Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Schedulability Analysis of Timed CSP Models Using the PAT Model Checker

Oğuzcan OĞUZ [1], Jan F. BROENINK and Angelika MADER

Robotics and Mechatronics (formerly CE), Faculty EEMCS, University of Twente

Abstract. Timed CSP can be used to model and analyse real-time and concurrent behaviour of embedded control systems. Practical CSP implementations combine the CSP model of a real-time control system with prioritized scheduling to achieve efficient and orderly use of limited resources. Schedulability analysis of a timed CSP model of a system with respect to a scheduling scheme and a particular execution platform is important to ensure that the system design satisfies its timing requirements.

In this paper, we propose a framework to analyse schedulability of CSP-based designs for non-preemptive fixed-priority multiprocessor scheduling. The framework is based on the PAT model checker and the analysis is done with dense-time model checking on timed CSP models. We also provide a schedulability analysis workflow to construct and analyse, using the proposed framework, a timed CSP model with scheduling from an initial untimed CSP model without scheduling. We demonstrate our schedulability analysis workflow on a case study of control software design for a mobile robot. The proposed approach provides non-pessimistic schedulability results.

Keywords. schedulability analysis, model checking, CSP, timed CSP, PAT, real-time systems.

Introduction

In model-based embedded control system design methodology, two main concerns need to be addressed to achieve reliable designs. The first is that an embedded system embodies a lot of concurrency and interaction that induce additional complexity to the software design. The second is that an embedded system has to interact with the environment in a timely fashion; the embedded system model is subject to real-time constraints in order to be correct. The concurrency and timeliness concerns need to be specified and verified at the model level at early design stages.

The CSP process algebra [1] provides a process-oriented foundation to specify, design and analyse concurrent and interacting systems. To allow for timed reasoning, extensions to the basic theory are available, which provide discrete-time [2] and continuous-time semantics [3–5]. The model checking tools FDR [6] and PAT [7] support automatic verification of timed CSP models. The timed interpretations of CSP assume that there are always sufficient resources for processes to execute. This is called the *maximal parallelism* assumption [5, Chapter 9]. With the maximal parallelism assumption, the implementation details such as resource limitations and the scheduling policies to resolve those limitations are abstracted away.

[1]Corresponding Author: *Oğuzcan OĞUZ, Robotics and Mechatronics, (formerly CE), Faculty EEMCS, University of Twente,*
P.O. Box 217 7500 AE Enschede, The Netherlands. Tel.: +31 53 4892606; Fax: +31 53 4892223; E-mail: o.oguz@ewi.utwente.nl.

However, practical CSP implementations face actual resource limitations. To have real-time support, they need scheduling of processes to ensure orderly and efficient use of resources [8]. For instance, occam 2 [9] has fixed-priority scheduling of processes. Priorities are defined relatively by using asymmetric parallel `PRI PAR` and asymmetric alternative `PRI ALT` constructs. CSP libraries CTJ [10], CT++ [11] and C++CSP2 [12] also have fixed-priority scheduling. These follow the tradition of occam to specify priorities relatively; but these support more levels of priority. With these libraries, it is possible to apply efficient fixed-priority scheduling solutions such as rate-monotonic (RM) scheduling. occam-pi/KRoC [13] and LUNA [14] support absolute and time-varying priorities. With these frameworks, applying earliest-deadline-first (EDF) scheduling is also possible as well as RM scheduling. The Toc [15] programming language directly supports EDF scheduling by fully integrating deadlines in process specifications.

For a CSP model of an embedded control system, schedulability analysis can be used to guarantee timeliness of the system. Performed on the model level, it provides a way to check whether all the time-constrained executions finish before their deadlines. Schedulability of the embedded control system should be analysed with respect to its particular execution platform and the associated scheduling scheme. It is required that there exists reliable execution time measurements for the tasks and that the variations in the execution times are accounted for in the analysis.

In this paper, we propose a model-based schedulability analysis framework for analysing CSP-oriented designs. Our framework is based on Stateful Timed CSP [16, 17], which is an extension of timed CSP [5], and analysis is performed using the PAT [7] model checker.

The framework is composed of task and processor models defined as timed CSP processes in the PAT tool. These framework structures are high-level abstractions that define the scheduling behaviour in the system. They are used to construct a timed CSP model with scheduling behaviour by instrumenting an initial CSP model without scheduling. In the constructed model, the tasks are assigned with fixed absolute priorities and mapped to processors. Processor models model the non-preemptive scheduling of the tasks. Best and worst case execution times for the tasks are also incorporated to account for variations in the execution times of the tasks. Schedulability of the constructed model can be analysed through dense-time model checking using the PAT tool.

We also propose a schedulability analysis workflow describing the successive refinement steps to construct a timed CSP model with scheduling from an initial untimed one without scheduling. The initial model is refined with timing and scheduling information by using the proposed schedulability framework. We show that, throughout the construction process, the safety properties of the initial model are preserved. On the resulting model we perform, as well as the schedulability analysis, verification of the liveness properties and deadlock freedom. We demonstrate our schedulability analysis workflow on a case study of control software design for a two-wheeled mobile robot.

Contributions of this work are as follows:

- A schedulability analysis framework to analyse multiprocessor schedulability of CSP models with non-preemptive fixed-priority tasks with variable execution times.
- A schedulability analysis workflow that describes how to refine, using the proposed framework, an untimed CSP model without scheduling into a timed one with scheduling behaviour. On the resulting model, we perform dense-time model checking to analyse its schedulability and verify liveness properties and deadlock freedom.
- A case study of control software design for a two-wheeled mobile robot where we demonstrate our schedulability analysis workflow.

Related Work

In classical real-time scheduling theory, there are well established traditional schedulability analysis methods to check schedulability guarantee for various preemptive or non-preemptive scheduling schemes with periodic and/or sporadic tasks with fixed or dynamic priorities [18, 19]. However, traditional schedulability analysis methods are seriously challenged in case of multiprocessor systems. Furthermore, the allowed task structures are restricted to periodic or sporadic tasks with simple precedence relationships. These methods cannot handle systems modelled in CSP which involve complex task relationships and arrival patterns.

Motivated by the process-oriented programming language Toc [15], Korsgaard et al. [20] provide a framework for traditional schedulability analysis of malleable jobs of arbitrary parallel structure. The framework models the jobs with a SEQ/PAR-structure that is suitable to the process-oriented formalism, and multiprocessor scheduling is supported. However, structuring of jobs is limited to only SEQ/PAR constructs without involving any synchronization and the jobs are required to be independent. The framework is used to conduct pessimistic but sustainable schedulability tests.

In recent years, real-time model checking emerged as an alternative way of doing schedulability analysis. These methods employ dense-time model checking to verify schedulability of a system. With these methods, multiprocessor schedulability analysis is inherently possible. The task arrival patterns and task synchronizations are less restricted, promoting the applicability of the analysis. Also, the analysis results are less pessimistic and provide higher processor utilization since systems and task arrivals can be modeled in more detail.

The prominent formalism in model-based schedulability analysis is timed automata [21] or the task automata [22], an extension of timed automata with tasks. UPPAAL real-time model checker [23, 24] is mostly employed in the implementations.

UPPAAL-based schedulability analysis frameworks [25,26] and the TIMES tool [27] (only single-processor scheduling) model the scheduling problem with task and resource models that synchronize with each other, and the related additional data structures that store task and resource data. Various preemptive and non-preemptive scheduling algorithms are supported. Complex task arrivals and task relationships can be modelled in UPPAAL. Then the schedulability problem is formulated as a reachability property and real-time model checking is performed. However, schedulability problems with preemptable tasks that have variable execution times require stopwatch automata power on which reachability analysis is undecidable in the general case [28]. For this reason, exact analysis for such problems is not possible; instead over-approximation methods are employed [25, 27].

Regarding our problem, schedulability analysis of systems described in CSP can be done using one of the UPPAAL-based solutions mentioned above. This requires transforming the CSP models into UPPAAL systems in accordance with the adopted UPPAAL-based schedulability framework. Then the schedulability of the system can be verified by reachability checking in UPPAAL. In case of an unschedulability result, a reverse mapping from the witness trace generated by UPPAAL to the CSP model should be provided to locate the cause of unschedulability in the CSP model. However, defining such a transformation and a reverse mapping would be a tedious approach.

Instead, our approach is based on dense-time model checking on Stateful Timed CSP [16, 17] models using the PAT tool [7]. Multiprocessor scheduling is supported and the schedulability scheme is non-preemptive fixed-priority scheduling. We also present an associated design workflow in which the schedulability analysis is embedded in the process-oriented design of the system where a scheduled and timed CSP model is constructed. On this model, besides the schedulability, it is also possible to formally verify other properties of the system, such as liveness properties and deadlock freedom, using the PAT tool.

Several theories of timed process algebra that include structures to define resources and

scheduling behaviour, such as ASCR [29] or PARS [30], have been proposed. In these process algebras, resource constrained processes and scheduling of them are included in the syntax and the semantics. Our approach does not propose a new process algebra nor does it extend the semantics of timed CSP that it is based on. The former approach results in more expressive frameworks and it enables syntactically more natural specification of various resources, resource demanding tasks and scheduling schemes within the system description. On the other hand, our framework can benefit from better and more mature tool support. Furthermore, it suits better to the well-established process-oriented design practices that use the CSP formalism.

