Designing Reusable and Run-Time Evolvable Scheduling Software

Güner Orhan · Mehmet Akşit · Arend Rensink

Abstract Scheduling processes have been applied to a large category of application areas such as processor scheduling in operating systems, assembly line balancing in factories, vehicle routing and scheduling in logistics and timetabling in public transportation, etc. In general, scheduling problems are not trivial to solve due to complex constraints. In this paper, we consider *reusability* and *run-time evolvability* as two important quality attributes to develop cost-effective software systems with schedulers. Although many proposals have been presented to enhance these quality attributes in general-purpose software development practices, there has been hardly any publication within the context of designing scheduling systems. This paper presents an application framework called First Scheduling Framework (**FSF**) to design and implement schedulers with a high-degree of reusability and run-time evolvability. The utility of the framework is demonstrated with a set of canonical examples and evolution scenarios. The framework is fully implemented and tested.

1 Introduction

Scheduling is a decision-making process in which the *resources* are allocated to the *activities*. Scheduling processes have been applied to a large category of application areas such as processor scheduling in operating systems [43], car scheduling in elevator systems [35], facility scheduling at airports [40], antenna scheduling in radar systems [23], work-force scheduling in project management [30] and assembly line

Güner Orhan
University of Twente
E-mail: g.orhan@utwente.nl

Mehmet Akşit
University of Twente
E-mail: m.aksit@utwente.nl

Arend Rensink
University of Twente
E-mail: arend.rensink@utwente.nl

balancing in factories [7], vehicle routing and scheduling in logistics and timetabling in public transportation [14].

Implementing software systems that incorporate scheduling systems can be a time consuming process. In addition to dealing with well-known challenges in designing software systems, the software engineer has to define and implement the required tasks, resources, associated parameters, objectives, strategies, and the constraints, and/or algorithms. Dealing with all these constraints can be a very time consuming and error-prone tasks. For example, the constraints must be considered in a very precise and robust manner: Tasks have to be scheduled within their *life-scope*; periodic tasks have to be spawned at each inter-arrival time; the resource requirements of the allocation have to be realized for each task; the precedence relations have to be satisfied for each allocation; the capacity constraints of resources have to be satisfied; the preemption capability is supposed to be realized; the migration capability has to be satisfied; the mutual exclusion constraint among resources have to be satisfied. A highly *reusable* and *run-time evolvable* framework designed specifically in the scheduling domain can ease this burden. Domain specific class libraries with the necessary operations and attributes can be instantiated with the parameters of the desired scheduler.

In this paper, we consider *reusability* and *run-time evolvability* [17,34] as two important quality attributes.

In the literature, to the best our knowledge, the studies do not aim to develop reusable and run-time evolvable scheduling software implementation; they rather concentrate on specific application-dependent solutions. Since there exists hardly any generic and expressive library, framework or design environment, the software engineer has to implement all the necessary scheduling abstraction by herself, which increases the complexity and effort. In addition, due to lack of run-time evolution support, maintenance of continuously operating scheduling systems becomes a challenge.

This paper introduces an object-oriented application framework called **FSF** which can be utilized in implementing schedulers with a high-degree of reusability and run-time evolvability. The utility of the framework is demonstrated with a set of canonical examples and evolution scenarios. The framework is fully implemented and tested.

The remaining sections of this paper are organized as follows: the next section presents the problem statement and objectives that are addressed in this paper. The related work is summarized in Section 3. The section 4 presents the necessary background knowledge to understand the terms and symbols used throughout this paper. The software architecture of the framework is described in Section 5. In Section 6, as case studies, a set of canonical examples is introduced to evaluate the proposed framework. Finally, the evaluation of the framework and concluding remarks are presented in Section 7.

2 The Problem Statement and Objectives

A considerable number of publications have been presented in the literature to guide software engineers in designing software systems [44]. It is generally agreed that certain quality attributes play an important role along this line. Although many proposals have been presented to enhance the quality attributes *reusability* and *run-time evolvability* in software development practices, there has been hardly any publication within the context of designing scheduling systems.

Our focus on these software quality attributes in this paper is limited to the scheduling domain.

We adopt the term *reusability* as *ease of use* of a dedicated software library and associated tools in creating a large category of scheduling systems. To this aim, to create a particular scheduling system, the code written from scratch must be much less than the code of the library that is reused. To fulfil the reusability requirement, the concept of *application-frameworks* [28] can be utilized. An object-oriented application framework is defined within a context of an application domain and consists of a set of dedicated class hierarchies which can be instantiated and/or sub-classed to create a specific application in that domain. An important motivation for using this approach is two-fold: to provide a reusable programming library for the programmer within the scheduling domain, and to give flexibility to the programmer to alter the library if needed.

We adopt the term *run-time evolvability* as an ease of modification of an existing scheduling software with respect to a new *meaningful* set of user requirements. Since many of scheduling systems, such as airport systems and production systems must be continuously operational, solutions to the new requirements must be introduced to the system at run-time. The term *meaningful* here refers to the fact that requirements are natural and defined within a single application context. It is assumed for example that an airport scheduling system is not expected to evolve into an elevator scheduling system.

Within the context of this paper, *run-time evolvability* must be supported for the following cases:

A. Changing (adding, removing or modifying) resources and/or tasks;
B. Changing the optimization criteria based on the number of existing tasks.
C. Changing the timing constraints of the tasks.
D. Changing the dependency specifications among existing tasks.
E. Changing the attributes of existing tasks.

3 Related Works

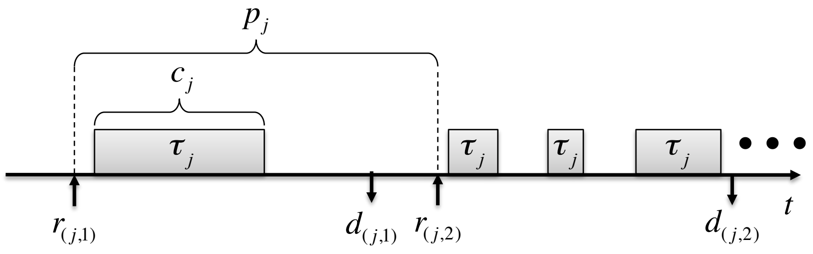
There have been a considerable number of publications which reports on the practical applications of frameworks [31, 28, 1]. To the best of our knowledge, none of them has been applied to the domain of scheduling.

There are many researchers focusing on scheduling problems, and such much research work has been published in this area [20, 10]. Accordingly, a large category of algorithms has been developed. In addition, different kinds of solver-based solutions have been studied and presented in the literature to address planning and scheduling problems [18, 22, 25]. There exists also a study [29] which presents a formal framework to implement reusable schedulers. However, these publications do not aim at creating a framework satisfying *reusability* and *run-time evolvability* as defined in this paper. In our related work, a software product line approach for schedulers is presented [36]. This paper, however, focuses on the early phases of product-line software development processes, as such it mainly deals with product management and feature models. The software architecture, run-time environment and the result of execution of schedulers are not addressed.

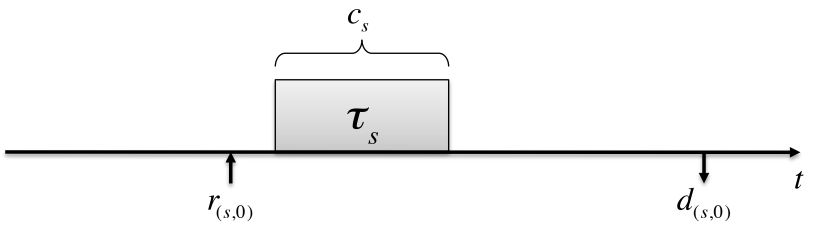
4 Background on Scheduling Domain

Constraints and *objectives* are the two basic factors that influences a scheduling process. Within the context of scheduling domain, *constraints* narrow down solution space, whereas *objectives* are used for determining the perfect fitting solution. From this perspective, any *scheduling problem* is an *optimization problem*.

The term *job* is defined as a set of *works* which need to be completed by utilizing *resources*. In addition to the term *job*, different names are also used, such as *task* [10], *activity*, *operation*, etc. [20,38]. For the sake of unity, we adopt the term *task* throughout the article.



(a) A task with infinitely many number of *instances*.



(b) A task with one *instance*.

Fig. 1: An example schedule of two tasks and their fundamental timing related attributes. The terms in the subscript of J refer to the index of a task and an index of an instance of a task, respectively.

A task is defined using the following fundamental timing related attributes:

- $r_{(i,j)}$: the *release time* of the instance j of a task i. It denotes the *earliest start time* of a task.

- $c_{(i,j)}(c_i)$: the *execution time* of the instance j of the task i. It corresponds to the duration when a task needs to be executed. It has similar meanings with the terms *processing time* in [38,47]; *execution time* in [9,38]; *computation time* in [10,42]; *run time* in [32]; and *worst-case execution time* in [13,46]. The term *worst-case execution time* corresponds to the term *execution time* in our terminology.
- $d_{(i,j)}$: the *deadline* (*due date*) of the instance j of a task i. It means the *latest finish time* of a task.
- p_i: the *period* of a task i. Unlike the previous attributes, this does not depend on any instance, but on a task; that is why it is shown with a single subscript. It corresponds to the inter-arrival time between two consecutive *instances* of a task. For tasks depicted in Figure 1b, the term *period* has no meaning.
- $w_i(w_{(i,j)})$: the *priority* of a task i (the instance j of a task i). Similarly, it may be either to a task- or an instance-dependent term, which indicates the relative importance of a task (an instance of a task) among other tasks (instances of tasks). The term *weight* is a synonym of the term *priority*.

All of these terms are shown in Figure 1 except the attribute *priority*. In Figure 1a, any task instance is supposed to complete its execution within the duration between its release time and deadline, which is called *life-cycle*.

In scheduling domain, the problems are defined using the traditional 3-field notation which has been introduced by Graham [20]:

$$\alpha|\beta|\gamma \tag{1}$$

According to this notation in Definition 1, any scheduling problem consists of three parts, namely the *machine environment* (α) (*resources*) on which the *tasks* are executed; the *task characteristics* (β) (*constraints*) which are restrictions based on tasks and resources; and *optimality criteria* (γ) (*objectives*); these are aimed to be optimized.

4.1 Machine Environment

Machine environment is denoted by a pair $\alpha_1\alpha_2$, namely machine identifier and number of machines, respectively. The terminology introduced in this section is used throughout this paper.

In [20], 6 fundamental machine identifiers (α_1) are introduced:

- 1: It refers to a single machine environment in which at most one task can execute at each time instant.
- P: It refers to the environment with more than one identical machines, meaning that the execution speed of the tasks does not vary from one machine to another. Moreover, unless indicated to the contrary, more than one task is eligible to execute on the machines in parallel.
- Q: Unlike identical machines, the resources have different execution speed; therefore the execution time of a task depends on the speed of a machine. The execution time of the task j on the resource m is denoted by $c_{(j,m)}$.

– O: It is a machine environment called *open-shop*, where the only restriction is that instances of tasks cannot execute on different resources in parallel. Moreover, there exists one-to-one relation between the instances of each task and resources, meaning that each instance has to execute on a dedicated resource.
– F: The *flow-shop* machine environment differs from the *open-shop* in one aspect. Each instance of a task has a predefined execution path on each machine.
– J: The *job-shop* machine environment is more like *flow-shop* environment in such a way that each instance of a task has a predefined path of execution, yet a task does not need to be executed on each machine.

