A Feature Model and Development Approach for Schedulers

Kardelen Hatun
TRESE Group
University of Twente
Enschede, Netherlands
hatunk@ewi.utwente.nl

Christoph Bockisch
TRESE Group
University of Twente
Enschede, Netherlands
c.m.bockisch@ewi.utwente.nl

Hasan Sözer
TRESE Group
University of Twente
Enschede, Netherlands
sozerh@ewi.utwente.nl

Mehmet Aksit
TRESE Group
University of Twente
Enschede, Netherlands
aksit@ewi.utwente.nl

ABSTRACT
Schedulers decide when to execute what in a system. They often work in constrained environments, where these decisions have high impact on performance. Since schedulers should be designed according to a system’s needs, it is imperative that scheduling requirements are well defined. Building a scheduler that satisfies these requirements is not a trivial task. In this position paper we present our initial work and ideas on a domain-specific framework for scheduling. We introduce our feature model of the problem space of scheduling and explain how it fits within a framework. Furthermore we present our ideas on how to make the scheduler framework modular by using an aspect oriented approach.

Categories and Subject Descriptors
D.2.13 [Reusable Software]: Domain Engineering; D.2.11 [Software Architectures]: Domain Specific Architectures

General Terms
Design

Keywords
scheduling, feature modeling, aspect oriented software, design framework

1. INTRODUCTION
A scheduler is a vital system component, which has strong influence on the efficiency of a system. Every system, that has to perform multiple tasks under a set of constraints, has a scheduler; may it be a couple of lines of code or a system within a system. The brain of a scheduler is a scheduling algorithm describing a strategy for handling a system’s scheduling needs. There are as many algorithms as there are needs, hence there is a substantial amount of literature on scheduling algorithms.

The role of scheduling in a system is generally defined as allocation of resources to jobs over a period of time while optimizing one or more objectives [6]. This definition points out three main actors in a scheduler: jobs, resources and objectives. In practice these abstract concepts have many different versions which are constructed according to the information given in scheduling requirements. For example, in a system where deadlines are important, one should make the deadline property of jobs explicit. Similarly priorities of jobs must be explicit if some jobs are more urgent than others.

As a result of this versatility, most systems have their own constructs and implementation of a scheduler, often built without reusability and flexibility in mind. The resulting code is usually scattered and tangled. Moreover integrating a scheduler into a system is a complex task, which requires knowing the details of the system.

In this position paper we introduce our feature model which was created to understand scheduling problems in depth and get familiar with scheduling-related features. We also propose a domain-specific approach to scheduler design, where we present our ideas on how to map a problem to the solution space and how we can use aspect-oriented approaches to our benefit. Our approach is a set of steps embedded into a framework, which takes the user from initial requirements to the final scheduler product in a structured manner.

In the following sections we go into the details of our approach. We start by stating our motivation then introduce our feature model and shortly explain the concepts included in it.

2. MOTIVATION
Operating systems, workflow engines and embedded systems are well known examples of scheduling intensive systems. In these systems the quality of the scheduler affects overall performance greatly. For each of these systems re-
Feature modeling is a domain analysis method for determining core assets and reusable components for a line of software products. It is a concise way of expressing commonalities and variabilities in a domain. The notation of our feature model presented in this paper can be seen in Table 1.

Table 1: Feature Model Notation

<table>
<thead>
<tr>
<th>Optional Feature</th>
<th>Mandatory Feature</th>
<th>Alternative</th>
</tr>
</thead>
</table>

Figure 1: TIME-RELATED Features of JOB

TIME RELATED features of a JOB can be seen in Figure 1. PROCESSING TIME and PERIOD are mandatory features. Processing time is the duration which a job needs to be completed. Also jobs must be defined as either PERIODIC or APERIODIC. Aperiodic jobs are required to have release dates if jobs are being scheduled in a dynamic environment. Another important TIME-RELATED feature of JOB is DEADLINE, which marks the point in time a job has to finish latest. It is an optional feature since having a deadline is not mandatory in every system. A DEADLINE can be HARD meaning not meeting this deadline will cause an error or SOFT meaning deadline is not a strict constraint. In safety critical systems hard deadlines are common, whereas in operating systems jobs usually do not have any deadlines. We assume that HARD DEADLINES are final values. Finally we have the DELAY feature of a JOB, which specifies if a job can be delayed or not. If a job is delayable and it has a deadline, then that deadline cannot be fixed.