1. Preliminaries: Process Analysis Toolkit (PAT)

In this section, we give a brief overview of the syntax and informal semantics of Stateful Timed CSP and of the features of PAT model checker. For details, see Sun et al. [16] and the PAT user manual [7].

PAT's Real-Time System (RTS) module is based on Stateful Timed CSP, which is an extension of Timed CSP [5]. The process model constructs used in our study to define processes in PAT's RTS module are as follows:

$$\begin{aligned} P = \mathit{Stop} \mid \mathit{Skip} \mid e \rightarrow P \mid e \twoheadrightarrow P \mid a\{\mathit{program}\} \rightarrow P \mid \mathit{ch}!\mathit{exp} \rightarrow P \\ \mid \mathit{ch}?x \rightarrow P \mid [b]\,P \mid \mathit{if}\,(b)\,\{P\}\,\mathit{else}\,\{Q\} \mid P \,\Box\, Q \mid P \sqcap Q \mid P \setminus X \\ \mid P\,;\,Q \mid P_1 \,||...||\, P_n \mid P_1 \,|||...|||\, P_n \mid \mathit{atomic}\{P\} \mid P\ \mathit{timeout}[d]\ Q \\ \mid \mathit{Wait}[d] \mid P\ \mathit{interrupt}[d]\ Q \mid P\ \mathit{within}[d] \mid P\ \mathit{deadline}[d] \end{aligned}$$

The process *Stop* idles and does not engage in any events. The process *Skip* successfully terminates, possibly after idling for some time. The process $e \rightarrow P$ is initially prepared to engage in the event e and, after performing the event, it behaves as P. Urgent event prefixing ($e \twoheadrightarrow P$) is used to define urgent events that cannot delay. An event is enabled if it is not blocked by any of the system processes and an urgent event must occur as soon as it is enabled. The process *a{program}* $\rightarrow P$ performs the data operation a executing *program* which may be a simple procedure updating data variables or a complicated sequential program.

ch!exp $\rightarrow P$ and *ch?x* $\rightarrow P$ denote the processes prefixed with the events for writing to and reading from the channel *ch*, respectively. A channel named *ch* of buffer size *n* is declared with the phrase: *"channel ch n;"*. If buffer size is given zero then the channel is synchronous. Channel readings can be guarded with predicates (*ch?[x>0]x* $\rightarrow P$) or with expected values (*ch?1* $\rightarrow P$).

[b]P defines a guarded process; it is blocked until the guard condition *b* is satisfied and then proceeds as *P*. A conditional choice based on the condition *b* is written as *if (b) P else Q*.

An external choice ($P \,\Box\, Q$) between two processes in initially ready to perform the events that either processes can engage in. The choice is resolved by the observation of the first visible event, in favor of the process that performs the event. The process $P \sqcap Q$ with an internal choice describes a non-deterministic choice between P and Q that is resolved internally by the process itself.

The notation $P \setminus X$ is used to encapsulate a set of events in X within a process. All of the events in X are made internal to the process. They are removed from the interface of the process; no other processes may engage in them.

$P\,;\,Q$ denotes the sequential composition of two processes. P starts first, and when it terminates, the control is immediately transferred to Q.

$P_1 ||...|| P_n$ denotes the parallel composition of a number of processes. Any event which appears in the interface of more than one process must involve all of the such processes whenever it occurs. All the involved processes must synchronize on the event by simultaneously engaging in it. In contrast, in the interleaving composition ($P_1 |||...||| P_n$), all processes perform their events completely independent of each other and do not interact on any event except the termination events.

A process *P* can be defined to be atomic (*atomic{P}*) to associate it with higher priority. If an atomic process has an enabled event, the event will have priority over all the events from the non-atomic processes in the system. No other non-atomic event may occur if there is an enabled atomic event. If multiple atomic processes are enabled, then they interleave each other. Furthermore, an enabled atomic event is urgent; it has to happen immediately.

The process *P within[d]* is restricted to react, by engaging an observable event, within *d* time units. The process *P deadline[d]* must terminate within *d* time units. If the time restrictions defined by *within* or *deadline* constructs are not met, then the system deadlocks.

We note that, time restricted actions dictated by urgent event prefixing, atomic processes, *within* or *deadline* can not be blocked and delayed beyond the restriction by the environment. So, it is wise to use such constructs for the actions that the environment is not supposed to engage in.

Regarding the data types, PAT supports global variables of integer, Boolean and integer arrays. It also supports importing and using user defined data types or external static methods, which can be implemented in C#. Calling an externally defined method (a static method or a method of a user defined object) named *f* with parameters p_1,...,p_n is done using the phrase *call(f,* p_1,...,p_n*)*. Method calls and updating and referencing variables can be done in the event prefixes with data operation or in the predicates of conditional statements.

PAT's RTS (Real-Time System) module supports verification of assertions (defined with the keyword *#assert*) of different kinds on the defined processes. The supported assertions include classical refinement/equivalence relationships, Linear Temporal Logic (LTL) properties, reachability properties and deadlock freedom. Based on clock zone abstraction [16], PAT's RTS module can handle dense-time model checking.

2. Schedulability Analysis with Timed CSP Models

In this section we present our schedulability analysis approach. Our approach is based on dense-time model checking on Stateful Timed CSP models. For the rest of the paper we will use the terms "Stateful Timed CSP process" and "timed CSP process" interchangeably.

Our approach consists of a schedulability analysis framework and an associated schedulability analysis workflow. The schedulability analysis framework consists of reusable task and CPU models described as timed CSP processes. These framework structures are used to add scheduling behaviour to a CSP model of a control software.

The schedulability analysis workflow describes how to construct and analyse a timed and scheduled CSP process from an initial untimed and unscheduled CSP process. We call the initial process the "platform-independent process (PIP)" and the process constructed using it, the "platform-specific process (PSP)". The schedulability workflow is depicted in Figure 1. Initially, we assume that there is a PIP of the control system. The PIP is the process that models the untimed, unscheduled platform-independent behaviour of the control system. Ideally, the PIP would be a result of an initial design and analysis phase and it would satisfy all the safety specifications. We explain our assumptions on the PIP in Section 2.1.

As the first step of the workflow, based on the PIP, the PSP of the control system is constructed by incorporating platform-independent timing information, execution platform constraints and hardware mapping and priority information (see the top three parallelograms

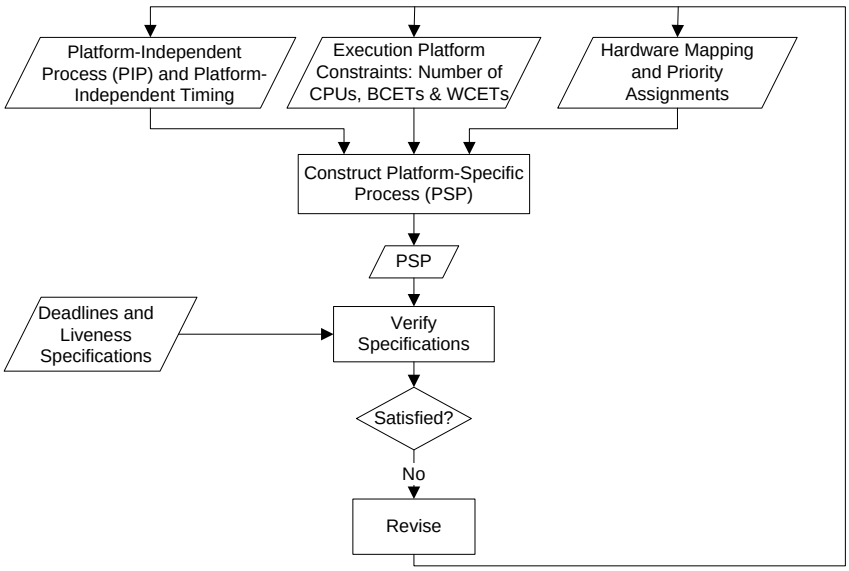

Figure 1. Schedulability analysis workflow.

in Figure 1). The step-by-step description of the construction process is as follows:

1. **Instrument the PIP with platform-independent timing** (Section 2.2)
   The PIP is refined into a timed process by incorporating platform-independent timing such as urgent events, cycle times for periodic processes, minimum inter-arrival times for sporadic events, timeout points, etc.
2. **Specify hardware mapping, priorities and execution times** (Section 2.3)
   In this step the number of processor or processor cores present in the execution platform is specified. For the computational tasks contained in the PIP, the hardware mapping, fixed priority values and best and worst case execution times are specified.
3. **Add task and CPU processes** (Section 2.4)
   The PIP is instrumented with task and CPU processes of the framework that, in combination, define the scheduling behaviour.

The resulting PSP defines the timed and scheduled behaviour of the control software tied to a particular execution platform. At the verification step (Figure 1), we perform dense-time model checking on the PSP to verify schedulability of the control system against the specified deadlines. Deadline violations are checked by introducing timeout points in the PSP with deadline values and performing model checking to see if any timeouts can occur (Section 2.5).