We express the number of the machines (α_2) using positive natural numbers $\mathbb{N}^+$. For instance, 2-machine flow-shop environment is denoted by $\alpha = F2$.

4.2 Scheduling Characteristics

In this subsection, we will further elaborate on the scheduling characteristics (β) introduced in Definition 1. The constraints are defined as follows:

– $\beta_1 = \{pmtn, \varepsilon\}$: It expresses preemption ability (*preemptability*) of a task. It gives a capability to a scheduler to suspend the execution of a task before completion.
– $\beta_2 = \{r_j, \varepsilon\}$: If this symbol exists in problem definition, then a task may have specific release time. Otherwise, any task may start at anytime.
– $\beta_3 = \{prec, \varepsilon\}$: Precedence relation of tasks blocks the commence of an instance of a task in case it *depends* on the completion of another task. For this reason, it is also called as *dependency* in the literature.
– $\beta_4 = \{M_j, \varepsilon\}$: The constraint *machine eligibility* obliges tasks to run on only specific set of resources.
– $\beta_5 = \{p_j = p, \varepsilon\}$: It is used to define all of the tasks with fixed execution time p.
– $\beta_6 = \{d_j = d, \varepsilon\}$: If this constraint is specified, it is guaranteed that each instance of a task is completed before fixed deadline d.
– $\beta_7 = \{s_{jk}, \varepsilon\}$: This constraint is known as *sequence dependent setup time* and defines the necessary time duration between the completion of the task j and beginning of the task k.
– $\beta_8 = \{batch(b), \varepsilon\}$: A resource may execute b tasks simultaneously if this constraint is defined. An entire batch is finished when the last task of a batch has been completed and the execution times of tasks may not be equal.
– $\beta_9 = \{prmu, \varepsilon\}$: It is only meaningful in flow-shop machine environment. The queues in front of a resource have *First-In-First-Out* policy. Therefore, the order of tasks is maintained on each machine.

The character ε indicates that the corresponding scheduling characteristic is not in the list β.

Criteria	Formula
Lateness	$L_j = C_j - d_j$
Earliness	$E_j = max\{-L_j, 0\}$
Tardiness	$T_j = max\{L_j, 0\}$
Absolute Deviation	$D_j = \|L_j\|$
Squared Deviation	$S_j = (L_j)^2$
Unit Penalty	$U_j = \begin{cases} 0 & \text{if } C_j \leq d_j \\ 1 & \text{Otherwise} \end{cases}$
Makespan	$C_{max} = \max\limits_j C_j$

Table 1: The commonly known functions defined in [8].

4.3 Scheduling Objectives

In the literature, we have found 7 criteria formulated based on the completion times of tasks, which is denoted as C_j [8]. The objectives are defined by *minimizing* or *maximizing* the functions shown in Table 1.

The quality of calculated schedule is measured based on the functions above. From this perspective, the aim of minimizing the function *lateness* is to complete the tasks immediately after their release time, whereas the function *earliness* is the complement of *lateness*. The objective of *minimizing* the function *tardiness* aims to avoid only the deadline misses regardless to the placement of tasks in time. The quality of a schedule remains the same if each task is scheduled and completed before its deadline. The remaining items in the list (Table 1) are derived from these functions and are considered self-explanatory. In addition, the common objective functions are formulated as maximum, summation and weighted summation over tasks. For instance, the maximum $L_{max} = \max\limits_j L_j$, the summation $L_{sum} = \sum\limits_j L_j$ and the weighted summation $L^w_{sum} = \sum\limits_j w_j L_j$ are versions of the objective *lateness*. The linear combinations of these formulas are also possible.

5 The Software Architecture of the Framework and its Configuration

In the following subsection, we present the software architecture of our scheduling framework. In subsection 5.2, we illustrate how this framework can be configured to create a scheduler as a solution to an example problem.

We define software architecture as an abstract (blue-print) representation of a software system [3]. Diagrams describing software architecture can ease understanding the essential elements of software systems. In addition, software architecture plays an important role in determining the software quality of systems. In the following subsections, both static and dynamic models of the architecture of the framework are presented. The static model is expressed in an UML component diagram notation, which shows the logical structure of our framework. It represents the important abstractions, called components which have well-defined interfaces. Each component corresponds to a piece of object-oriented program that implements a logical concern.

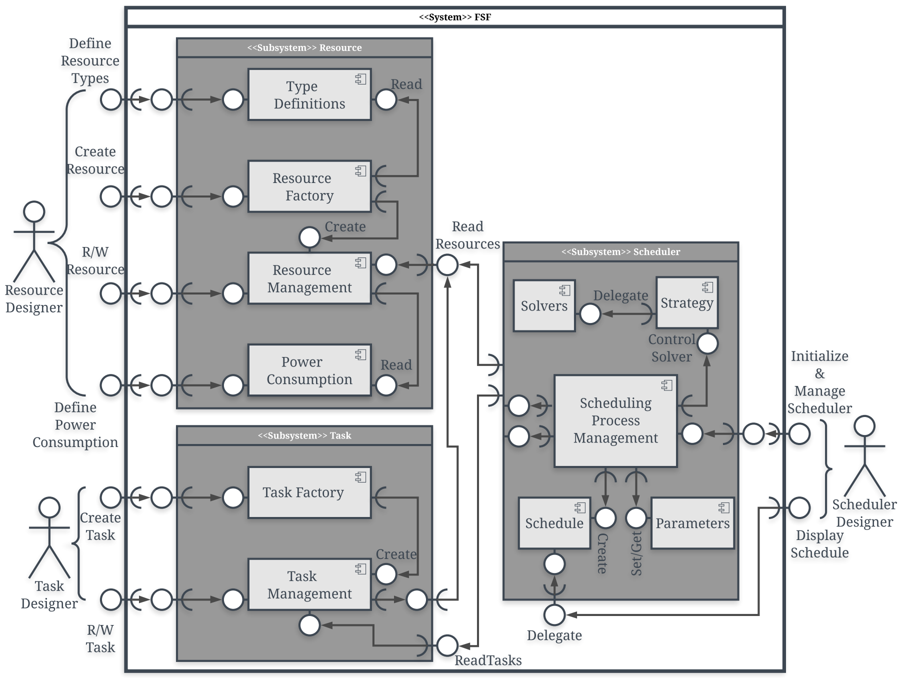


Fig. 2: The software architecture of the framework FSF depicted in UML component diagram notation.

A component can only be invoked through its interface functions. Subsystems group related components together. The dynamic model is expressed in an UML sequence diagram notation. It shows how components interact with each other to perform a system-wide behavior. In our example, we utilize sequence diagrams to express the instantiation processes in creating scheduling software.

5.1 A Component Diagram of the Software Architecture of First Scheduling Framework

To fulfill the objectives given in Section 2, we have designed and implemented a comprehensive framework [28] called **FSF** (First Scheduling Framework). The software architecture of **FSF** is designed after an extensive study of the scheduling theory. The components and the relationships of the architecture are derived from the essential concepts of the theory. Furthermore, the architecture is justified by considering the applications of the theory to a large category of scheduling problems. It is implemented using an object-oriented library and supported by a set of open-source software and own developed tools.

The software architecture of **FSF** is symbolically shown in Figure 2. The overall architecture is depicted as a large rectangle with thick lines and denoted by the

UML stereotype (≪*System*≫). The architecture consists of three subsystems: *Resource*, *Task* and *Scheduler*. These are shown as dark gray rectangles and are denoted by the UML stereotype (≪*Subsystem*≫). The components are shown as light gray rectangles and placed in the subsystems. The users of the system are shown using the UML actor notation (웃). These indicate the roles and named as *Resource Designer, Task Designer* and *Scheduler Designer*. In practice, one or more person can fulfill the roles. Interfaces are shown using specific symbols. Components exchange information among each other through interfaces. These are classified under provided (○) and required (→) interfaces. The direction of an arrow indicates a dependency relationship from a required to a provided interface.

5.1.1 The Subsystem Resource

The subsystem *Resource* contains four components: *Type Definitions*, *Resource Factory*, *Resource Management* and *Power Consumption*. The interfaces of these components are exported to the user *Resource Designer*.

The component *Type Definitions* includes all resource types defined. Currently, three kinds of abstract resource types are provided: *Active*, *Passive* and *Composite*. Abstract resource types are parameterized to create the concrete resource types such such as *Memory, CPU, Machine, Antenna, Bus, GPU, Sensor,* etc.

The user *Resource Designer* interacts with the component *Resource Factory* to create instances of the desired resources by "using" the predefined concrete resource types.

The component *Resource Management* contains all the instances and provides an interface to the user for reading and writing their properties. Furthermore, a read interface is provided to the subsystem *Scheduler*.

The component *Power Consumption* is used to read and/or write the power-consumption related properties of instances. The motivation for defining a separate component is to provide sharing: different kinds of instances may share similar power-consumption characteristics. In this case, these instances can simply denote to the same power-consumption definition.

The rationale to define the subsystem *Resources* in this way is to create a hierarchically organized resource structures, which is motivated in the following:

According to [47], the resources in computing systems are classified as either *active* or *passive*. While a task can only be executed on an *active* resource, it may also require one or more *passive* resources. The traditional resource model introduced in [20] does only support active resource. This makes it cumbersome to express tasks which require also passive resources.

Recently, general purpose computing on graphic cards (GP-GPU) has gained importance for the problems which can be divided into sub-problems such as rendering images, performing audio operations, etc., where each of them can be executed in parallel. While defining resources, it may be necessary to consider the hardware architecture of GPU's. This requires hierarchical resource models where each resource element in hierarchy may define its own rules of accessibility. To represent complex resource structures such as the ones adopted in GPU's, we define both active, passive and composite resources. Active and passive resources are represented as the terminal nodes in the hierarchy.

A composite resource may embody one or more active, passive and/or composite resources. The accessibility of resources is defined as follows: A task running on an active resource has access to the terminal resources of each of its ancestors, and all the terminal resources of each of its sibling composite resources. For example, assume that a task executes on cpu shown in Figure 3. It has access to the terminal resources of its ancestors, antenna, bus, memory-1 memory-2, temp_sensor and prox_sensor; and the terminal resources of its sibling composite resources, cache_cpu-1 and cache_cpu-2.

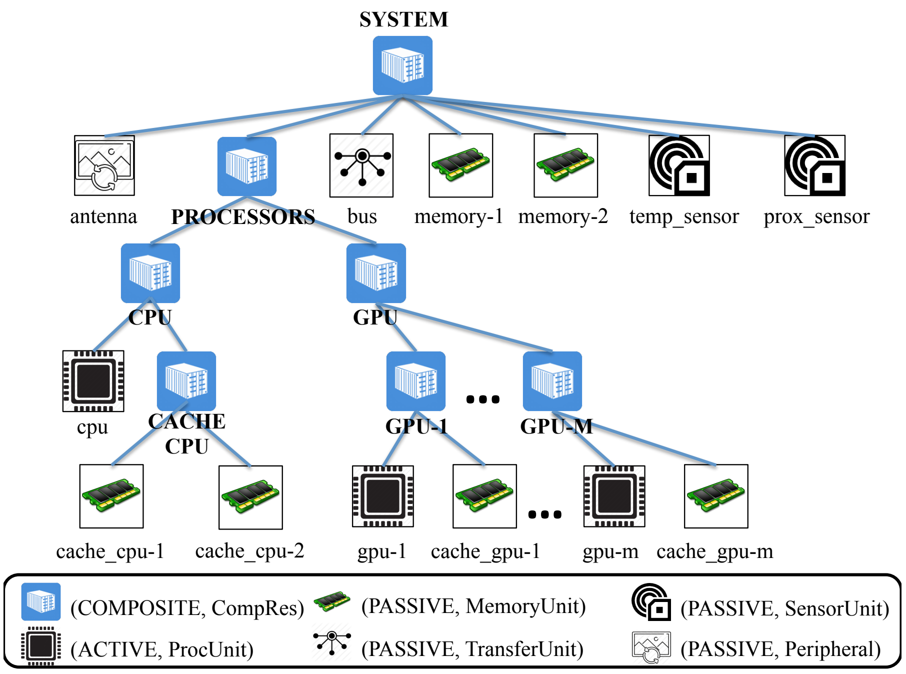


Fig. 3: An example *resource tree* defines the accessibility relation among the resources.

