Scheduling Time-bounded Dynamic Software Adaptation

Serena Fritsch, Aline Senart
Distributed Systems Group
Trinity College Dublin
Dublin, Ireland
fritschs, senarta@cs.tcd.ie

Douglas C. Schmidt
Institute for Software Integrated Systems (ISIS)
Vanderbilt University
Terrace Place, USA
d.schmidt@vanderbilt.edu

Siobhán Clarke
Distributed Systems Group
Trinity College Dublin
Dublin, Ireland
sclarke@cs.tcd.ie

ABSTRACT

Dynamic adaptation of component-based software is playing an increasing role for applications in domains such as automotive, avionics or robotic systems. It includes the integration of new, previously unanticipated features and the update of existing features without requiring any system downtime. Due to the mobile nature of many of the target applications, adaptations must be executed within time bounds. Inconsistent or inaccurate behaviour may result from an adaptation that does not complete within specified time constraints. A service provider must therefore take time constraints into account when scheduling adaptation actions. In this paper, we propose an algorithm for the time-bounded scheduling of adaptation actions. We include evaluation results of this scheduling approach and present a success metric that helps in comparing our approach with other scheduling algorithms.

1. INTRODUCTION

Next-generation embedded systems in domains such as automotive, avionics or robotics, need to adapt swiftly to changing environmental conditions to better reflect their current situation [12]. We have previously shown, that these systems require varying levels of support for dynamic adaptation: from a very limited support for fault tolerance in safety-critical systems, to dynamic adaptation of resource allocation in avionic systems, to content adaptation in multimedia based systems, and to the actual runtime adaptation of software itself in component-based driver information systems [10].

The adaptation of software is often referred to as compositional adaptation and allows the dynamic integration and exchange of features and resulting application behaviour at runtime [15]. Previously unanticipated or updated features can be integrated into running systems in response to external triggers, making these systems more flexible and maintainable [16].

Key research challenges. Due to the mobile nature of many of these systems, software adaptations must often be time-bounded. For example, a plugin to the driver information system can only be downloaded when the vehicle is in the neighbourhood of a server providing this plugin. However, because the vehicle is moving, this download and integration needs to be executed before the vehicle is out of communication range from the server. Likewise, adaptations should minimise software update time to ensure that software applications and data are fully integrated into vehicles before they are used [10]. An adaptation that is out of its time bound may result in inconsistent or inaccurate behaviour. For example, the installation of an ice warning module needs to be performed while the weather conditions still is valid for this module.

Software adaptations can only be triggered when a server that provides features is in close range. As a server may potentially handle thousands of requests concurrently and has the knowledge of all the characteristics of the features it provides, it can decide which adaptation actions to execute based on the current configuration of the requester. Example adaptation actions are the installation or upgrade of a feature. This decision process may be affected by time constraints because the limited available time may lead to execution of only a few selected adaptations.

Additionally, adaptations can be constrained by other factors, like the available memory space of a consumer. For example, it might not be possible to download and integrate all necessary plugins because of a client's memory limitations.

Summary of research contributions. This paper proposes an approach for the constraint-based scheduling of adaptation actions that could be executed by the service provider. The algorithm takes time constraints into account, by maximising the amount of features that can be adapted within a given time on a software platform. Additionally, the algorithm considers constraints such as limited memory and importance of features.

The algorithm assumes features to be ordered after some criteria in an ordered list. The ordered list of features is obtained by applying weighted functions on the features' properties, e.g., priority and size. This ordering is not statically defined but can be adapted to better reflect the current requester's constraints.

We provide empirical results that validate the algorithm by means of a representative case study. Additionally, we propose a success metric function that helps in comparing our algorithm with existing approaches.

Paper organisation. The remainder of this paper is organised as follows: Section 2 motivates our work with an

ICSE 2008 Leipzig, Germany

example taken from the automotive domain; Section 3 describes the algorithm in more detail and also introduces our system model; Section 4 discusses some of the design challenges and their solutions; Section 5 shows some (preliminary) evaluation results of our approach; Section 6 summarises related work on scheduling and code distribution; Section 7 concludes this paper.