Additionally, at this step, the PSP is also model checked to verify deadlock freedom and liveness properties of the system. Regarding only the events present in the PIP, the PSP is a timewise trace refinement of the PIP; all the safety specifications, which hold for a PIP also hold for the corresponding PSP. This feature suggests that platform-independent untimed safety analysis can be done on the PIP and the results do carry over to any PSP constructed from this PIP. We show this refinement relation in Appendix A.

In case of any negative results at the verification step, the witness traces, if any, should be examined, and the PSP should be revised by modifying the execution platform constraints, priority assignments or hardware mapping.

2.1. Platform-Independent Process (PIP)

In this section we present our assumptions on the structure of a PIP. For the schedulability analysis, we assume there is a PIP that models the untimed, platform-independent behaviour

```
1 //CP0 periodically computes a setpoint for CP1
2 CP0 = cp0_in -> task.0 -> write_setpoint ->
3        CP0;
4
5 //CP1 periodically reads a setpoint and computes a control output
6 CP1 = read_setpoint -> task.1 -> task.2 -> cp1_out ->
7        CP1;
8
9 SYSTEM = CP0 ||| CP1;
```

Listing 1: A sample platform-independent process (PIP) called SYSTEM including 3 task kinds.

of the control system. Even though timeliness is important for a control system, there is no timing yet in the PIP. The PIP only deals with the order of events and it is assumed to satisfy all the safety specifications of the system.

A typical control system is composed of a number of periodic and/or aperiodic control computations. Each of these control computations can involve a number of task executions such as control algorithm computations or processing of sensor data. In this direction, we assume that, in the control system that the PIP represents, there are a number of computational tasks.

In the PIP description we require that each computational task is represented by a single “task event" denoting the end of the task execution. Task events are not meant to be involved in any synchronization; they are actually placeholder events that are to be replaced by the task processes while constructing the PSP. Consequently, we require two kinds of events in the PIP process description. The first is the task event kind, as explained above. The second is the normal event kind. All the events (reading and writing to channels, synchronization events and data operation events, etc.) that are not specified as task events are normal events.

A PSP constructed from a PIP models the dispatching, scheduling and execution of computational tasks in timed CSP. These operations require tasks to be tracked with unique id’s. To ensure that a PSP is feasible to model check, we restrict the set of task events in a PIP to be fixed and finite with each task event given a unique name. We also require that, in a PIP, there is no interleaving or parallel combination of any two processes which include the same task event so as to prevent overlapping dispatches of the same task in a PSP. These restrictions enable us to avoid online generation of unique task id’s, an operation that would prohibitively increase the statespace of a PSP. The designer should ensure that a PIP obeys these restrictions. For complex models it can be hard to check the validity of a PIP manually, however, the validity check can easily be automated.

An example PIP named SYSTEM is shown in Listing 1. It is composed of two processes, CP0 and CP1. CP0 computes and writes a setpoint value which CP1 reads but the communication is not synchronized since we would like the two processes to be executed at different frequencies. Actual transfer of the setpoint values are abstracted away in the model. Writing and reading the setpoint values are denoted by two distinct events, namely the write_setpoint and read_setpoint events. In SYSTEM there are three kinds of computational tasks which are denoted by the placeholder task events task.0, task.1 and task.2.

2.2. Instrumenting the PIP with platform-independent timing

In this section we explain the first step of constructing a PSP from a PIP. In this step, the PIP is refined into a timed process by incorporating platform-independent timing information such as urgent events, cycle times for periodic processes, minimum inter-arrival times for sporadic events, timeout points, etc. In order to specify such time-

```
1 CP0 = ((cp0_in ->> task.0 -> write_setpoint ->> Skip) ||| Wait[20]);
2       CP0;
3
4 CP1 = ((read_setpoint ->> task.1 -> task.2 -> cp1_out ->> Skip) ||| Wait[10]);
5       CP1;
6
7 SYSTEM = CP0 ||| CP1;
```

Listing 2: The process SYSTEM after it is converted into a timed process.

sensitive behaviour, the PIP is instrumented with timed operators of Stateful Timed CSP (`Wait[t]`, `Timeout[t]`, `Interrupt[t]`, `Deadline[t]`, `Within[t]` and the urgent event prefix `->>`).

All the events in the PIP that cannot be delayed are converted to urgent events. Hidden events, which are not of interest for verification, placeholder task events and the other events that can be delayed are left unmodified. The delay process `Wait[t]` can be inserted anywhere in the PIP description to introduce delays. Similarly, the timed operators `Deadline[t]` and `Within[t]` can be inserted anywhere to introduce time restricted behaviour. The untimed event interrupts and external choices can be replaced by timed interrupts (`Interrupt[t]`) and by timeouts (`Timeout[t]`), respectively.

In Listing 2, the running example `SYSTEM` process is shown after it is converted to a timed process. All the events except the task events are converted to urgent events. Also, the processes `CP0` and `CP1` are modified to have periods of 20 and 10 time units respectively. This is done by first adding `Skip` processes in interleaving combination to the parts of the processes, and then replacing the `Skip`'s with `Wait` processes.

After this step the resulting process is a timed version of the PIP that is still platform-independent since there is no scheduling of the tasks and the execution times of the tasks are not incorporated into the model yet.

2.3. Specifying Hardware Mapping, Priorities and Execution Times

In order to perform the schedulability analysis for a particular execution platform, the PIP needs to be supplemented with execution times and priorities of the tasks and their mapping to the CPUs (or the CPU cores) present in the execution platform. In our schedulability framework, tasks are statically mapped to CPUs. To accommodate this, we adopt a task model, which specifies the following attributes for a task:

- T_ID: Unique id of the task.
- BCET: Best case execution time (≥ 1) of the task for the assigned CPU.
- WCET: Worst case execution time ($\geq$ BCET) of the task for the assigned CPU.
- PRIORITY: Priority of the task. Higher the priority value, higher the priority of the task is.
- CPU_ID: Id of the CPU that the task is assigned to.

For each unique task event in the PIP, we need to specify its attributes. However, the tasks grouped into a process representing a component of the system usually share the same priority and might be assigned to the same CPU. So, most of the time, it is more convenient to assign a CPU and a priority value to a process that includes a number of tasks instead of doing that explicitly for each of the tasks. A process that is explicitly assigned to a particular CPU with a priority value is called a “mapped process". Priority of a mapped process is defined according the selected priority assignment scheme. A process that is not explicitly assigned to a CPU inherits the mapping and the priority of its parent process. Depending on the design

```
1  //task_arr[t_id][attr]
2  //attr: 0 -> BCET
3  //      1 -> WCET
4  //      2 -> MP_ID
5  var task_arr[3][3] = [4,6,0, //t_id=0: task.0
6                        1,3,1, //     1: task.1
7                        1,3,1];//     2: task.2
8
9  //mp_arr[mp_id][attr]
10 //attr: 0 -> PRIORITY
11 //      1 -> CPU_ID
12 var mp_arr[2][2] = [1,0, //mp_id=0: CP0
13                       2,0];//      1: CP1
```

Listing 3: Sample Task and Mapped Process Attribute Arrays for `SYSTEM`.

decisions, the selection of the mapped processes can be done in several ways as long as it unambiguously specifies a CPU and a priority value for each of the task events in the PIP.

A mapped process has the following attributes:

- MP_ID: Unique id of the mapped process.
- PRIORITY: Priority of the mapped process. Higher the value, higher the priority of the mapped process is.
- CPU_ID: Id of the CPU that the mapped process is assigned to.

In a PSP, attributes of all the mapped processes and the tasks are stored in two 2D arrays called `task_arr` and `mp_arr`. The arrays are indexed first by the unique id of the task or the mapped process and then by the attribute. In a task attribute array, for each task, we include an indirection to the mapped process that the task belongs to. Using this indirection, the priority and the mapping of a task are inherited from the mapped process containing the task.

In Listing 3, sample attribute arrays for the tasks and mapped processes defined for the PIP example `SYSTEM` are shown. The selected mapped processes are `CP0` and `CP1`, which are assigned to the same CPU with the id 0. `CP1` is assigned with higher priority.

Modifying an existing hardware mapping configuration (existing task and mapped process arrays) for a particular execution platform might require altering not only the mapped process array but also the task array. This is due to that, in the new configuration, the selection of mapped processes might be different. This may require updating the MP_ID fields of the task array. Additionally the BCET and WCET values for the tasks might be different for their new CPU assignments. However, given BCET and WCET values of the tasks for all kinds of CPUs in the execution platform, generation of mapped process and task arrays for a particular selection and mapping of mapped processes can be easily automated with a script.

2.4. Adding task and CPU Processes

So far, we have refined the PIP into a timed process with platform-independent timing. We have specified hardware mapping, priorities and BCET and WCET of the tasks in task and mapped process attribute arrays. However, the modified PIP does not have the scheduling of the tasks yet. The information specified in the task and mapped process attribute arrays is not yet used.