Recently, reducing energy consumption has become more and more important. For this reason, for example, *Dynamic Voltage Scaling (DVS)* have been introduced to reduce power consumption of processing units [11,33,37]. This requires dedicated task scheduling. To reduce energy consumption within the timing constraints, the scheduler has to consider both voltage levels and corresponding executing speeds.

To express such scheduling problems, in **FSF** power consumption is explicitly modeled in two options: *discrete-* and *continuous-state power consumption* options. For the resources specified as the former, there exist power states, each of which is the pair of the value *sc* ($0 < sc \leq 1.0$) and the power consumption value *pc*; whereas for the resources belonging to the latter, any value between minimum and maximum power scales can be chosen.

In [24], resources are categorized as *multi* or *single-unit* capacity. A resource with *multiple units*, each of which is serially accessible entity is reserved partially to more than one tasks; whereas a *single-unit* resource can only be accessed by a task at a time. Therefore, we defined the numeric attribute *capacity* representing the number of units for resources with both single- and multi-unit capacities.

To express simultaneous access in utilizing the capacity of a resource as explained in [10], we defined the term *mode*. If a resource operates at a *shared* mode, each capacity unit can be accessed by tasks, simultaneously. Otherwise, the resource can either work in *capacity-based* exclusive mode at which each capacity unit can be accessed by at most one task at a time or *semantic-based* exclusive mode at which the utilization of any capacity unit of a resource is blocked in case a task is executing on one of its exclusive resources.

5.1.2 The Subsystem Task

The subsystem *Task* includes two components: *Task Factory* and *Task Management*.

The user *Task Designer* utilizes the component *Task Factory* to create instances of tasks.

The component *Task Management* contains all the instances and provides an interface to the user *Task Designer* for reading and writing their properties. Furthermore, a read interface is provided to the subsystem *Scheduler*.

Like resources, tasks can also have composite structure. A task can be classified either *composite* or *terminal*. The tasks assigned to a composite resource are recursively dispatched to resources within its life-cycle until there is no composite task left. The design rationale for this way of allocation of tasks is to ease the scheduling process since it divides a scheduling problem into simpler sub-problems and deals with each of them in its own time scope. Since the computational complexity is supposed to decrease, this process should reduce the execution time of the scheduler. In addition, it groups the relatively similar sub-tasks which have the same resource requirements.

There are also a number of task attributes which are not shown in Figure 2 for brevity reasons. Some essential ones are described in the following:

Class *Time* has been defined to provide system-wide consistency for time-related attributes, and it is used within the definition of tasks. It has class variables such as *resolution of time*, and *unit of time*.

The attribute *precedence constraint* is also used in the definition of tasks. It ensures tasks to execute in certain order [10]. The predecessor task has to complete its execution to let the successor task of it start.

In [2], a precedence constraint has been expressed by *data dependency*; a predecessor task fires a token when it finishes and the successor task has to consume this token in order to start. We have adopted a more expressive constraint specification, which extends the token-based dependency with the relational operators AND, OR and the temporal operator AFTER.

The attribute *resource requirements* is defined for each task to express capacity requirements for a set of actual resources belonging to the same concrete resource type.

For brevity, the attributes that are used in tasks such as *time*, *precedence* and *resource requirements* are not shown in Figure 2. These are set in the component *Task Management*.

The other timing related attributes of tasks will be handled in Section 6.

5.1.3 The Subsystem Scheduler

The subsystem *Scheduler* consists of five components: *Scheduling Process Management*, *Parameters*, *Strategy*, *Solvers* and *Schedule*.

The component *Scheduling Process Management* functions as the coordinator. To this aim, it first interacts with the user *Scheduler Designer* to set the scheduler-related properties and store them in the component *Parameters*, then retrieves the information about the resources and tasks from the corresponding subsystems, determines the strategy to be used, and activates the solver. Finally, it stores the result of the solver in the component *Schedule*.

The component *Strategy* determines which solver algorithm to be utilized. To accomplish this, it interacts with the *Scheduling Process Management* and makes a request to the component *Solver* to select a particular solver algorithm.

The component *Solvers* incorporates a set of solver algorithms.

In addition to contain the schedule, the component *Schedule* provides the scheduling execution context and offers various utilities to display the results in different output format.

To realize the token-based data dependency explained in Section 5.1.2, each scheduler instance includes an attribute corresponding to a token pool.

We consider the scheduling problem as an optimization problem; This requires the definition of the optimization criteria. In Section 4.3, various criteria are shown in Table 1. The *purpose* of the optimizer is to either *minimize* or *maximize* the selected criterion. In our framework, all the criteria that are shown in the table have been implemented.

The scheduling policy is used to determine the relative importance of tasks for an underlying system. Currently, our framework supports the policies FIFO (*First-In-First-Out*), EDF (*Earliest Deadline First*), SJF (*Shortest Job First*), LJF (*Longest Job First*), RM (*Rate-Monotonic*, i.e. Shortest Period First), ERT (*Earliest Release Time*). It is also possible to define new policies by modifying the related parts in the framework.

Since our framework adopts solver-based approach, the solver has to be specified. The available solvers are SCIP, *MiniSat*, *MipWrapper*, *Mistral*, *Mistral2*, *SatWrapper*, *Toulbar2* and Walksat.

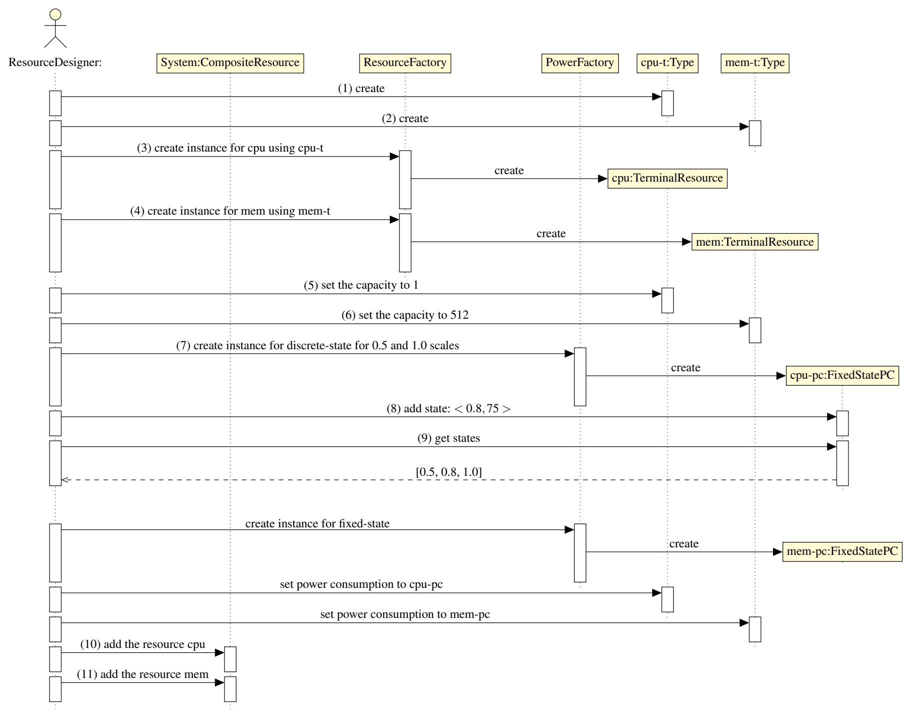

ResourceDesigner:
System:CompositeResource
ResourceFactory
PowerFactory
cpu-t:Type
mem-t:Type
(1) create
(2) create
(3) create instance for cpu using cpu-t
create
cpu:TerminalResource
(4) create instance for mem using mem-t
create
mem:TerminalResource
(5) set the capacity to 1
(6) set the capacity to 512
(7) create instance for discrete-state for 0.5 and 1.0 scales
create
cpu-pc:FixedStatePC
(8) add state: < 0.8, 75 >
(9) get states
[0.5, 0.8, 1.0]
create instance for fixed-state
create
mem-pc:FixedStatePC
set power consumption to cpu-pc
set power consumption to mem-pc
(10) add the resource cpu
(11) add the resource mem

Fig. 4: Sequence diagram to create and configure cpu and mem resources; and add them to the composite resource System.

5.2 Instantiation of the Framework to Create a Scheduler

Application frameworks [28] offer a reusable library in a certain domain which must be instantiated and if necessary extended to create a particular application. In our approach, to create a dedicated scheduler, the framework must be instantiated according to the requirements of the desired scheduler. Since run-time evolvability is one of the key objectives, an instantiation process is realized at run-time.

In the following subsections we illustrate an instantiation process for a particular scheduling example in three steps: instantiation of resources, tasks, and the scheduler.

5.2.1 Instantiation of Resources

Assume that we would like to create an implementation of a resource model with two elements: cpu and mem. In **FSF**, as shown in Figure 4. this can be realized at run-time by calling on the necessary operations on the corresponding components/objects.