A job may have dependencies; it may depend on another job or it may depend on a condition. This is included in our feature model under the DEPENDENCY feature (Figure 2). We have defined two kinds of dependencies: INTER-JOB DEPENDENCY and CONDITION DEPENDENCY. The most commonly found INTER-JOB DEPENDENCY is START-TO-FINISH, where a job cannot start until another one finishes. A job can also depend on a condition. We have included RESOURCE CONDITION and TIME CONDITION in our model. An example for RESOURCE CONDITION can be an extra power source.
which is available sporadically and a job which requires this resource must be scheduled while the power source is on. Similarly a job can have a time dependency, which constrains it to be scheduled after a certain point in time.

The last two important features of job are granularity and importance. The granularity feature specifies if a job can be interrupted or not (preemption) or if it can be decomposed into multiple jobs (atomicity). The importance feature groups priority and weight (Figure 3).

Resources have fewer features than jobs. We have included three crucial resource characteristics: if a resource allows multiple access or single access, if it is limited or unlimited and finally if it is a general purpose resource or a dedicated resource which can only be allocated by certain types of jobs (Figure 4).

The objective feature is divided into two categories, time objectives and resource objectives. Under time objective we have listed widely used objective functions. If the user chooses to have total lateness as the objective function, then job has to have the deadline property. Resource objectives are currently defined as maximize utilization or minimize utilization. These are abstract objectives and will be detailed in the future. The objective feature tree is illustrated in Figure 5.

We found relevant to include a fourth feature which defines if the type of scheduling is static or dynamic. In a static scheduling it is assumed that all information is known prior to scheduling, in dynamic scheduling this information becomes available as the system continues running. This feature is needed since there are different logical relationships between features when a scheduling is static or when it is dynamic. Some of the logical relationships in the feature model are shown in (1).

\[
\text{total lateness} \Rightarrow \text{deadline} \\
\text{reassignable deadline} \Leftrightarrow \neg \text{non delayable} \\
\text{dynamic} \wedge \text{aperiodic} \Rightarrow \text{release date} \\
\text{preemption} \Rightarrow \text{priority} \\
\text{preemption} \wedge \text{deadline} \Rightarrow \text{reassignable deadline} \quad (1)
\]

4. APPROACH

Our approach consists of three main steps. The first step is to create a product instance of the feature model of the scheduling problem space. This instance will represent what kind of properties are present in the application the scheduler is being designed for. These properties also act as design-time requirements. The corresponding scheduler model must include components to handle the constraints and needs these properties come with. The second step is specifying the behavior of the abstract solution-space components, which are generated using the information given in the first step. In the third step, the scheduler model is bound to the system by using aspects. Scheduler-relevant events are detected by pointcuts and corresponding scheduling advices are executed (Figure 7). The following subsections give details on each of these steps.

4.1 Product Instantiation

In section 3 we have introduced a feature model depicting possible properties of scheduler-related components. A product of this model is created via choosing a concrete type for every property. The combination of these choices forms a single product, which matches a system specification. Figure 6 illustrates partial specification of a job. Here we have specified a job concept which has an arbitrary processing time, a fixed soft deadline, is aperiodic and is non-delayable. Such a product represents a model for a job. Assigning values to these properties will result in a unique job object. It is assumed that all of these properties will be relevant for the scheduler and the next step will be built on the choices that are made in the product-creation step.