In this section we instrument the PIP process with schedulability framework structures which define the scheduling behaviour. Schedulability framework structures consist of the template processes for tasks and CPUs, related macro definitions and related channel and variable declarations (Listing 4). The template processes for tasks and CPUs are called `TASK`

```
1  channel dispatch_task[_NUM_OF_CPUS] 1;
2  channel finished_task[_NUM_OF_CPUS] 0;
3
4  var<PriorityQueueList> cpu_queues = new PriorityQueueList(_NUM_OF_CPUS);
5  #define push(cpu_id, t_id)
6    cpu_queues.Push(cpu_id, mp_arr[task_arr[t_id][2]][0], t_id);
7  #define get_first(cpu_id) cpu_queues.First(cpu_id);
8  #define get_count(cpu_id) cpu_queues.GetCount(cpu_id);
9  #define remove(cpu_id, t_id)
10   cpu_queues.Remove(cpu_id, mp_arr[task_arr[t_id][2]][0], t_id);
11
12 //Task process
13 #define task_cpu(t_id) mp_arr[task_arr[t_id][2]][1];
14 TASK(t_id) = push_task{call(push, call(task_cpu, t_id), t_id)} ->>
15                finished_task[call(task_cpu, t_id)]?[x==t_id]x ->> Skip;
16
17 //CPU process
18 CPU(cpu_id) = CPU_NEXT(cpu_id) ||| CPU_EXEC(cpu_id);
19
20 var exec_id[_NUM_OF_CPUS] = [-1(_NUM_OF_CPUS)];
21
22 CPU_NEXT(cpu_id) =
23  [call(get_count, cpu_id)>0 && exec_id[cpu_id] != call(get_first, cpu_id)]
24   assign_top{exec_id[cpu_id] = call(get_first, cpu_id)} ->>
25    CPU_NEXT(cpu_id);
26
27 CPU_EXEC(cpu_id) =
28  [exec_id[cpu_id] != -1]execute ->>
29   atomic{Wait[1];
30          dispatch_task[cpu_id]!exec_id[cpu_id] ->>
31           remove_task{call(remove, cpu_id, exec_id[cpu_id])} ->>
32            reset_exec{exec_id[cpu_id] = -1} ->>
33             Skip};
34  dispatch_task[cpu_id]?t_id ->> CPU_DELAY(t_id, cpu_id);
35  CPU_EXEC(cpu_id);
36
37 CPU_DELAY(t_id, cpu_id) =
38  (Wait[task_arr[t_id][0] - 1]; end_task -> Skip) within[task_arr[t_id][1] - 1];
39  finished_task[cpu_id]!t_id ->> Skip;
40
41 CPUS = ||| x:{0..(_NUM_OF_CPUS-1)} @ CPU(x);
```

Listing 4: Description of TASK and CPU processes.

and CPU, respectively. The TASK and CPU processes are boilerplate models that are reused for every CSP model to be analysed.

2.4.1. Task process

A task process is an instance of the template process TASK (Line 14, Listing 4) parametrized by the unique id of the task. When enabled, an instance of TASK signals its release by the urgent event push_task that calls the macro push to add the task into the priority queue of the corresponding CPU. Then TASK waits to synchronize on the finished_cpu[cpu_id] channel with the corresponding CPU process; this communication event signals that the execution of the task is finished.

We use the TASK process to instrument the PIP: *each placeholder task event in the PIP*

```
1  CP0 = ((cp0_in ->> TASK(0); write_setpoint ->> Skip) ||| Wait[20]);
2        CP0;
3
4  CP1 = ((read_setpoint ->> TASK(1); TASK(2); cp1_out ->> Skip) ||| Wait[10]);
5        CP1;
6
7  PSP_SYSTEM = (CP0 ||| CP1) || CPUS;
```

Listing 5: SYSTEM process after instrumented with TASK and CPU processes.

process description is replaced by an instance of the TASK *process.* The unique id of a replaced task event is given as the argument to the replacing TASK instance. In Listing 5, CP0 and CP1 processes of the PIP SYSTEM are shown after replacing all the placeholder task events with TASK process instances.

2.4.2. CPU process

In a PSP, scheduling and execution of the released tasks are modelled by processes that represent CPUs. Each CPU (or CPU core) present in the execution platform is represented by a separate instance of the template process CPU (Line 18, Listing 4). Instances of CPU are given unique ids from 0 to _NUM_OF_CPUS-1 which is the total number of CPUs in the system.

In our framework, the associated scheduling policy is non-preemptive fixed priority scheduling. Each instance of CPU uses an external FIFO priority queue data structure called cpu_queues to sort the tasks according to their priorities. The priority queue data structure is implemented externally in C# and imported in the PAT tool. It has the push, get_first, get_count and remove operations which are defined as macros. cpu_queues and its operations are defined in lines 4-10 of Listing 4.

All the processes, variables, channels and queues used in the definition of the process CPU are parametrized with cpu_id of the corresponding CPU. An instance of CPU fetches tasks from its priority queue and, upon dispatching, delays for a certain time to simulate task executions. CPU is the interleaving combination of two sub-processes, CPU_NEXT and CPU_EXEC. The variable exec_id (Line 20, Listing 4) stores the id of the first task in the priority queue which is going to be executed next. Initially and after dispatching a task, exec_id is reset to -1.

CPU_NEXT (Lines 22 - 25) is responsible for assigning and, whenever necessary, refreshing the value of exec_id by polling the priority queue. It assigns exec_id with the t_id of the first task in the priority queue whenever the queue is not empty and the value of exec_id is different from the t_id of the first task in the queue.

CPU_EXEC (Lines 27 - 35) defines the behaviour for dispatching and executing a task. Whenever the urgent event execute is enabled (when the CPU is idle and there is at least one released task in the queue), a task is dispatched at that time. The execution time of a dispatched task is represented by a delay for one time unit (with the Wait[1] process inside the atomic block), plus, in CPU_DELAY process, a delay for the interval [BCET-1,WCET-1]. Then, CPU_DELAY synchronizes on channel finished_cpu[cpu_id] with the TASK associated with the id of the dispatched task to signal the end of the task execution.

The task dispatch point is the start of the atomic block (Line 29, Listing 4). By the start of the atomic block (before the clock for Wait[1] starts ticking), the Wait[1] process forces all the urgent events in the PSP that might happen, to happen. This guarantees that all the task releases in the PSP that cannot delay happen and exec_id is assigned, in CPU_NEXT, with the first task in the priority queue.

The atomic block in CPU_EXEC has a critical function. Without the atomic block, value

```
1  CP0 = ((d_start.0 ->>
2            cp0_in ->> TASK(0); write_setpoint ->> d_end.0 ->> Skip)
3          ||| Wait[20]);
4        CP0;
5
6  CP1 = ((d_start.1 ->>
7            read_setpoint ->> TASK(1); TASK(2); cp1_out ->> d_end.1 ->> Skip)
8          ||| Wait[10]);
9        CP1;
10
11 PSP_SYSTEM = (CP0 ||| CP1) || CPUS || DEADLINES;
```

Listing 6: PSP_SYSTEM process after two time constrained processes in CP1 and CP1 are marked with d_start.i and d_end.i events.

of exec_id might change due to a later task release before or just after the end of Wait[1] process causing a wrong value (for the t_id of the task to be dispatched) written to the buffered channel dispatch_task. We are abusing here the property that the enabled atomic events have higher priority than all the enabled non-atomic events. Given that the PSP does not include any other atomic blocks, the atomic block in CPU_EXEC ensures that no other event happens between the start of the atomic block, which is the activation of the clock for Wait[1], and the end of the atomic block, which is the occurrence of Skip.

This atomic block seems to cause a time digitization such that an event of the PSP can either happen before the atomic block or after the Skip at the end of it which happens exactly after one time unit. However, this digitization effect does not remove any detectable behaviour. This is because, in PAT, one can only use integers inside the timed operators and it is not possible specify an event to happen strictly after or before a specified time.

In order to add to the PIP the scheduling behaviour defined by CPU processes we use a process called CPUS (Line 41, Listing 4). For each CPU present in the execution platform, a separate instance of CPU is initiated and all the CPU instances are put in interleaving combination to define the process CPUS. Then CPUS *is put in a parallel combination with the PIP which was instrumented with* TASK *processes.* For the running example SYSTEM process, the corresponding PSP called PSP_SYSTEM is shown in Listing 5 after the PIP is instrumented with TASK and CPU processes as the last step of the PSP construction. Note that instrumentation of the PIP with TASK and CPU processes can be automated once all the placeholder task events in the PIP process description are specified and enumerated.

2.5. Schedulability Analysis

After all the described construction steps are performed, the resulting PSP is a scheduled and timed CSP model of the system. In this section, we explain how to perform schedulability analysis on a PSP. The purpose of schedulability analysis is to ensure that the control system (control software and execution platform) will satisfy its timing requirements. To verify the schedulability of the system, first we specify, on the PSP, a number of processes that, once started, have time constraints (deadlines) to finish.

The number of such time constrained processes are denoted by the macro definition _DEADLINE_COUNT. We instrument the PSP with such number of urgent event pairs d_start.i and d_end.i to mark the start and the end of the i^{th} time constrained process, respectively. The example PSP PSP_SYSTEM is shown in Listing 6 after two time constrained processes are specified in CP0 and CP1.