Figure 4, shows a sequence diagram in UML to illustrate the creation process. On the top left of the figure, as an UML actor notation, the role *Resource Designer* is shown who is in charge of defining the resource model. The vertical bars on the most left side of the figure depict the actions that are initiated by *Resource Designer*. The

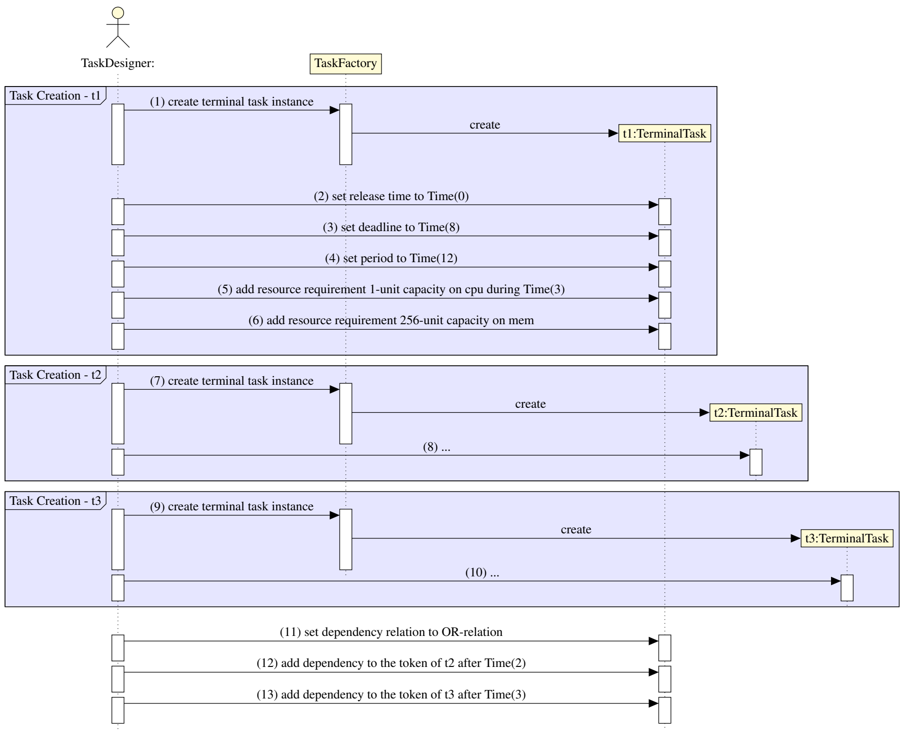

TaskDesigner:
TaskFactory
Task Creation - t1
(1) create terminal task instance
create
t1:TerminalTask
(2) set release time to Time(0)
(3) set deadline to Time(8)
(4) set period to Time(12)
(5) add resource requirement 1-unit capacity on cpu during Time(3)
(6) add resource requirement 256-unit capacity on mem
Task Creation - t2
(7) create terminal task instance
create
t2:TerminalTask
(8) ...
Task Creation - t3
(9) create terminal task instance
create
t3:TerminalTask
(10) ...
(11) set dependency relation to OR-relation
(12) add dependency to the token of t2 after Time(2)
(13) add dependency to the token of t3 after Time(3)

Fig. 5: Sequence diagram to create and configure the tasks t1, t2 and t3.

other vertical bars correspond to the instances which are involved in the interaction process. The types of these components/objects, namely *SystemCompositeResource, ResourceFactory* and *PowerFactory*, are represented at the top of the picture and linked to their instances by dashed-lines; The sequence of interactions are from top to bottom. In our framework, *System* incorporates all the resources that are created. Initially, *System* is empty.

As shown in the figure, the sequence of call have the following meaning: In the first two calls (1) and (2), the *Resource Designer* creates the identities of cpu and mem types. The text on the call arrows illustrates their meaning informally. In calls (3) and (4), by calling on *ResourceFactory* with the identities as parameters, actual resource objects are created. In calls (5) and (6), the capacities of cpu and mem are defined as 1 and 512, respectively. In calls (7), (8) and (9), the power consumption characteristics of the resources are defined as *discrete states*. For illustration purposes, the call (9) is defined as a read operation. The dashed line from right to left illustrates the response for this call. Finally, in calls (10) and (11) are used to add the created resources to the system.

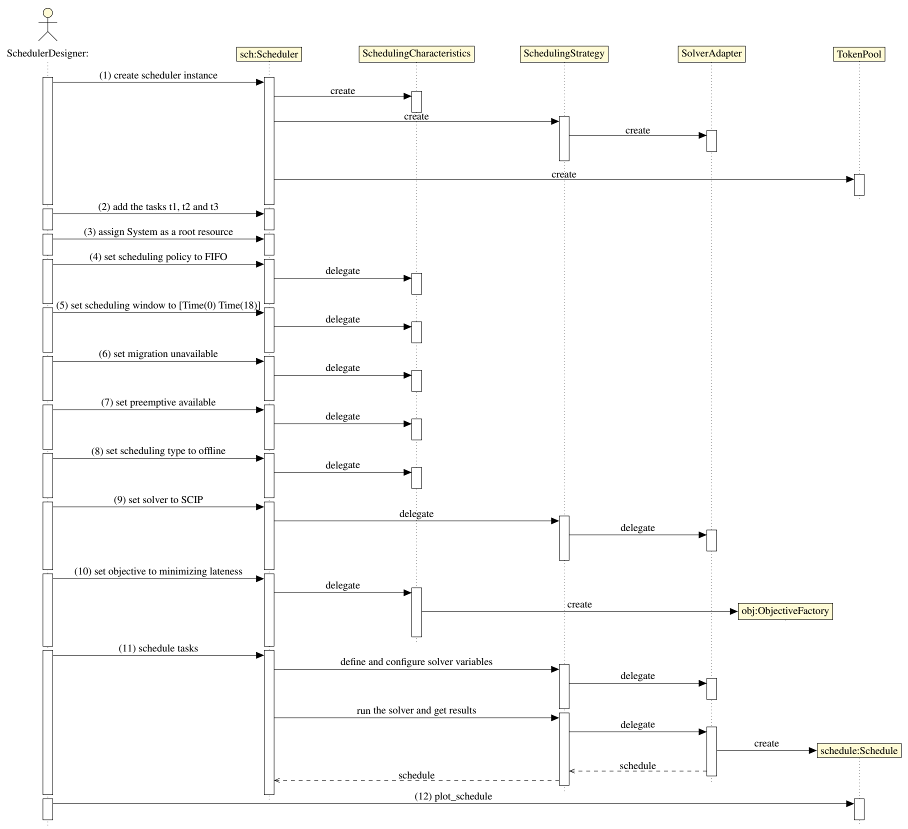

SchedulerDesigner:
sch:Scheduler
SchedulingCharacteristics
SchedulingStrategy
SolverAdapter
TokenPool
(1) create scheduler instance
create
create
create
create
(2) add the tasks t1, t2 and t3
(3) assign System as a root resource
(4) set scheduling policy to FIFO
delegate
(5) set scheduling window to [Time(0) Time(18)]
delegate
(6) set migration unavailable
delegate
(7) set preemptive available
delegate
(8) set scheduling type to offline
delegate
(9) set solver to SCIP
delegate
delegate
(10) set objective to minimizing lateness
delegate
create
obj:ObjectiveFactory
(11) schedule tasks
define and configure solver variables
delegate
run the solver and get results
delegate
create
schedule:Schedule
schedule
schedule
(12) plot_schedule

Fig. 6: Sequence diagram to create, configure the scheduler sch and getting the optimized schedule as a graph.

5.2.2 Instantiation of Tasks

Assume that we would like to create 3 tasks called t1, t2, and t3. The actor *Task Designer* represent the role who creates, instantiates and configures the tasks. In Figure 5, the call (1) is used to create an instance of the task t1. The calls (2), (3), and (4) are used to set the timing parameters of this task. The calls (5) and (6) are used to set the resource requirements of t1. Similarly, the calls (7) and (8), and (9) and (10) symbolize the creation and definition of the tasks t2 and t3. For brevity the details are not shown. The calls (11), (12), and (13) are used to illustrate possible definitions of dependencies between tasks. An interested reader should refer to the **FSF** website (previously called LFOS [1]) for the details.

5.2.3 Instantiation of the Scheduler

This part illustrates how a scheduler can be created at run-time based on the tasks and resources defined in the previous sections. The actor *Scheduler Designer* represents the role who creates, instantiates and configures the scheduler. In Figure 6, the call (1) symbolizes the initial creation operation of the necessary objects. The call (2) initializes the scheduler with the previously defined tasks. The call (3) is responsible for setting the resource tree as presented in Section 5.2.1. The calls (4) to (10) symbolize how the important parameters of the schedulers are set. Where necessary, the component scheduler delegates the calls to the responsible components. Finally, the call (11) starts the scheduling process. This call sets the parameters of the solver, instantiates it with the given type and starts the algorithm of the solver. The solver returns a schedule, which can be in turn utilized to execute the tasks accordingly. Eventually, the obtained schedule can be plotted. The call (12) represents such an action.

6 Case Studies

In Section 2, design and implementation of a scheduler framework which has a high-degree of *reusability* and *run-time evolvability* are defined as the two key research objectives. To obtain these quality attributes, a common practice today is to design an application framework. As a design method, we have adopted the method of *deriving the key abstractions of the framework from the corresponding theories* [4]. In application frameworks, to provide a high-degree of *reusability* in a given domain, the framework library must be expressive enough to implement the well-known examples of that domain. To this aim, to demonstrate *reusability* of **FSF**, this section presents a set of canonical examples from the scheduling domain which are instantiated from the framework. To demonstrate the quality attribute *run-time evolvability*, each example is extended with a set of evolution scenarios. A more detailed evaluation of **FSF** is given in Section 7.

6.1 Rate Monotonic Scheduling (RMS)

Rate Monotonic Scheduling (RMS) is a scheduling method deployed in real-time operating systems. Although in real-time systems tasks can be defined both as periodic and aperiodic tasks, in RMS only *periodic* tasks are assumed. The priority values of the instances of the tasks are *fixed* and determined with respect to inter-arrival time (period) of instances. The shorter period a task has, the more privileged it becomes. Unlike aperiodic tasks, the periodic ones have hard deadline requirements and these are equal to the beginning of the next request of the task. The scheduling process is preemptive. As a consequence, a task can never be in a waiting state for a less privileged task [32].

[1] `https://github.com/gorhan/LFOS/tree/master/LFOS`

6.1.1 Initial Requirement

Assume that the following RMS is desired, which is expressed using the notation 1 which is presented in Section 4:

$$1|pmtn, r_j|\sum_j w_j L_j. \tag{2}$$

In fact, an RMS is a "*a fixed-priority online scheduling problem for scheduling independent, preemptable, periodic tasks on a single processing unit aiming at minimizing the total weighted lateness objective*".

6.1.2 Implementation of the Initial Requirement

We assume that our taskset consists of four tasks, and a single resource named cpu. We specify the end of the scheduling window as the completion time of the latest second instance of a task in the taskset[2].

As for the required parameters, we have generated the timing attributes of the tasks randomly; the values are shown in Table 2.

Tasks	**Release Times**	**Execution Times**	**Period**	**Priorities**
τ_1	$r_{(\tau_1,0)} = 0$	$c_{(\tau_1,\text{cpu})} = 7$	$p_{\tau_1} = 29$	⋆
τ_2	$r_{(\tau_2,0)} = 3$	$c_{(\tau_2,\text{cpu})} = 3$	$p_{\tau_2} = 28$	⋆⋆
τ_3	$r_{(\tau_3,0)} = 2$	$c_{(\tau_3,\text{cpu})} = 2$	$p_{\tau_3} = 22$	⋆⋆⋆⋆
τ_4	$r_{(\tau_4,0)} = 1$	$c_{(\tau_4,\text{cpu})} = 4$	$p_{\tau_4} = 25$	⋆⋆⋆

Table 2: The randomly generated values of the timing attributes for the taskset. The number of stars in the column priority corresponds to the relative importance of the corresponding task in the taskset.

Based on this initial requirement, a scheduler is instantiated and executed. The result is displayed in Figure 7. Since τ_4 is more privileged than τ_1, the resource is reserved to τ_4 at $t = 1$. Again for the same reason, τ_3 and τ_2 take the permission of utilization of the resource cpu at $t = 2$ and $t = 31$, respectively.

6.1.3 Evolution of the requirement: The platform is extended with an additional CPU

To demonstrate *run-time evolvability* of the scheduler, we assume the following change in the requirements: The system is migrated to a new platform where 2 CPU's are utilized. Each CPU has the same characteristics as the initial one.

[2] An interested reader can refer to our repository: `https://github.com/gorhan/LFOS/blob/master/Tests/RMS.py`

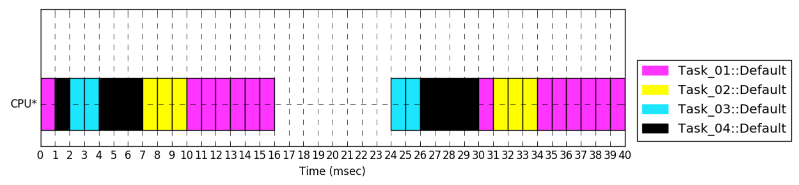


Fig. 7: A graphical output of the example RMS instantiated with the initial requirements.

