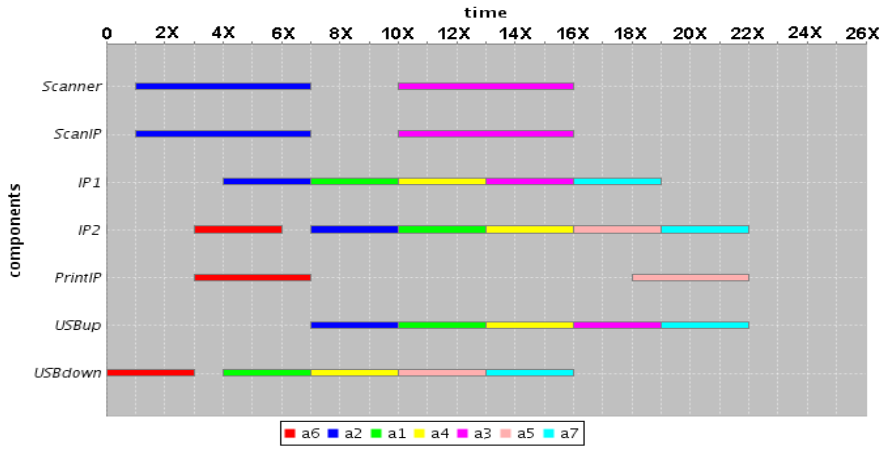


Fig. 15: Execution chart for TA model with static USB behavior.

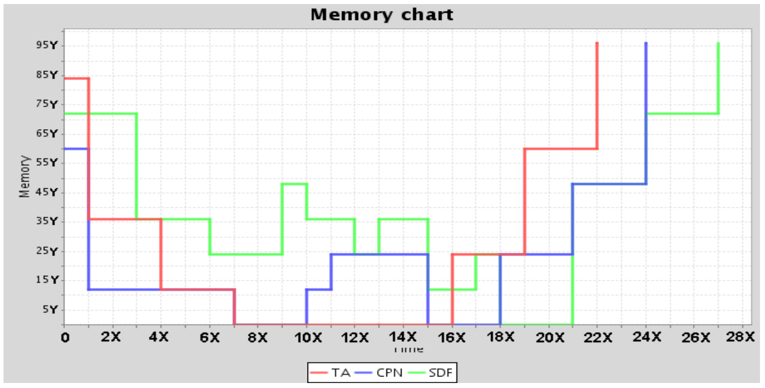


Fig. 16: Memory usage for static USB model.

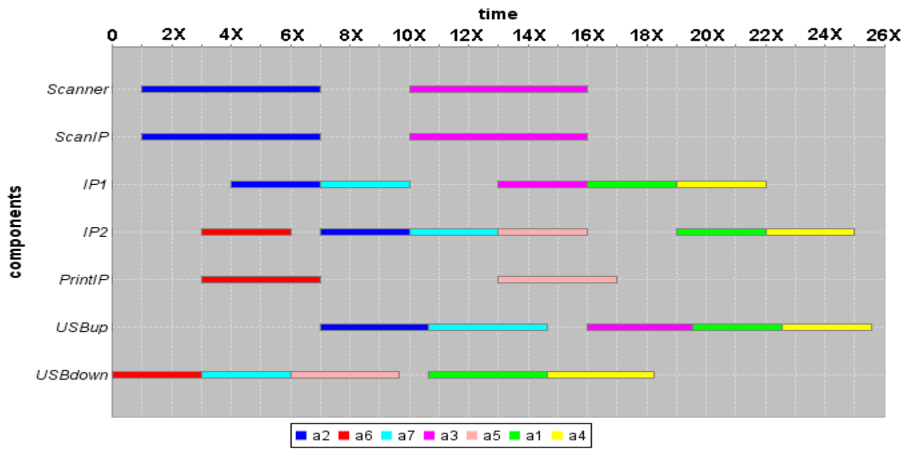


Fig. 17: Execution chart for CPN model with dynamic USB behavior.

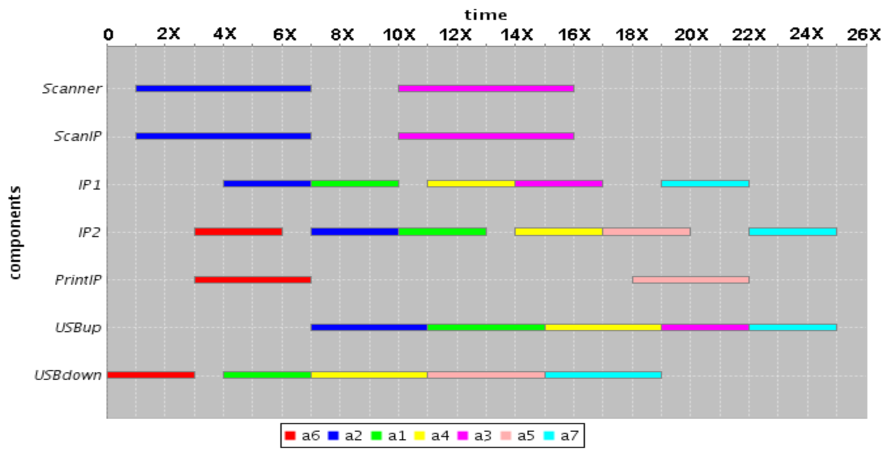


Fig. 18: Execution chart for TA model with dynamic USB behavior.