The deadline values for the time constrained parts are specified in an array called deadline_arr that is indexed by the id of the time constrained part. A sample deadline_arr

```
#define _DEADLINE_COUNT 2;
var deadline_arr[_DEADLINE_COUNT]= [18,9];
```

Listing 7: Deadline count and values defined for PSP_SYSTEM.

```
#alphabet DEADLINE {i:{0.._DEADLINE_COUNT-1} @ d_start.i, missed.i, d_end.i};
DEADLINE(i, d) = d_start.i ->>
                   (d_end.i ->> Skip timeout[d+1] missed.i ->> d_end.i ->> Skip);
                  DEADLINE(i, d);

DEADLINES  = ||| i:{0..(_DEADLINE_COUNT-1)} @ DEADLINE(i, deadline_arr[i]);
```

Listing 8: Descriptions of DEADLINE and DEADLINES processes.

```
#assert PSP_SYSTEM |= []!(missed.0 || missed.1);

#assert PSP_SYSTEM deadlockfree;
```

Listing 9: Assertions for PSP_SYSTEM.

for PSP_SYSTEM, which specifies deadline values 18 and 9 for the two time constrained processes, is shown in Listing 7.

We need to verify that, for each of the time constrained processes, the associated deadline is not violated. For this purpose we use a deadline violation checking process called DEADLINE(i,d) for each time constrained process (Listing 8). DEADLINE(i,d) involves a timeout event missed.i denoting a deadline violation for the specified time constrained process (i) and specified deadline value (d); a missed.i event may only happen if the corresponding d_end.i event does not happen within d time units.

To verify the schedulability, we first formulate a process called DEADLINES (Listing 8) which is the combination of all DEADLINE processes. DEADLINES is put in a parallel combination with the PSP (see Listing 6). Then, we formulate and verify a schedulability assertion that says none of the missed.i events may occur in the PSP combined with DEADLINES process. If the verification of the schedulability assertion fails, then the witness traces should be examined to see which of the deadlines can be violated and the reasons. For the example PSP PSP_SYSTEM, the schedulability assertion (Listing 9) holds.

In addition to the schedulability assertion, we may also verify other system specifications on a PSP. A PSP is a trace timewise refinement of an initial PIP implying that, regarding only the events in the PIP, all the finite traces of the PSP are also included in the traces of the PIP (Refer to Appendix A for the refinement relation.). Any safety property that holds for the PIP is also valid for the PSP. So, only the remaining liveness specifications and deadlock freedom should be verified on the PSP (Listing 9). In case of any negative verification result, the witness traces, if any, should be examined to locate the cause.

When the verification of the schedulability assertion or any of the other system specifications fails, it can be caused by any of the parts (Figure 1) that were used to construct the PSP. It is left to the designer to locate the cause and revise the related part to make the PSP schedulable and satisfy all the system specifications.

3. Case Study: Schedulability Analysis of R2-G2P Robot Control Software

In this section, we present a case study of our schedulability analysis approach. The subject of the case study is the CSP model of a robot control software. We assume there is an existing PIP of the control software and then we construct and analyse the corresponding PSP.

First, we explain the main features of the robot and the behaviour requirements. Second, we present the PIP of the control software which is modeled as an untimed process according to the specifications of the software. Third, we present the construction of the PSP. Last, we analyse schedulability of the PSP which reveals that the PSP is not schedulable. Then we show how to locate the cause of unschedulability and make the PSP schedulable by modifying the hardware mapping of the tasks.

3.1. R2-G2P Robot and behaviour Specification

The R2-G2P robot is a mobile, two-wheeled differential drive robot (Figure 2). It features many sensors and actuators therefore it is possible to define a behaviour with multiple timing requirements for reading sensors and writing to actuators. For our case study, we assume that it is equipped with a dual-core CPU. The sensors and the actuators of the robot are as follows:

- Two infrared sensors for line following that are positioned to look towards the floor.
- Two infrared distance sensors that sense the objects in the forward driving direction.
- A contact switch in the front that senses when there is a physical contact with an object.
- Two encoders (one per wheel) that sense the angular position of the wheels.
- Two servo motors that are connected to the wheels and driven by PWM (Pulse Width Modulation) signals.

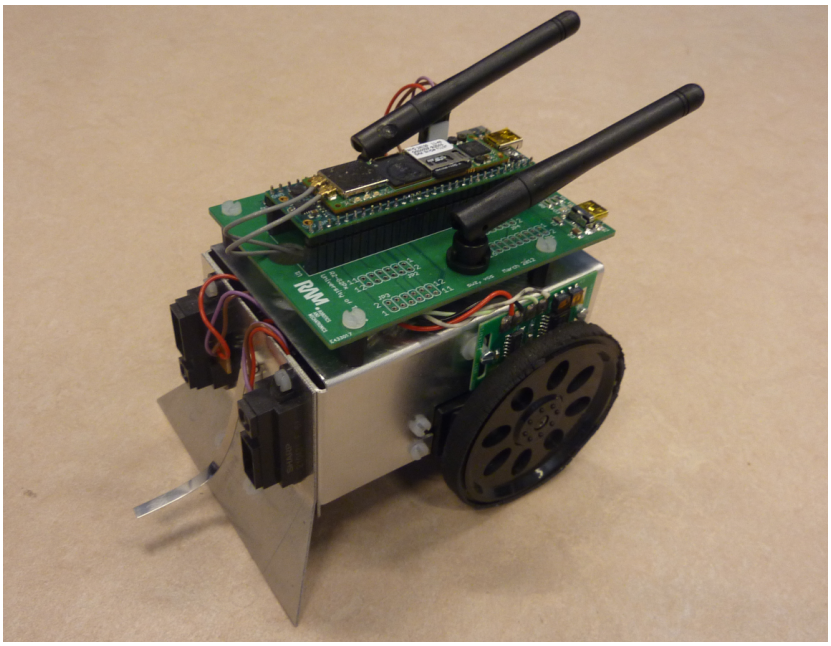

Figure 2. The R2-G2P Robot.

The robot is required to drive forward while following a black line printed on the floor. It is on the line if at least one of the line sensors detects the line. Additionally, the robot should keep a predefined distance to the obstacles that it detects in front of it. As a safety requirement, the robot needs to stop when it gets out of line (whenever neither of the line sensors detects the line) or it bumps into an obstacle. An initial control design results in a two-layer cascaded design composed of a sequence controller and a loop controller. The timing requirement for the initial control design to work is that the sequence and loop controllers have periods of 80 and 20 milliseconds, respectively.

```
1  ROBOT_CONTROL = SEQUENCE_CONTROL || LOOP_CONTROL;
```

Listing 10: Description of ROBOT_CONTROL process.

3.2. Platform-Independent Process (PIP) of Control Software

The PIP of the control software serves as a model of the software architecture that focuses on concurrent components and their synchronization. The model abstracts away from the control algorithms; all the algorithmic computations are represented by the placeholder task events in the PIP. We assume the processes in the PIP that include control computations (denoted by the task events) collaboratively achieve the control goals if the PIP obeys the specifications of the initial control design that defines the correct way for the processes to synchronize with each other and the frequencies they should be executed at.

3.2.1. Top Level Robot_Control Process

The PIP modeling the robot control software is called ROBOT_CONTROL. The external events of the ROBOT_CONTROL process are shown in Figure 3. ROBOT_CONTROL is a parallel combination of two sub-processes: SEQUENCE_CONTROL and LOOP_CONTROL. The process diagram and the PAT description for the top level decomposition of the ROBOT_CONTROL are shown in Figure 4 and in Listing 10.

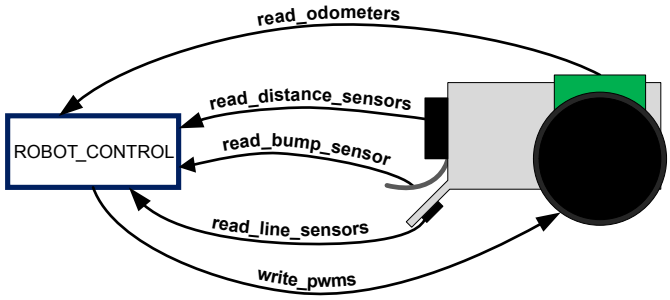

Figure 3. External events of ROBOT_CONTROL process.

Denoted by the dashed arrow in Figure 4, the parallel processes communicate with each other asynchronously using a shared variable motor_speed_setpoints. This is due to the fact that the processes are required to execute at different frequencies and that LOOP_CONTROL needs to oversample the latest value provided by SEQUENCE_CONTROL. The reading and writing to the shared variable are represented by internal events in the description of LOOP_CONTROL and SEQUENCE_CONTROL.