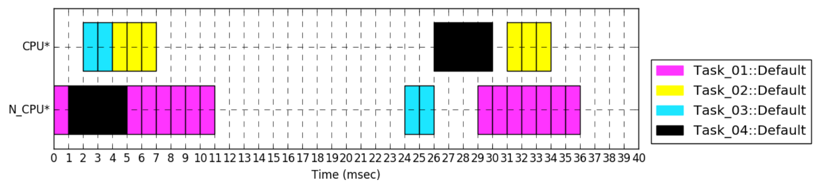


Fig. 8: A graphical output of the example RMS according to the new requirements.

The evolved requirement can be defined as follows:

$$P2|pmtn, r_j|\sum_j w_j L_j. \tag{3}$$

Since both resources are assumed to be identical, the execution times of the tasks do not change.

6.1.4 Implementation of the new requirement

We will now extend the implementation of the previously instantiated scheduler at run-time in 3 steps: (i) creating an additional resource; (ii) setting the resource properties, which are equal to the ones of the first resource; and (iii) introducing this new resource instance to the system.

After run-time evolution, a new scheduler is configured. The output of it is shown in Figure 8. Since there exists no available resource and the task τ_4 is ready to execute, the scheduler preempts the task τ_1 at 1. Due to the additional resource, the second instances of the tasks have completed their executions in 4 units of time earlier than the initial case.

The output of the evolved implementation is shown in Figure 8. Since there exists no available resource and the task τ_4 is ready to execute, the scheduler preempts the task τ_1 at 1. Due to the additional resource, the second instances of the tasks have completed their executions four units of time earlier than the initial case.

6.2 Multiple Resource Scheduling (MRS)

This example is defined to demonstrate the implementation of a complex scheduling problem involving multiple tasks and resources.

6.2.1 Initial Requirement

Assume that there are 2 tasks and 2 resources each with different characteristics. There are periodic and aperiodic tasks, which are referred to as *system-level* and *application-level* tasks, respectively. A system-level task has a higher priority than an application-level task. Unlike an application-level task, a system-level task cannot be suspended. It is also assumed that an application-level task requires a system-level task to complete. Therefore, aperiodic tasks *depend* on periodic tasks. The resources in this example are classified as active and passive resources and named as processing unit and memory, respectively. The active resources can process the tasks with different speeds by adjusting the exerted power. This is not possible with the passive resource. The scheduling process is defined as *offline*. It is also assumed that some tasks are preemptable. As a design choice, the tasks are prioritized with respect to their release times. Therefore, the scheduling policy is defined as Earliest-Release-Time-First.

Instances of a task are re-prioritized after the completion of each instance for the following 2 reasons: (1) the release time of each instance can be different; and (2) an instance with earliest release time compared to the other instances may not have the same release time characteristics in the next scheduling window with respect to its period.

Finally, the overall objective of the scheduler is defined as minimizing the consumed power on the resources while executing the tasks.

The requirement is expressed using Definition 1:

$$Q2|r_j, d_j, prec, pmtn, M_j, s_{jk}, batch|\sum_{i,j} PW(i,j), \tag{4}$$

where $PW(i,j)$ is the total exerted power of the resource i on running the task j.

6.2.2 Implementation of the Initial Requirement

The framework is configured in the following way[3]: There are two instances for each task; $\tau_{(1,1)}$ and $\tau_{(1,2)}$, and $\tau_{(2,1)}$ and $\tau_{(2,2)}$ are defined as instances of periodic tasks and aperiodic tasks, respectively. There are also two instances for each resource; cpu1 and cpu2, memory1 and memory2 are instances of active resources with single-unit capacities, and as passive resources with 512-unit capacities, respectively. In addition, in terms of power consumption, the active resources have two modes, half-scale (0.5) and full-scale (1.0). If a resource is running at half-scale mode, the execution time of any task on that resource becomes two times longer than its actual execution time.

In Table 3, the timing attributes of instances of tasks are shown. Since the attributes used for periods are redundant for aperiodic tasks they are shown as *NA* (not applicable).

Data dependency between tasks is shown in Figure 9. The numbers above the edges represents the *sequence dependent setup times* explained in [38]. The direction

[3] An interested reader can refer to our repository: `https://github.com/gorhan/LFOS/blob/master/Tests/MRSP.py`

Tasks	Release Times	Execution Times	Deadlines	Period
$\tau_{(1,1)}$	$r_{(\tau_{(1,1)},0)} = 0$	$c_{(\tau_{(1,1)},\text{cpu1})} = c_{(\tau_{(1,1)},\text{cpu2})} = 3$	$d_{(\tau_{(1,1)},0)} = 6$	$p_{\tau_{(1,1)}} = 6$
$\tau_{(1,2)}$	$r_{(\tau_{(1,2)},0)} = 2$	$c_{(\tau_{(1,2)},\text{cpu1})} = c_{(\tau_{(1,2)},\text{cpu2})} = 1$	$d_{(\tau_{(1,2)},0)} = 4$	$p_{\tau_{(1,2)}} = 4$
$\tau_{(2,1)}$	$r_{(\tau_{(2,1)},0)} = 3$	$c_{(\tau_{(2,1)},\text{cpu1})} = c_{(\tau_{(2,1)},\text{cpu2})} = 2$	$d_{(\tau_{(2,1)},0)} = 14$	$p_{\tau_{(2,1)}} = NA$
$\tau_{(2,2)}$	$r_{(\tau_{(2,2)},0)} = 8$	$c_{(\tau_{(2,2)},\text{cpu1})} = c_{(\tau_{(2,2)},\text{cpu2})} = 1$	$d_{(\tau_{(2,2)},0)} = 11$	$p_{\tau_{(2,2)}} = NA$

Table 3: The timing attributes of tasks. NA: Not Applicable

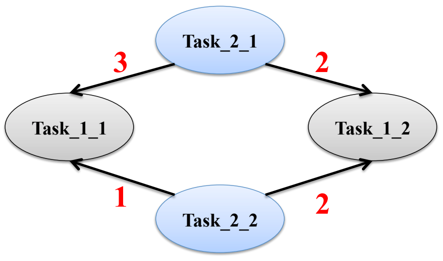


Fig. 9: The data dependency graph.

of an arrow indicates the dependency of tasks. The target tasks $\tau_{(1,1)}$ and $\tau_{(1,2)}$ depend on the source tasks $\tau_{(2,1)}$ and $\tau_{(2,2)}$. An instance of a task is eligible to execute at any time after at least one of the dependency relations is satisfied. We term this type of dependency as *OR-dependency*.

In addition, the instances of tasks $\tau_{(1,1)}$ and $\tau_{(2,1)}$ are instantiated with 650- and 140-unit capacities on the passive resources, respectively.

With these settings, the framework is instantiated and executed. The result is shown in Figure 10. As can be seen from the figure, to lower the power consumption, the scheduler utilizes the resource cpu2 at half-scale mode. Due to the dependency relation defined for $\tau_{(2,1)}$, this task cannot start immediately after its release time, and consequently its completion is deferred. The resource cpu1, therefore, has to operate at a full-scale mode at $t = [9, 11)$ so that the tasks $\tau_{(2,2)}$ and $\tau_{(1,2)}$ can be completed within their deadlines.

6.2.3 Run-time evolution of the requirement: change of the objective due to increasing task demand

To evaluate the run-time evolvability of the example, we introduce the following new requirement. Assume that the number of instances of tasks is becoming more than the underlying system can support. In this case, the overall objective of the scheduling process is changed to minimize the total weighted lateness. The justification of this change is to avoid the deadline misses. Since the evolution is not correlated with the scheduling parameters but the objective, the specification of the evolved scheduler is expressed as follows:

$$Q2|r_j, d_j, prec, pmtn, M_j, s_{jk}, batch|\sum_j w_j.L_j. \qquad (5)$$

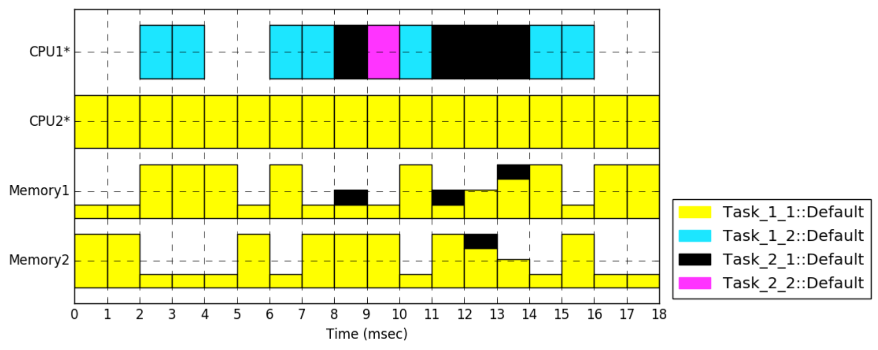


Fig. 10: A schedule aimed to minimize power consumption of the resources.

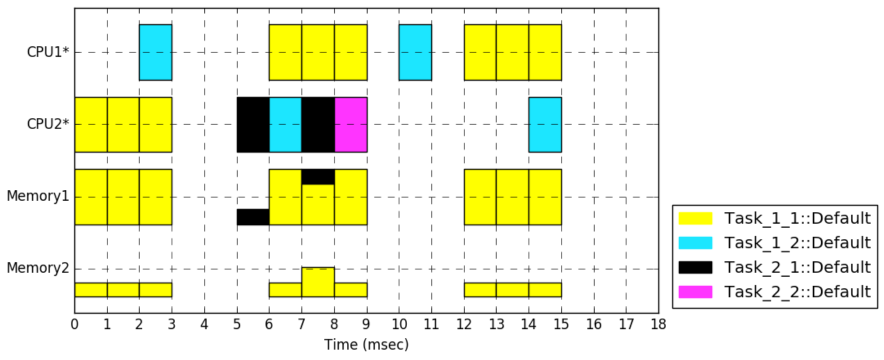


Fig. 11: A schedule of the example which aims at minimizing total weighted lateness.

To instantiate the scheduler at run-time with respect to the new requirements, we execute the following calls: (i) altering the objective to minimizing the total weighted lateness; and (ii) calling the method schedule on *Scheduler*.

6.2.4 Implementation of the new requirement

To realize the new requirement, the following modifications are carried out at run-time: (i) before each scheduling process, a conditional statement is added to check the number of instances. (ii) if there exist more than five instances of tasks, the overall objective is set to the total weighted lateness. After the re-instantiation, the schedule is computed. The result is shown in Figure 11. The scheduler chooses to run the resources at a full-scale mode and complete the tasks as soon as possible to realize the desired objective.

6.3 Job-shop Scheduling (JS)

In Job-shop scheduling, the resources are identical. They show differences based on the dependency relation between tasks and the machine eligibility of them. In JS, each job has a predetermined path of execution and it is not necessary for a job to visit each resource during its execution [6]. In addition, the execution orders of jobs on resources may be different. For instance, the job k_1 and k_2 may have the execution orders (3,2,4,1) and (1,4,2,3) on the resources. The eligible resources for the third activities of the jobs are denoted by $\mu_{(3,k_1)} = 4$ and $\mu_{(3,k_2)} = 2$.

In the following sub-section, we present a specific JS example.