4.2 Mapping to Solution Space

The product built in the first step also represents design-time requirements. Every problem-space choice corresponds to a design decision in the solution space. The relationship between the two spaces is derived using the information from the domain analysis.
The job in Figure 6 has a fixed soft deadline. One design decision that should be made is specifying a 'deadline miss policy'. To define such a policy, one can extend or use well known deadline miss policies such as giving an error when a deadline is missed, adding the cost of missing a deadline, tolerance/handling or doing nothing.

Another important part of this step is defining the events the scheduler is sensitive to. These events are also highly related to the specification made in the first step. An event is relevant when an object relevant to the scheduler is the subject of the event or is affected by this event. Some relevant events for the job shown in Figure 6 can be a job finishing on time and a job missing its deadline. The design decisions also include what kind of responses should be given when such events occur.

We are in the process of devising the possible design decisions which correspond to the features in our problem space. The composition of the chosen components will form the final scheduling policy. This way we aim to have modular schedulers with reusable components and robust composition mechanisms.

4.3 Use of Aspects

Using aspect-oriented benefits our approach in two ways; it solves the tangling problem in schedulers and it provides a natural interface for integrating the scheduler into a system.

The aspects are responsible for detecting scheduler-related events and, according to the type of events (shown as triangles), invoke the method responsible (shown as circles) for handling them (depicted in Figure 7). This enables decoupling of event detection and event response tasks. Since event detection tangles with the main functionality of a scheduler and is crosscutting, we put that functionality in aspects.

The scheduler is bound to a system via pointcut declarations. The events determined in the second step should be connected to the events in the system. This way pointcuts detect the relevant events in the system. For example an event is defined to be a state change of a resource from IDLE to BUSY. The pointcut will look for method calls which cause this kind of a state change. When such a method call occurs it is seen as a relevant event and the corresponding event response is invoked.

The responses to the same event may be different, since the state of the components involved in this decision also affects the response. To illustrate this fact we can give the following simple example. Assume that a new job is released that needs to allocate a general-purpose resource. There can be two responses to this event, if the resource is idle we can schedule the job or if the resource is busy we can delay this job until the resource becomes available. Here the response of the same job release event is affected by the state of the resource.

In [1] the authors propose a declarative yet expressive way of specifying events. This work can act as a basis for implementing our approach. Since systems get complex we will require such expressive pointcut declarations to make the connection between the scheduler and the system.

5. RELATED WORK

Domain analysis is one of the crucial parts of this study, hence we spent a substantial amount of time on studying books ([2], [6], [8]) and other sources ([7], [4]) for commonly known scheduling problems.

Few literature is available on scheduling-specific design frameworks. One powerful example is BOSSA DSL [5], which is developed to provide domain-specific constructs for kernel schedulers in operating systems. The study aims to simplify the scheduler development process “to the point that kernel expertise is not required to add a new scheduler to an existing kernel”.

In [9] an aspect-oriented synchronization library is introduced. Similar to our idea of modularizing scheduling and attaching it to the system with an aspect interface, the authors use aspects to attach different synchronization mechanisms to the same code structure to allow customizable design.

Coyler et al. introduce the notion of orthogonal and weakly orthogonal aspects in [3]. They demonstrate these principles on the software product line case, which is also interesting for our approach.

6. FUTURE WORK

We have presented our preliminary work and ideas on a domain-specific scheduling framework. Our future plans include modeling the solution space and finding the semantic relationships between solution space features for meaningful composition. We will also investigate event types in various systems to understand common events and which of them are relevant to schedulers. We will examine aspect languages to see if they are expressive enough to be used as a binding mechanism between the system and the scheduler. For the implementation of the framework we will choose the most appropriate aspect-oriented approach, respectively refine an existing approach to meet our requirements.
Acknowledgements

This work has been carried out as part of the OCTOPUS project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute program.

7. REFERENCES