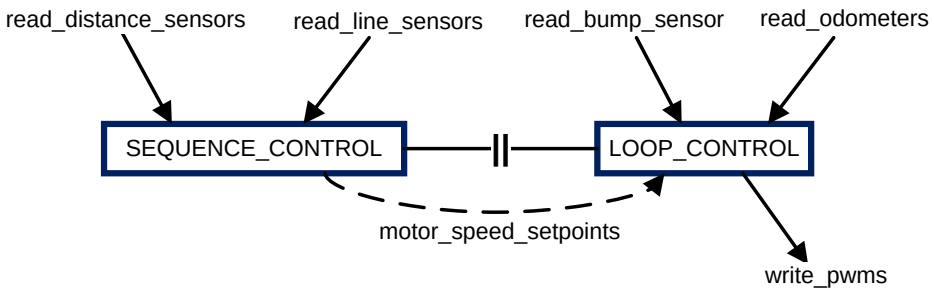

Figure 4. Process diagram showing the top level composition of the ROBOT_CONTROL process.

```
OBJECT_DISTANCE = read_distance_sensors ->
                    compute_object_distance -> distance_meas -> Skip;

ROBOT_SPEED = distance_meas -> compute_robot_speed -> robot_speed -> Skip;

MOTOR_SPEED = robot_speed -> read_line_sensors -> Skip;
              (offline -> set_zero_speed_offline -> Skip
               []
               ontheline ->
                ((compute_motor_speed_setpoint.0 -> Skip)
                 ||| (compute_motor_speed_setpoint.1 -> Skip)));
              write_motor_speed_setpoints -> Skip;

SEQUENCE_CONTROL = (OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED);
                    SEQUENCE_CONTROL;
```

Listing 11: Description of SEQUENCE_CONTROL process.

3.2.2. *Sequence_Control Process*

The process SEQUENCE_CONTROL models the sequence controller component of the control software. The PAT description of SEQUENCE_CONTROL is shown in Listing 11. As illustrated in Figure 5, at the top level it is composed of three parallel components which communicate with each other through synchronous channels.

OBJECT_DISTANCE component reads distance measurements from distance sensors and computes a single value for the distance of the robot to the obstacle. The distance computation task is denoted the event compute_object_distance in the process description.

ROBOT_SPEED implements a distance controller that periodically checks the distance of the robot to the obstacle and computes a set point value for the speed. The speed computation task is denoted by the event compute_robot_speed in the process description.

MOTOR_SPEED component computes speed set point values for the two motors of the robot after reading the robot speed set point and the line sensor values. If the line sensor values indicate that the robot is off the line (denoted by the first external choice operand starting with offline in Listing 11) then motor speed set points are set to zero; no set point computation is required. Otherwise the robot is still on the line and individual motor speed setpoints are computed with the aim of keeping the robot on the line. The computational tasks are denoted by the two compute_motor_speed.i events in the process description.

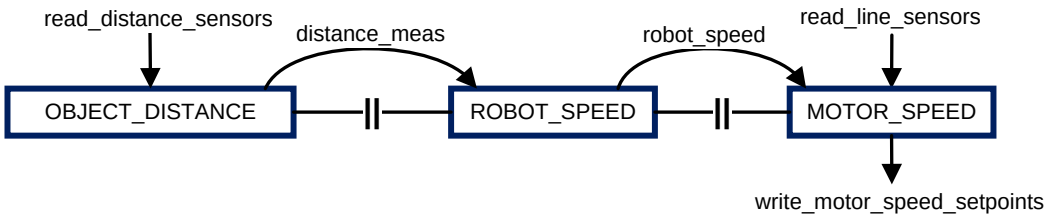

Figure 5. Process diagram showing the top level composition of SEQUENCE_CONTROL process.

3.2.3. *Loop_Control Process*

The process LOOP_CONTROL models the loop controller component that controls the speed of the motors via pulse-width modulation (PWM). In order to compute the PWM values, it reads the speed setpoints for the motors, which are output by SEQUENCE_CONTROL, and the

```
SPEEDOMETER(m_id) = compute_actual_speed.m_id -> speed_meas.m_id -> Skip;
SPEEDOMETERS = read_odometers -> (SPEEDOMETER(0) ||| SPEEDOMETER(1));

SAFETY_FILTER = read_motor_speed_setpoints -> read_bump_sensor ->
                  (safe_speed.0 -> Skip ||| safe_speed.1 -> Skip);

MOTOR_CONTROL(m_id) = (safe_speed.m_id -> Skip ||| speed_meas.m_id -> Skip);
                      compute_motor_control.m_id -> Skip;

MOTOR_CONTROLS = (MOTOR_CONTROL(0) ||| MOTOR_CONTROL(1)); write_pwms -> Skip;

LOOP_CONTROL = (SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS);
               LOOP_CONTROL;
```

Listing 12: Description of LOOP_CONTROL process.

odometer values, which are used to compute the actual speed of the wheels. Description of the LOOP_CONTROL is shown in Listing 12.

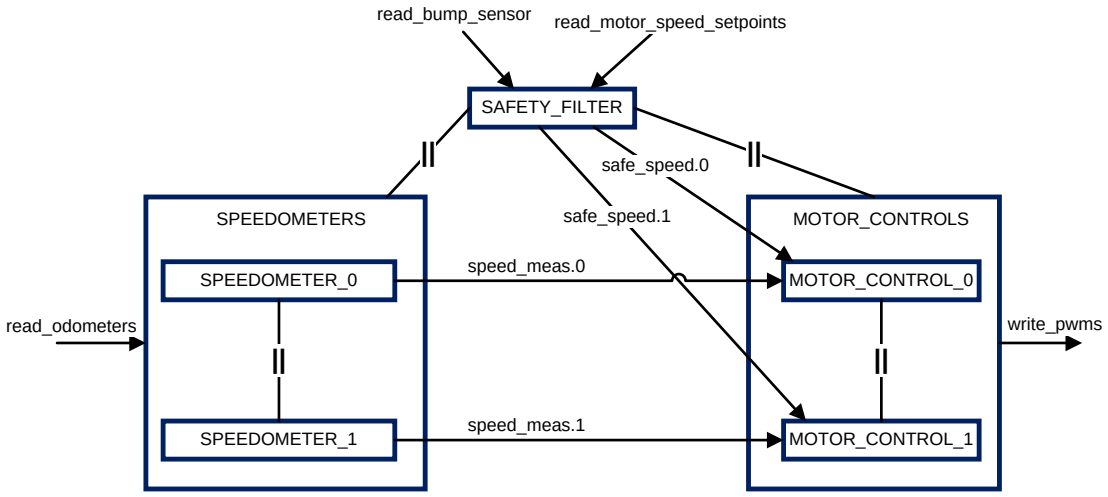

Figure 6. Process diagram of LOOP_CONTROL process.

As illustrated in Figure 6, at the top level, LOOP_CONTROL is the parallel combination of three processes. The process SPEEDOMETERS reads wheel positions from odometers and computes the actual turning speeds of the wheels; the speed computation is denoted by the compute_actual_speed.m_id events.

SAFETY_FILTER reads the bump sensor and the speed setpoint values for the the motors. Unless there is a bump, it passes the setpoint values to MOTOR_CONTROL processes unaltered. In case of a bump, it feeds zero speed values to the MOTOR_CONTROL processes.

MOTOR_CONTROLS process models the loop controllers that control the speed of the motors. The speed set points are provided by SAFETY_FILTER component and the measured wheel speeds are provided by SPEEDOMETERS. The control computations are denoted by compute_motor_control.m_id events.

3.3. Construction of the Platform-Specific Process (PSP)

We assume that the presented PIP ROBOT_CONTROL satisfies all the safety specifications of the system. In this section we explain how we construct a PSP from it.

```
#define _SEQ_PERIOD 80;
SEQUENCE_CONTROL = ((OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED)
                    ||| Wait[_SEQ_PERIOD]);
                   SEQUENCE_CONTROL;

#define _LOOP_PERIOD 20;
LOOP_CONTROL = ((SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS)
                ||| Wait[_LOOP_PERIOD]);
               LOOP_CONTROL;
```

Listing 13: Descriptions of SEQUENCE_CONTROL and LOOP_CONTROL processes after they are instrumented with periodic delays.

```
//task_arr[t_id][attr]
//attr: 0 -> BCET
//      1 -> WCET
//      2 -> MP_ID
var task_arr[8][3] = [4,7,0,  //t_id=0: compute_actual_speed.0
                      4,7,1,  //     1: compute_actual_speed.1
                      5,7,2,  //     2: compute_motor_control.0
                      5,7,3,  //     3: compute_motor_control.1
                      3,7,4,  //     4: compute_object_distance
                      1,3,5,  //     5: compute_robot_speed
                      2,3,5,  //     6: compute_motor_speed_setpoint.0
                      2,3,5]; //     7: compute_motor_speed_setpoint.1

//mp_arr[mp_id][attr]
//attr: 0 -> PRIORITY
//      1 -> CPU_ID
var mp_arr[6][2] = [2,0,  //mp_id=0: SPEEDOMETER(0)
                    2,1,  //      1: SPEEDOMETER(1)
                    2,1,  //      2: MOTOR_CONTROL(0)
                    2,0,  //      3: MOTOR_CONTROL(1)
                    1,1,  //      4: OBJECT_DISTANCE
                    1,0]; //      5: ROBOT_SPEED || MOTOR_SPEED
```

Listing 14: Task and mapped process attribute arrays specified for ROBOT_CONTROL. Schedulability analysis of ROBOT_CONTROL with this configuration fails to hold.