6.3.1 Initial Requirements

Assume that there are three tasks with 3, 4 and 3 instances. The machine environment consists of four resources with single-unit capacity.

From the perspective of scheduling process, the priority is determined with respect to the execution duration of a job, which is constant. Due to this condition, the priorities are assigned to jobs once and remains constant unless it is modified. The scheduling algorithm is chosen to be *non-preemptive*. The objective is to minimize the makespan.

The initial requirement is expressed as follows:

$$J4|M_j, prec|C_{max}. \tag{6}$$

6.3.2 Implementation of the Initial Requirement

In our implementation, jobs and machines are modeled as instances of tasks and resources, respectively, We have adopted the parameters as defined in Example 7.1.1 in [38][4]. Each task is defined *aperiodic* and *non-preemptable*. To oblige each instance of a task to execute on a specific resource, the *machine eligibility* constraint is defined. In addition, we define the *dependency* relation among instances of tasks to specify the execution path of a task on different resources. Since the tasks have no release time or deadline constraints, a task may start to execute if the dependency constraint is satisfied. The eligible resources and execution times of the instances of tasks are given in Table 4.

The execution paths of jobs as defined in JS are expressed as dependency relations of tasks. These are shown in Figure 12.

The framework is instantiated and executed. The obtained schedule can be seen in Figure 13. The tasks $\tau_{(4,2)}$ and $\tau_{(3,1)}$ are ready to execute at time 18 on the resource Resource_03. Since the task with the longer execution time ($\tau_{(4,2)}$) has a higher priority than the others, it executes first.

[4] The details about the implementation can be found in our repository `https://github.com/gorhan/LFOS/blob/master/Tests/JSP.py`

Run-time Tasks	$\mu_{(i,j)}$	$c_{(i,j)}$
$\tau_{(1,1)}$	1	10
$\tau_{(2,1)}$	2	8
$\tau_{(3,1)}$	3	4
$\tau_{(1,2)}$	2	8
$\tau_{(2,2)}$	1	3
$\tau_{(3,2)}$	4	5
$\tau_{(4,2)}$	3	6
$\tau_{(1,3)}$	1	4
$\tau_{(2,3)}$	2	7
$\tau_{(3,3)}$	4	3

Table 4: The execution times of the instances of tasks.

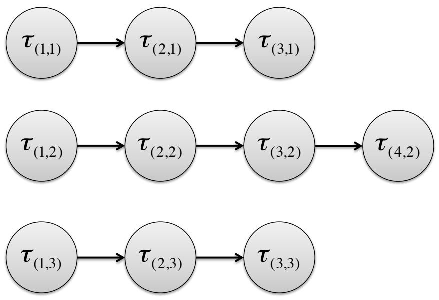

$$\begin{tikzcd}
\tau_{(1,1)} \arrow[r] & \tau_{(2,1)} \arrow[r] & \tau_{(3,1)} & \\
\tau_{(1,2)} \arrow[r] & \tau_{(2,2)} \arrow[r] & \tau_{(3,2)} \arrow[r] & \tau_{(4,2)} \\
\tau_{(1,3)} \arrow[r] & \tau_{(2,3)} \arrow[r] & \tau_{(3,3)} &
\end{tikzcd}$$

Fig. 12: The dependency graph of tasks.

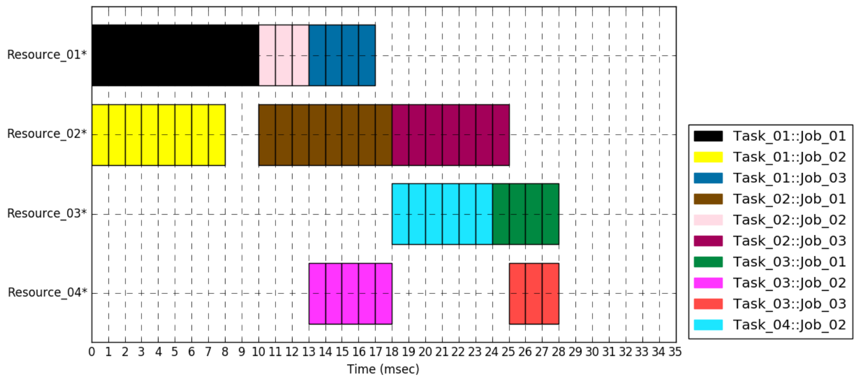


Fig. 13: An optimized schedule by minimizing the makespan.

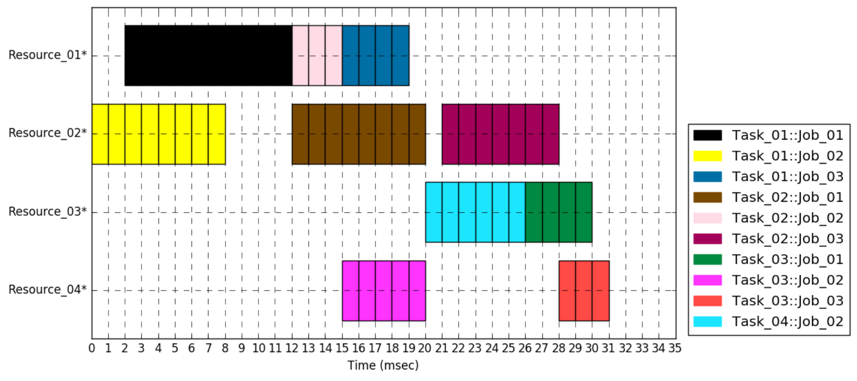


Fig. 14: A graphical representation of the schedule of the evolved JS example.

6.3.3 Run-time evolution of the requirement: adding release time constraint

As a run-time evolution of the previous example, now assume that some of the tasks cannot execute immediately after requesting a schedule. The new problem definition is accordingly expressed as follows:

$$J4|M_j, prec, r_j|C_{max}. \tag{7}$$

6.3.4 Implementation of the new requirement

To implement this evolution request, a new release time for the targeted instances of tasks must be set. To this aim, we define the release times of the tasks $\tau_{(1,1)}$ and $\tau_{(2,3)}$ as 2 and 21, respectively. To this aim, the corresponding method to set the release time of each instance of tasks is called.

As shown in Figure 14, due to the restriction on release times, the completion time of the latest task $\tau_{(3,3)}$ is delayed to 31.

6.4 Flow-shop Scheduling (FS)

Flow-shop Scheduling is commonly utilized in industrial production. It is a kind of shop problem where jobs have the same execution order on each machine, $\forall_{i,j}\ \mu_{(i,j)} = i$. Therefore, the execution path of jobs in the first machine have to be preserved on other resources, which is defined as *permutation* in [38]. Since the process of activities of a job in assembly line should not be altered, the scheduling algorithm is not preemptive. Furthermore, all the queues are postulated to operate under the policy of *First-In-First-Out (FIFO)*.

	j_1	j_2	j_3	j_4	j_5
$\tau_{(1,j)}$	5	5	3	6	3
$\tau_{(2,j)}$	4	4	2	4	4
$\tau_{(3,j)}$	4	4	3	4	1
$\tau_{(4,j)}$	3	6	3	2	5

Table 5: The execution times for the tasks.

6.4.1 Initial Requirements

In our framework, jobs and machines are represented as instances of tasks and resources, respectively. Each instance of a task is assigned to exactly one resource. This is specified as *machine eligibility*. Since each instance of a task has to execute on each instance of a resource once, tasks are defined as aperiodic. There exist dependency relations among tasks to ensure the order of executions on each instance of a resource. According to the requirements:

- The tasks are not *preemptable*.
- The resources are active and have single-unit capacities.
- The speed of resources are assumed to be constant.
- The scheduling process is defined as *offline*.
- The scheduling policy is determined as *FIFO*.
- The priority assignment is defined as *dynamic*.
- The overall objective is chosen as *minimizing* the *makespan*.

This requirement can be represented as follows:

$$F4|prec, prmu, M_j|C_{max}. \tag{8}$$

6.4.2 Implementation of the Initial Requirement

The example 6.1.1 in the book of Pinedo [38] is adopted in the implementation of our FJ[5]. There are five instances of tasks and four instances of resources. Since each job is supposed to execute on each resource, the taskset consists of 20 instances (5 *jobs* x 4 *resources*). The execution times of the tasks are shown in Table 5. The columns correspond to the jobs. The row $\tau_{(i,j)}$ correspond the execution time of the job j on the resource i.

The dependency relations of the tasks are shown in Figure 15. Here, the tasks (nodes) with two incoming edges have to wait until both dependency constraints are satisfied. We call this constraint as *AND-dependency*.

The example is implemented and executed. The obtained schedule is given in Figure 16. Since the objective is minimizing the makespan, it is not required to schedule the tasks as soon as possible. For this reason, some instances of tasks starts later than its earliest start time., such as $\tau_{(2,3)}$ and $\tau_{(3,3)}$. As it can be seen in the figure, the makespan of the schedule is 34.

[5] The implementation using our design environment and details can be found in our repository `https://github.com/gorhan/LFOS/blob/master/Tests/FSP.py`

$$\begin{tikzcd}
\tau_{(1,1)} \arrow[r] \arrow[d] & \tau_{(1,2)} \arrow[r] \arrow[d] & \tau_{(1,3)} \arrow[r] \arrow[d] & \tau_{(1,4)} \arrow[r] \arrow[d] & \tau_{(1,5)} \arrow[d] \\
\tau_{(2,1)} \arrow[r] \arrow[d] & \tau_{(2,2)} \arrow[r] \arrow[d] & \tau_{(2,3)} \arrow[r] \arrow[d] & \tau_{(2,4)} \arrow[r] \arrow[d] & \tau_{(2,5)} \arrow[d] \\
\tau_{(3,1)} \arrow[r] \arrow[d] & \tau_{(3,2)} \arrow[r] \arrow[d] & \tau_{(3,3)} \arrow[r] \arrow[d] & \tau_{(3,4)} \arrow[r] \arrow[d] & \tau_{(3,5)} \arrow[d] \\
\tau_{(4,1)} \arrow[r] & \tau_{(4,2)} \arrow[r] & \tau_{(4,3)} \arrow[r] & \tau_{(4,4)} \arrow[r] & \tau_{(4,5)}
\end{tikzcd}$$

Fig. 15: The dependency graph of the flow-shop scheduling example.

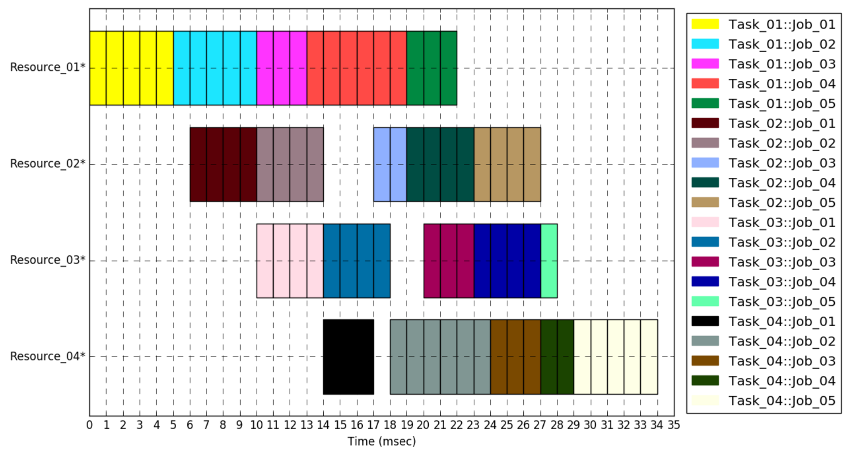


Fig. 16: A graphical representation of the schedule.