The construction starts with refining ROBOT_CONTROL into a timed process with platform-independent timing. Since we assume that the only events that take time are the task events, we convert all the other non-placeholder events in ROBOT_CONTROL to urgent events. Then, we modify SEQUENCE_CONTROL and LOOP_CONTROL processes to have periods of 80 and 20 time units (one time unit signifies one millisecond) by instrumenting them with Wait processes. SEQUENCE_CONTROL and LOOP_CONTROL are shown in Listing 13 after these modifications. We omit here the rest of description of the ROBOT_CONTROL process obtained after these modifications.

In the next step, we specify the best and worst case execution times, priorities and mapping of the tasks in ROBOT_CONTROL. Each task is assigned with a unique id and the task and mapped process arrays are specified (Listing 14).

When the sum (28 time units) of the WCET's of the tasks (t_id=0,1,2,3) in a single iteration of LOOP_CONTROL is inspected, it can be seen that it is larger than the expected period (20

```
#define _DEADLINE_COUNT 2;
var deadline_arr[_DEADLINE_COUNT]= [_SEQ_PERIOD,
                                    _LOOP_PERIOD];

SEQUENCE_CONTROL = ((d_start.0 ->> (OBJECT_DISTANCE || ROBOT_SPEED || MOTOR_SPEED);
                     d_end.0 ->> Skip)
                    ||| Wait[_SEQ_PERIOD]);
                   SEQUENCE_CONTROL;

LOOP_CONTROL = ((d_start.1 ->> (SPEEDOMETERS || SAFETY_FILTER || MOTOR_CONTROLS);
                  d_end.1 ->> Skip)
                 ||| Wait[_LOOP_PERIOD]);
                LOOP_CONTROL;

PSP_ROBOT_CONTROL = SEQUENCE_CONTROL || LOOP_CONTROL || CPUS || DEADLINES;

#assert PSP_ROBOT_CONTROL |= []!(missed.0 || missed.1);
#assert PSP_ROBOT_CONTROL deadlockfree;
```

Listing 15: Description of PSP_ROBOT_CONTROL after it is instrumented to specify two time constrained processes in SEQUENCE_CONTROL and LOOP_CONTROL. PSP_ROBOT_CONTROL is combined with DEADLINES process. The schedulability assertion and the assertion for deadlock-freedom are formulated.

time units) of LOOP_CONTROL. Therefore, in order to keep the execution of a single iteration of LOOP_CONTROL within 20 time units, its tasks are distributed into two CPU cores (Listing 14). Also, due to the shorter period of LOOP_CONTROL, priorities of the tasks in LOOP_CONTROL are assigned with a higher priority value than that of the tasks in SEQUENCE_CONTROL (Listing 14).

As the last step of constructing the PSP, we replace all the placeholder task events in ROBOT_CONTROL with TASK processes. The task_arr index (Listing 14) of each replaced task event is given as the argument to the TASK process replacing it. We also define _NUM_OF_CPUS as 2 and combine ROBOT_CONTROL with CPUS process, which is the combination of two CPU processes in this case. We omit here the description of the resulting PSP.

3.4. Analysis of the Platform-Specific Process (PSP)

There are two time constrained processes in ROBOT_CONTROL; deadlines defined for single iterations of the SEQUENCE_CONTROL and LOOP_CONTROL processes are 80 and 20 time units, respectively. We mark start and end points of time constrained processes and specify the deadline array as shown in Listing 15. We also combine the PSP constructed from ROBOT_CONTROL with the DEADLINES process. Then, we formulate two assertions: the schedulability assertion and the assertion for deadlock freedom (Listing 15).

We perform dense-time model checking on the schedulability assertion using PAT's verification engine with "Shortest Witness Trace with Zone Abstraction" setting. On a notebook PC with Intel Core Duo 2.53GHz CPU and 4GB of memory, the verification fails in 3 seconds giving a shortest witness trace.

Inspection of the witness trace reveals that when the tasks with ids 0 and 1, which correspond to compute_actual_speed.m_id tasks in LOOP_CONTROL, are dispatched, the execution of the task 0 may finish earlier and, since the next highest priority task (t_id=3) in the same CPU (CPU 1) waits to be released after the task 1 is finished, the task 4, a lower priority task, is scheduled in CPU 1. In this case, the task with id 3 is dispatched after the task 4 and, if they both take their WCET's to finish, the LOOP_CONTROL process misses its deadline.

```
1 //mp_arr[mp_id][attr]
2 //attr: 0 -> PRIORITY
3 //      1 -> CPU_ID
4 var mp_arr[6][2] = [2,0,  //mp_id=0: SPEEDOMETER(0)
5                     2,1,  //      1: SPEEDOMETER(1)
6                     2,0,  //      2: MOTOR_CONTROL(0)
7                     2,1,  //      3: MOTOR_CONTROL(1)
8                     1,1,  //      4: OBJECT_DISTANCE
9                     1,0]; //      5: ROBOT_SPEED || MOTOR_SPEED
```

Listing 16: The modified mapped process attribute array which makes PSP_ROBOT_CONTROL schedulable.

This is a typical example of a multiprocessor scheduling anomaly involving a counter-intuitive, hard-to-catch timing behaviour. A locally faster execution (the task with id 0 finishes before its WCET) leads to an increase of the overall execution time (execution time for a single iteration of LOOP_CONTROL process).

In order to fix this, we can modify the mapped process array by swapping the CPU_IDs of mapped processes with id's 2 and 3 as shown in Listing 16. Verification of the schedulability assertion with this modified mapped-process array ends in 16 seconds indicating that the assertion is valid. The assertion for deadlock freedom is also verified to hold for this modified configuration in 13 seconds.

4. Conclusions

We have proposed a schedulability analysis framework based on dense-time model checking in PAT tool to check the schedulability of control systems modeled in CSP. Our framework enables analysing multiprocessor schedulability of the systems modeled in CSP involving complex task synchronizations and variable task execution times which traditional schedulability analysis methods fall short on solving. Since our framework is based on model checking and the CSP model of the control system is incorporated in the analysis, the results are non-pessimistic.

We also presented a schedulability workflow associated with the proposed schedulability framework. On a control software design for a mobile robot, we demonstrated the proposed schedulability workflow where we constructed and analysed a timed and scheduled CSP model of the control system from an initial untimed and unscheduled CSP model. Schedulability analysis performed on the constructed model revealed that the system is unschedulable due to a multiprocessor scheduling anomaly. To resolve this, we modified the hardware mapping and we verified that the system is schedulable with the new configuration.

A limitation of the proposed schedulability framework is that the communication and context switching actions are assumed to take insignificant amounts of time. This is a valid assumption for the cases where the total execution time is dominated by the computational task executions and the communication and context-switching times are bounded. For such cases the times for the non-computational tasks can be accommodated within the execution times of the computational tasks.

The scalability of the proposed approach is highly dependent on the performance of the verification engine of the PAT tool and the size of state space of the analysed CSP model. Our case study indicates that the schedulability analysis of a small sized system can be performed in less than 20 seconds on a notebook PC of moderate performance. However, the scalability needs to be investigated further in the future to assess the applicability of the proposed method to more complex systems.

The schedulability framework can be extended to support scheduling schemes with dynamic priorities and/or preemption at integer time points. It will be also beneficial to include the communication times in the framework to better support the distributed control systems in which the communication events might take significant amounts of time. However, we note that these extensions would presumably increase the state space of the schedulability problems and decrease the scalability of the proposed method.

5. Acknowledgements

The authors wish to thank Jaco van de Pol for his fruitful discussions and helpful comments. We wish to thank the PAT development team, especially Yang Liu and Jun Sun, for their support in using the PAT tool. We are also grateful to the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper.

Appendix

A. Refinement Relation between PIP and PSP

We would like to show that, if only the non-placeholder events in a PIP description are considered, there is a trace timewise refinement relation between a PIP and a corresponding PSP constructed from it. The idea is to divide the construction of a PSP from a PIP into successive steps and then to show that at each step the refinement relation is preserved.

In the following formulas we adopt the notation used by Schneider in [5]. Essentially we show that when we hide all the events in the PIP and the PSP except the non-placeholder events in the alphabet of the PIP, then the PSP is a trace timewise refinement of the PIP. Let P_{PE} denote the process that results when all the events of a process *P* are hidden except the non-placeholder events for a particular PIP:

$$P_{PE} \quad = \quad P \setminus \{e : \alpha P \mid e \notin \alpha PIP \ \vee \ e \in ET\}$$

where *ET* denotes the set of placeholder task events in the PIP. Then we claim that, for a PIP and a corresponding PSP constructed from it, the following trace timewise refinement relation holds:

$$PIP_{PE} \quad {}_{T} \sqsubseteq_{TF} \quad PSP_{PE}$$

In a general sense, this kind of refinement relation between a *PIP* and a corresponding *PSP* can be demonstrated by dividing the construction of the *PSP* into successive steps and showing that the refinement relation is preserved at each step. To this end, the construction process can be divided into four steps, successively refining the *PIP* by adding scheduling and timing, and in the end obtaining a *PSP*:

$$PIP \rightarrow PIP' \rightarrow UPSP \rightarrow PSP$$

In the first step, a process *PIP'* is constructed by removing all occurrences of the placeholder task events from the *PIP*. Intuitively, *PIP'* is essentially the same process as PIP_{PE}, since none of the placeholder events in the *PIP* are involved in any synchronization; removing them is the same as hiding them, not changing the partial order of other events in the traces:

$$PIP_{PE} \quad = \quad PIP'$$

Second, we construct a process *UPSP*, an untimed version of the *PSP*, by inserting `TASK` processes in the places of previously removed placeholder events in *PIP'*, combining it with

CPUS, including particular mapped process and task attribute arrays, and then converting the resulting process to an untimed CSP process. Conversion to untimed CSP is done by removing `Wait` and `Within` operators in CPUS, changing atomic blocks to normal blocks and converting urgent event prefixes to normal event prefixes.