6.4.3 Evolution of the requirement: relaxation of dependencies

Assume that dependencies among the tasks have to be relaxed at run-time by removing the dependencies shown as vertical edges in Figure 15. Since there is no change in general scheduling attributes, the problem definition given in Equation 8 is still valid.

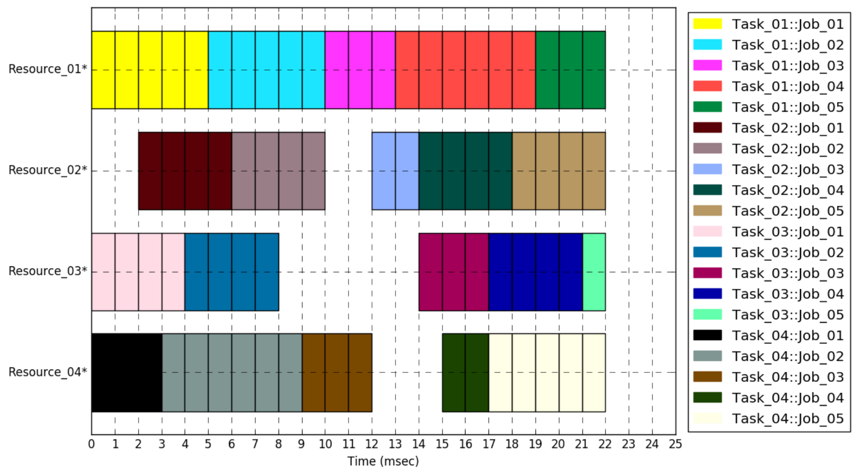


Fig. 17: A graphical representation of the output of the evolved FS example.

6.4.4 Implementation of the new requirement

There are two ways to implement the required evolution: Data dependency relations of each task are abandoned and the new ones are defined, or the undesired dependencies (shown as vertical edges) for each task is removed.

This evolution requirement is instantiated at run-time and executed. The resulting output is shown in Figure 17.

6.5 Open-shop Scheduling (OS)

In open-shop scheduling, like the previous shop examples, a job is dedicated to one machine. On the other hand, the dependencies among jobs are relaxed. A job can be freely allocated the corresponding machine when it is available. However, any two activities of a job cannot execute in parallel and therefore they cannot be active at the same time; an activity should finish its execution before another activity of the same job starts to execute. Like all shop problems, the jobs are not preemptable.

6.5.1 Initial Requirements

Assume that there are five jobs and three active resources. Each job is supposed to execute on each resource. Therefore, there exist 15 instances (5 *jobs* x 3 *resources*) according to our model. The details of the example are adopted from Example 8.4.1 in the book of Pinedo [38][6] and modified according to our model.

[6] The implementation using our design environment and details can be found in our repository `https://github.com/gorhan/LFOS/blob/master/Tests/OSP.py`

	j_1	j_2	j_3	j_4	j_5
$\tau_{(1,j)}$	1	2	2	2	3
$\tau_{(2,j)}$	3	1	2	2	1
$\tau_{(3,j)}$	2	1	1	2	1
r_j	1	1	3	3	3
d_j	11	9	8	9	11

Table 6: The execution times, release times and deadlines for the taskset.

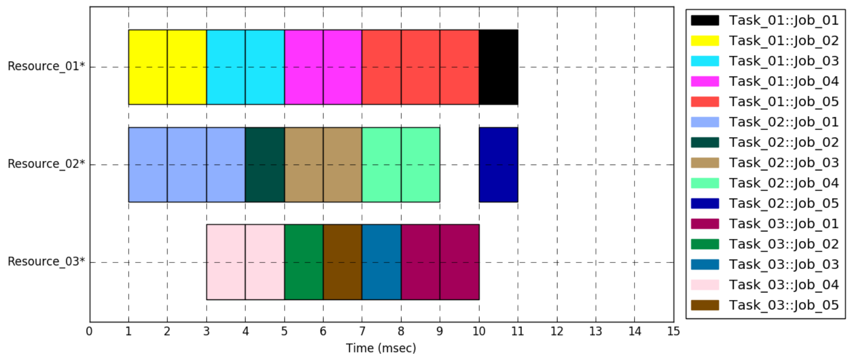


Fig. 18: A graphical representation of a schedule of the OS example.

The adapted problem definition of this example is as follows:

$$O3|r_j, d_j, M_j|L_{max}. \tag{9}$$

6.5.2 Implementation of the Initial Requirement

Similar to other shop examples, also in this example, jobs and machines are defined as tasks and resources, respectively. Each instance of a task is assigned to exactly one resource using the *machine eligibility* specification. Instances of tasks in the same subset as *mutually-exclusive*.

The execution times of the tasks on each resource, their release times and deadlines are shown in Table 6.

This example is instantiated and executed. The resulting schedule is displayed in Figure 18. There exists only one solution to this scheduling problem. The duration between release times and deadlines of the job task j_3 and j_4 is equal to their execution times. Therefore, there is no any other scheduling possibility. Since the execution time of the task $\tau_{(1,5)}$ is 3 and its deadline is 11, it is supposed to be scheduled immediately after the task $\tau_{(1,4)}$. Due to the non-preemptable tasks, although the task $\tau_{(1,1)}$ has the highest priority among the tasks, the task $\tau_{(1,2)}$ has to be scheduled at the time when its release time starts.

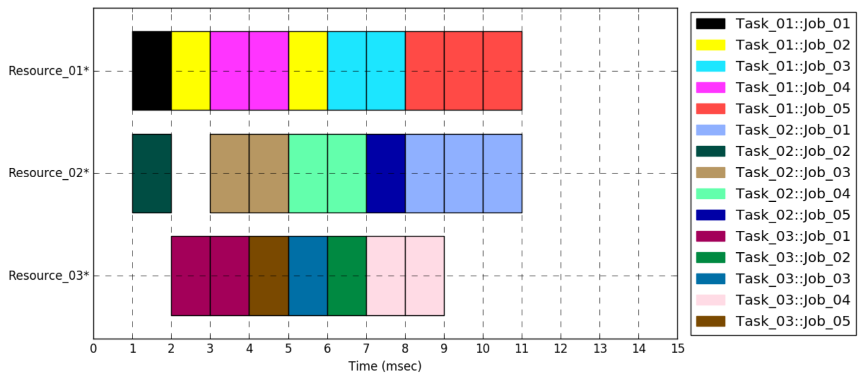


Fig. 19: A graphical representation of the output of the evolved example.

6.5.3 Evolution of the requirement: preemptable tasks

As an evolution step, now assume that the tasks are defined to be preemptable. This new requirement can be expressed as:

$$O3|r_j, d_j, M_j, pmtn|L_{max}. \tag{10}$$

6.5.4 Implementation of the new requirement

To implement this evolution, the taskset must be traversed and the tasks must be re-specified as preemptable. The previously defined schedule is modified at run-time accordingly and executed. The output schedule is shown in Figure 19. Here, the task $\tau_{(1,1)}$ is able to start immediately after its release time as the task $\tau_{(1,2)}$ is preempted by the task $\tau_{(1,2)}$.

7 Evaluation and Conclusions

In this section, the framework is evaluated against the objectives described in Section 2.

7.1 Our Assessment Method

The contribution of this paper is evaluated against the two required quality attributes *reusability* and *run-time evolvability*.

From the perspective of *reusability*, it is stated that to create a particular scheduling system, the code written from scratch must be much less than the code of the library that is reused. This definition refers to Lines-of-Code (LoC) metric [16]. There is much debate on the preciseness of the metric, because it may not accurately express the effort spent. The metric may be influenced from many factors such as the characteristics of the adopted programming language, the formatting styles used in coding etc. Therefore, the validity of LoC metric in a particular measurement context must be considered carefully. In addition, the definition of *reusability* within the context of application frameworks implies that the framework must be expressive enough to instantiate a large category of implementations in the domain of the framework.

Run-time evolvability is defined as an ease of modification of an existing scheduling software with respect to a new *meaningful* set of user requirements during the operational phase of the software. This implies that all relevant parameters of a system must be set by invoking operations on the corresponding objects. To validate this quality attribute, one can define evolution scenarios for each possible parameter change. One disadvantage of this evaluation is that there may be too many possible evolution scenarios. Nevertheless, within the domain of scheduling, the number of relevant attributes is limited. For example, in Section 2, run-time evolution support is required for five cases.

		α_1				α_2	
		Q	F	O	J	1	M
β	r_j	MRS		OS		RMS	MRS, OS
	$prec$	MRS	FS		JS		MRS, FS, JS
	$pmtn$	MRS		OS		RMS	MRS, OS
	$d_j = d$	MRS		OS			MRS, OS
	M_j	MRS		OS	JS		MRS, JS, OS
	s_{jk}	MRS					MRS
	$batch$	MRS					MRS
	$prmu$		FS				FS

Table 7: Domain coverage of examples used. Abbreviations represent the scheduling examples referred to in this paper.

7.1.1 Reusability of the Framework

To evaluate the expressivity, in Section 6, five canonical examples from the scheduling domain are presented.

It is argued that the scheduling domain can be represented using the Table 7[7]. The cells refer to the abbreviations of the example schedulers. The parameters in the rows of the table corresponds to the *scheduling characteristics* and are denoted by β; whereas the columns of the table are grouped into two categories, namely the *machine identifier* and the number of machines which are denoted by α_1 and α_2, respectively.

[7] http://www2.informatik.uni-osnabrueck.de/knust/class

Within these categories, the parameters *Q, F, O, J, 1, M* are explained in detail in Section 4.1. As it can be seen, in each column and row from top to bottom and left to right, respectively, at least one example resides. This illustrates that the examples cover at least one case of the parameters in the scheduling domain.

As previously shown in Figure 2, the framework can be divided into three subsystems: *Resources, Tasks* and *Scheduler*. As for implementation, Python and C/C++ are used. The third-party software that is adopted in the architecture are Numberjack [21], SCIP [19], MiniSat [45], MipWrapper [21], Mistral [15], Mistral2 [21], SatWrapper [21], Toulbar2 [12] and Walksat [41]. To implement the supporting functions, we have integrated the following third-party tools: Clafer [5], Alloy [27], Chocosolver [39] and MatplotLib [26].

The framework library including third-party software contains 18071 and 927904 LoC written in Python and C/C++ programming languages, respectively. The LOC of the supporting software is not included in this count.

	RMS	MRS	FS	JS	OS
Additional LoC (Python)	67	94	58	67	57

Table 8: The LoC for each example in Section 6.

In Table 8, the columns refer to the examples presented in this paper. The row indicates the LoC of the examples.

The LOC metric is not very precise and not all the code of the library is used in every example. Nevertheless, the amount of reuse of the library code is so much higher than the metrics shown in the row of Table 8 that the impreciseness in this context is considered negligible. Therefore, it is assumed the framework satisfies the *reusability* requirement.

7.1.2 Run-time Evolvability of the Framework

Requirement	**Scenario**	**Example**
A	Adding new resource	RMS
B	Changing the objective	MRS
C	Adding different release times to the tasks	JS
D	Removing some dependency relations	FS
E	Setting the tasks preemptable	OS

Table 9: The table that is used to evaluate *run-time evolvability* of the framework.

In Section 2, five cases for *run-time evolvability* are given. An evaluation of our framework with respect to this objective is given in Table 9. Here, the capital letters

	Parameter	Symbol	RMS	MRS	FS	JS	OS
BrE	**# Constraints**	n_c^i	4067	738	13980	2818	545
	Time for constraint definition (sec)	t_d^i	1.3961	0.46911	11.36172	5.1782	0.50092
	Time for solver execution (sec)	t_s^i	268.93	107.41	22342.83	274.71	7.71
	Overhead	$t_o^i = t_d^i/(t_d^i + t_s^i)$	0.0052	0.00435	0.00051	0.0185	0.06101
ArE	**# Constraints**	n_c^e	4629	738	12322	2417	754
	Time for constraint definition (sec)	t_d^e	1.91886	0.5057	12.22426	4.92931	0.5021
	Time for solver execution (sec)	t_s^e	251.65	107.18	14955.39	167.76	11.76
	Overhead	$t_o^e = t_d^e/(t_d^e + t_s^e)$	0.00757	0.0047	0.00082	0.02854	0.04095

Table 10: The table which is used to evaluate the time performance overhead of **FSF** with respect to a *bare solver* alternative.

in the first column correspond to the cases. The second column *Scenario* describes briefly the evolution scenarios presented for each example in Section 6. The last column lists the abbreviations of each corresponding example. It is clear from the table that these evolution scenarios can be realized at run-time. Therefore, it is assumed that the framework satisfies the *run-time evolvability* requirement.

7.1.3 Evolution of the Time-Performance Overhead

As demonstrated in the previous two subsections, **FSF** provides a high-degree of *reusability* and *run-time evolvability*. A legitimate question one may ask is *what is the cost of enhancing these quality attributes in terms of time performance?* To answer this question, the time performance of **FSF** is compared with *bare* solver-based solutions. Consider Table 10. Here, two tables are integrated into one. The upper and lower tables, which are named as **BrE** and **ArE** refer to the examples before and after evolution scenarios, respectively. The columns **Parameter** and **Symbol** refer to the relevant parameters for our evaluation. The columns **RMS, MRS, FS, JS**, and **OS** represent the measured parameters of the examples presented in Section 6. The row **# Constraints** indicates the number of constraints generated; these are to be considered by the solver. Obviously, the number of constraints gives an indication about the complexity of the problem and the required time-delay caused by the solver. The actual time performance of the solver also depends on the nature of the constraints and how they are related to each other. The parameter **Time for constraint definition** refers to the time spent for the generation and transformation of constraints realized by **FSF**. The parameter **Time for solver execution** refers to the time required by the *bare solver*. In this case, it is assumed that no time is spent in the preparation of the constraints since they are readily expressed in the specification language of the solver. A ratio of these two parameters is defined as **Overhead**.

As can be seen from the table, **Overhead** varies between 0.00051 and 0.06101. It is clear that the execution time caused by **FSF** is comparably much less than the execution time of the solver adopted.

7.2 Conclusions

In this paper, *reusability* and *run-time evolvability* are defined as the two key requirements of an object-oriented framework aimed at creating scheduling software. To this aim, a framework called **FSF** has been implemented. With the help of canonical examples, it is shown that **FSF** satisfies the *reusability* requirement. The *run-time evolvability* of the framework is demonstrated with a set of evolution scenarios. To the best of our knowledge, **FSF** is the first framework that provides a scheduling design framework with these quality attributes.

Acknowledgements This work has been supported by Aselsan A.Ş. generously.

References

1. Django project. http://www.djangoproject.com/
2. Ahmad, W., d. Groote, R., Hölzenspies, P.K.F., Stoelinga, M., v. d. Pol, J.: Resource-constrained optimal scheduling of synchronous dataflow graphs via timed automata. In: 2014 14th International Conference on Application of Concurrency to System Design, pp. 72–81 (2014)
3. Aksit, M.: Software Architectures and Component Technology. The Springer International Series in Engineering and Computer Science. Springer Verlag (2002)
4. Aksit, M., Tekinerdogan, B., Marcelloni, F., Bergmans, L.: Deriving Object-Oriented Frameworks from Domain Knowledge, pp. 169–198. John Wiley & Sons (1999)
5. Antkiewicz, M., Bąk, K., Murashkin, A., Olaechea, R., Liang, J., Czarnecki, K.: Clafer tools for product line engineering. In: Software Product Line Conference. Tokyo, Japan (2013). Accepted for publication.
6. Baptiste, P., Le Pape, C., Nuijten, W.: Constraint-based Scheduling: Applying Constraint Programming to Scheduling Problems. Kluwer Academic, Springer (2001)
7. Becker, C., Scholl, A.: A survey on problems and methods in generalized assembly line balancing. European Journal of Operational Research **168**(3), 694 – 715 (2006). Balancing Assembly and Transfer lines
8. Brucker, P.: Scheduling Algorithms, 5th edn. Springer-Verlag Berlin Heidelberg (2007)
9. Burns, A.: Scheduling hard real-time systems: a review. Software Engineering Journal **6**(3), 116–128 (1991)
10. Buttazzo, G.C.: Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series). Springer US, Santa Clara, CA, USA (2011)
11. Chen, J.J., Kuo, T.W.: Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In: 2005 International Conference on Parallel Processing (ICPP'05), pp. 13–20 (2005)
12. Cooper, M.C., de Givry, S., Sanchez, M., Schiex, T., Zytnicki, M., Werner, T.: Soft arc consistency revisited. Artif. Intell. **174**(7-8), 449–478 (2010)
13. Davis, R.I., Burns, A.: A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35:1–35:44 (2011)
14. Desrosiers, J., Dumas, Y., Solomon, M.M., Soumis, F.: Chapter 2 time constrained routing and scheduling. In: Network Routing, *Handbooks in Operations Research and Management Science*, vol. 8, pp. 35 – 139. Elsevier (1995)
15. Dillig, I., Dillig, T., Aiken, A.: Cuts from Proofs: A Complete and Practical Technique for Solving Linear Inequalities over Integers, pp. 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
16. Frakes, W., Terry, C.: Software reuse: Metrics and models. ACM Comput. Surv. **28**(2), 415–435 (1996). DOI 10.1145/234528.234531
17. Frakes, W.B., Kang, K.: Software reuse research: Status and future. IEEE Trans. Softw. Eng. **31**(7), 529–536 (2005)
18. Fromherz, M.P.: Constraint-based scheduling. In: American Control Conference, 2001. Proceedings of the 2001, vol. 4, pp. 3231–3244. IEEE (2001)
19. Gamrath, G., Fischer, T., Gally, T., Gleixner, A.M., Hendel, G., Koch, T., Maher, S.J., Miltenberger, M., Müller, B., Pfetsch, M.E., Puchert, C., Rehfeldt, D., Schenker, S., Schwarz, R., Serrano, F., Shinano, Y., Vigerske, S., Weninger, D., Winkler, M., Witt, J.T., Witzig, J.: The scip optimization suite 3.2. Tech. Rep. 15-60, ZIB, Takustr.7, 14195 Berlin (2016)

20. Graham, R., Lawler, E., Lenstra, J., Kan, A.: Optimization and approximation in deterministic sequencing and scheduling: a survey. Annals of Discrete Mathematics **5**, 287 – 326 (1979). Discrete Optimization II
21. Hebrard, E., O'Mahony, E., O'Sullivan, B.: Constraint Programming and Combinatorial Optimisation in Numberjack, pp. 181–185. Springer Berlin Heidelberg (2010)
22. Heinz, S., Ku, W.Y., Beck, J.C.: Recent improvements using constraint integer programming for resource allocation and scheduling. In: International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems, pp. 12–27. Springer (2013)
23. Hochwald, B.M., Marzetta, T.L., Tarokh, V.: Multiple-antenna channel hardening and its implications for rate feedback and scheduling. IEEE Transactions on Information Theory **50**(9), 1893–1909 (2004)
24. Holenderski, M., Bril, R.J., Lukkien, J.J.: Parallel-task scheduling on multiple resources. In: 2012 24th Euromicro Conference on Real-Time Systems, pp. 233–244 (2012)
25. Hooker, J.N.: Planning and scheduling by logic-based benders decomposition. Oper. Res. **55**(3), 588–602 (2007)
26. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing In Science & Engineering **9**(3), 90–95 (2007)
27. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press (2012)
28. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of object-oriented programming **1**(2), 22–35 (1988)
29. Kim, J.H., Legay, A., Traonouez, L.M., Acher, M., Kang, S.: A formal modeling and analysis framework for software product line of preemptive real-time systems. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, SAC '16, pp. 1562–1565. ACM, New York, NY, USA (2016)
30. Kolisch, R., Hartmann, S.: Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis, pp. 147–178. Springer US, Boston, MA (1999)
31. Linden, F.J.v.d., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer-Verlag New York, Inc., Secaucus, NJ, USA (2007)
32. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973)
33. Matic, S., Goraczko, M., Liu, J., Lymberopoulos, D., Priyantha, B., Zhao, F.: Resource modeling and scheduling for extensible embedded platforms. Tech. rep., MSR-TR-2006-176 (2006)
34. Mili, H., Mili, F., Mili, A.: Reusing software: issues and research directions. IEEE Transactions on Software Engineering **21**(6), 528–562 (1995)
35. Nikovski, D., Brand, M.: Decision-theoretic group elevator scheduling. In: ICAPS, vol. 3, pp. 9–13 (2003)
36. Orhan, G., Akşit, M., Rensink, A.: A formal product-line engineering approach for schedulers. In: 22nd International Conference on Emerging Trends and Technologies in Convergence Solutions, pp. 15–30. The Society for Design and Process Science (SDPS) (2017)
37. Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. SIGOPS Oper. Syst. Rev. **35**(5), 89–102 (2001)
38. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 3rd edn. Springer Publishing Company, Incorporated (2008)
39. Prud'homme, C., Fages, J.G., Lorca, X.: Choco Documentation. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2016). URL `http://www.choco-solver.org`
40. Saraf, A.P., Slater, G.L.: An efficient combinatorial optimization algorithm for optimal scheduling of aircraft arrivals at congested airports. In: 2006 IEEE Aerospace Conference, pp. 11 pp.– (2006)
41. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. In: DIMACS SERIES IN DISCRETE MATHEMATICS AND THEORETICAL COMPUTER SCIENCE, pp. 521–532 (1995)
42. Sha, L., Abdelzaher, T., årzén, K.E., Cervin, A., Baker, T., Burns, A., Buttazzo, G., Caccamo, M., Lehoczky, J., Mok, A.K.: Real time scheduling theory: A historical perspective. Real-Time Systems **28**(2), 101–155 (2004)
43. Silberschatz, A., Galvin, P.B., Gagne, G., Silberschatz, A.: Operating system concepts, vol. 4. Addison-Wesley Reading (1998)
44. Sommerville, I.: Software Engineering, 10th edn. Pearson (2015)
45. SÃűrensson, N., Een, N.: Minisat v1.13 - a sat solver with conflict-clause minimization. 2005. sat-2005 poster. 1 perhaps under a generous notion of âĂIJpart-timeâĂİ, but still concurrently taking a statistics course and leading a normal life. Tech. rep. (2002)

46. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem & overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. **7**(3), 36:1–36:53 (2008)
47. Zhao, W., Ramamritham, K., Stankovic, J.A.: Preemptive scheduling under time and resource constraints. IEEE Transactions on Computers **C-36**(8), 949–960 (1987)