$UPSP_{PE}$ cannot have any new traces that *PIP'* does not have. *UPSP* preserves the process structure of *PIP'* regarding the events of *PIP'*; it only includes additional `TASK` processes which do not interfere with the events or data that are common with *PIP'*. The scheduling of `TASK` processes alters only the partial order of the communication events `ready_cpu` and `finished_cpu` which are contained in the `TASK` processes.

However, the traces of $UPSP_{PE}$ cannot be any less than the traces of *PIP'* either. This is due to that, at any time, execution of enabled `TASK` processes (an execution of a `TASK` process is defined as a pair of `ready_cpu` and `finished_cpu` occurrences) can interleave each other. So $UPSP_{PE}$ is actually a trace equivalent of *PIP'*:

$$PIP_{PE} \quad =_T \quad UPSP_{PE}$$

As the last step, we add timing to *UPSP*. We convert the `TASK` and `CPU` processes back to their timed versions. We can introduce `Wait` processes into any point in *UPSP* to introduce delays. Similarly, we can introduce `Deadline` and `Within` operators at any point in *UPSP* to introduce time constrained behaviours. We can translate any external choice into a timeout choice and any event interrupt into a timed interrupt. We can convert normal event prefixes to urgent event prefixes and enclose some processes into atomic blocks.

The resulting process is *PSP* and we claim it is a trace timewise refinement of *UPSP*. We base our argument on Schneider [5, pp. 397-403], according to which "all of the process operators preserve timewise refinement" and "timewise refinement process is compositional: a system may be refined by introducing timing information to each of its components independently."

Schneider only addresses timed CSP semantics and Stateful Timed CSP has extensions to it such as urgent event prefixes (`->>`), atomic blocks (`atomic\{..\}`), and `Within` and `Deadline` operators. All of these extensions impose time constrained behaviour. Additionally, the events in the atomic blocks have higher priority than all the other non-atomic events.

However, regarding the finite traces, the added time constrained behaviour and priority of some of the events cannot introduce any new traces; they can only restrict some of the traces. Therefore, we can make the following additions to the list in Schneider [5, pp. 400] of unary CSP operators that preserve trace timewise refinement:

if $P \ _T \sqsubseteq_{TF} \ Q$ then

$$\begin{aligned}
a \rightarrow P \ &_T \sqsubseteq_{TF} \ a \twoheadrightarrow Q \\
P \ &_T \sqsubseteq_{TF} \ atomic\,\{Q\} \\
P \ &_T \sqsubseteq_{TF} \ Q \ within[d] \qquad \text{for any d} \\
P \ &_T \sqsubseteq_{TF} \ Q \ deadline[d] \qquad \text{for any d}
\end{aligned}$$

So, *PSP* is a trace timewise refinement of *UPSP* which means that PSP_{PE} is a trace timewise refinement of PIP_{PE}:

$$UPSP \ _T \sqsubseteq_{TF} \ PSP \quad \Rightarrow \quad PIP_{PE} \ =_T \ UPSP_{PE} \ _T \sqsubseteq_{TF} \ PSP_{PE}$$

.

References

[1] Charles A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, London, UK, 1985.

[2] Andrew W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, Upper Saddle River, NJ, USA, 1997.
[3] George M. Reed and Andrew W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3):249–261, June 1988.
[4] George M. Reed. *A uniform mathematical theory for real-time distributed computing*. PhD thesis, 1988.
[5] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, New York, NY, USA, 1999.
[6] FDR2: Failures-Divergence Refinement. http://www.cs.ox.ac.uk/projects/concurrency-tools/, 2012.
[7] PAT: Process Analysis Toolkit. http://www.comp.nus.edu.sg/~pat/, 2012.
[8] Frederick R. M. Barnes and Peter H. Welch. Prioritised dynamic communicating and mobile processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.
[9] Occam 2.1 reference manual. Technical report, Inmos Limited, May 1995.
[10] Jan F. Broenink, Andrè W. P. Bakkers, and Gerald H. Hilderink. Communicating Threads for Java. In Barry M. Cook, editor, *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pages 243–262, mar 1999.
[11] Bojan Orlic and Jan F. Broenink. Redesign of the c communicating threads library for embedded control systems. In Frank Karelse, editor, *5th Progress Symposium on Embedded Systems, Nieuwegein, The Netherlands*, pages 141–156. STW Technology Foundation, 2004.
[12] Neil C. C. Brown. CCSP2: A Many-to-Many Threading Model for Multicore Architectures. In Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.
[13] Peter H. Welch and Frederick R. M. Barnes. Communicating mobile processes: introducing occam-pi. In *In 25 Years of CSP*, pages 175–210. Springer Verlag, 2005.
[14] Maarten M. Bezemer, Robert J.W. Wilterdink, and Jan F. Broenink. Luna: Hard real-time, multi-threaded, csp-capable execution framework. In Peter H. Welch, Adam T. Sampson, Jan B. Pedersen, Jon M. Kerridge, Jan F. Broenink, and Frederick R.M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press.
[15] Martin Korsgaard and Sverre Hendseth. Combining EDF scheduling with occam using the toc programming language. In Peter H. Welch, Susan Stepney, Fiona Polack, Frederick R. M. Barnes, Alistair A. McEwan, Gardiner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors, *The thirty-first Communicating Process Architectures Conference, CPA 2008, organised under the auspices of WoTUG and the Department of Computer Science of the University of York, York, Yorkshire, UK, 7-10 September 2008*, volume 66 of *Concurrent Systems Engineering Series*, pages 55–66. IOS Press, 2008.
[16] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Etienne Andre. Modeling and verifying hierarchical real-time systems using stateful timed CSP. *ACM Transactions on Software Engineering and Methodology*. To appear.
[17] Jun Sun, Yang Liu, Jin Song Dong, and Xian Zhang. Verifying stateful timed CSP using implicit clocks and zone abstraction. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer, 2009.
[18] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2nd edition, October 2004.
[19] Alan Burns. Advances in real-time systems. chapter Preemptive priority-based scheduling: an appropriate engineering approach, pages 225–248. Prentice Hall, Upper Saddle River, NJ, USA, 1994.
[20] Martin Korsgaard and Sverre Hendseth. Schedulability analysis of malleable tasks with arbitrary parallel structure. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, volume 1, pages 3–14. IEEE, August 2011.
[21] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
[22] Elena Fersman, Pavel Krcal, Paul Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 205(8):1149–1172, August 2007.
[23] UPPAAL. http://www.uppaal.com/.
[24] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.
[25] Alexandre David, Jacob Illum, Kim G. Larsen, and Arne Skou. Model-based framework for schedulability analysis using uppaal 4.1. In *Model-Based Design for Embedded Systems*, pages 93–119. CRC Press,

2010.

[26] Marius Mikučionis, Kim G. Larsen, Jacob I. Rasmussen, Brian Nielsen, Arne Skou, Steen U. Palm, Jan S. Pedersen, and Poul Hougaard. Schedulability analysis using uppaal: Herschel-Planck case study. In *Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation - Volume Part II*, ISoLA'10, pages 175–190, Berlin, Heidelberg, 2010. Springer Verlag.

[27] Tobias Amnell, Elena Fersman, Leonid Mokrushin, Paul Pettersson, and Wang Yi. TIMES: a tool for schedulability analysis and code generation of Real-Time systems. In Peter Niebert and Kim G. Larsen, editors, *Proceedings of FORMATS'03*, Lecture Notes in Computer Science, pages 60–72. Springer Verlag, 2004.

[28] Pavel Krcal, Martin Stigge, and Wang Yi. Multi-processor schedulability analysis of preemptive real-time tasks with variable execution times. In *Proceedings of the 5th international conference on Formal modeling and analysis of timed systems*, FORMATS'07, pages 274–289, Berlin, Heidelberg, 2007. Springer-Verlag.

[29] Patrice Brémond-Grégoire and Insup Lee. A process algebra of communicating shared resources with dense time and priorities. *Theoretical Computer Science*, 189(1-2):179–219, December 1997.

[30] Mohammadreza Mousavi, Michel Reniers, Twan Basten, and Michel Chaudron. PARS: A process algebra with resource and schedulers. In Kim G. Larsen and Peter Niebert, editors, *Proceedings of the First International Workshop on Formal Modeling and Analysis of Timed Systems (Formats'03)*, volume 2791 of *Lecture Notes in Computer Science*, pages 134–150. Springer-Verlag, Berlin, Germany, May 2004.