2. MOTIVATING SCENARIO

Our motivating scenario for the need of time-bounded scheduling of adaptation actions is taken from next-generation intelligent lane reservation systems, so-called managed highways and is illustrated in Figure 1. Managed highways aim at a reduction of traffic congestion and a control of traffic flow, e.g., allowing emergency vehicles to arrive safely and faster at accidents [17]. One way to schedule and enforce vehicle QoS on a managed highway is to allow drivers to reserve lanes "slots".

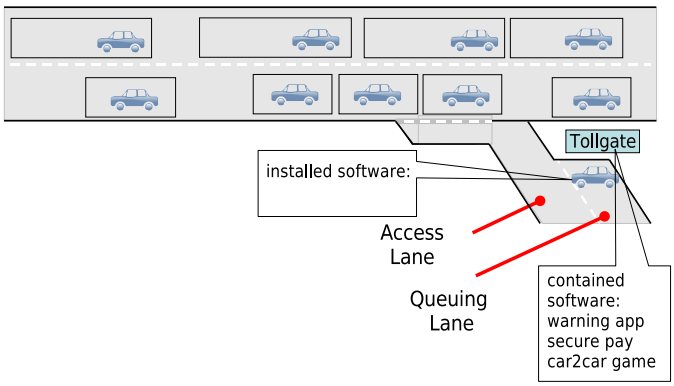

Figure 1: Managed Highway Scenario

To ensure proper admission control, vehicles wait in a queueing lane for their reserved slot to become available before entering the highway. A highway entrance assistance system (*e.g.*, a tollgate) uses short-range communication and relays between queued vehicles to ensure the vehicles have proper software versions and necessary hardware before allowing them to enter the highway. Example software includes warning applications, secure payment and communication algorithms, as well as infotainment applications, such as hotel and restaurant finder or car-to-car gaming applications.

The scenario motivates the need for various software adaptations. Adaptations comprise the integration of software, previously not installed at the vehicle, as well as the upgrade of software that is available with a newer version on the tollgate. The downgrade of a software or the complete deinstallation due to memory limitations of the vehicle software platform or expirations of licences are other examples of possible software adaptations [6].

A decision process located at the tollgate determines the actual adaptations and affected software based on a vehicle's current software configuration. After the download of the relevant software to the vehicle, the actual adaptation then takes place by executing the adaptations on the vehicle's software platform. The overall adaption process itself is time-bounded since the decision which adaptations to execute and the download and adaptation of the software itself must be executed before the vehicle can enter the highway. Additionally, the decision process can be influenced by (1) the available memory on a vehicle platform, (2) software interdependencies and (3) versioning of a software.

3. CONSTRAINT-BASED ADAPTIVE SCHEDULING

The purpose of this section is two-fold. The first part introduces our system model and describes common terminology. Additionally, different time bounds are explained in more detail. The second part explains in more depth our constraint-based algorithm that schedules adaptation actions within a bounded time. Our algorithm supports additional constraints that are input parameters at the start of the overall adaptation process.

3.1 Software Adaptation

Software adaptation is traditionally performed on systems composed out of binary software components with specified interfaces and explicit dependencies called modules [18]. Dynamic adaptation actions include (1) integration of code modules, (2) deintegration of code modules and (3) exchange with existing code modules for up- or downgrades.

In our approach, we assume the presence of a common underlying software platform on which code modules and their dynamic adaptations can be executed [8]. Code modules have additional non-functional properties, like priority, version number, dependencies on other modules and timeliness properties, that are provided by a module's developer in form of meta-data.

We distinguish two entities in our system model: (1) a *service provider*, e.g., a tollgate, that stores code modules and provides them for download. (2) a *service consumer*, e.g., a vehicle, receives code modules and associated adaptation actions from the service provider. An adaptation may occur when a consumer gets in communication range of a service provider. The service provider then determines the adaptation actions and the order of these actions to execute based on the consumer's current configuration, i.e., the code modules already integrated on the consumer's software platform. After the scheduling order of adaptation actions is decided, all scheduled code modules and associated adaptation actions are downloaded by the consumer, and the actual adaptation actions are executed on its local software platform.

Time bounds are imposed on the overall adaptation due to the highly mobile environment in which consumers are located. An adaptation action that is executed too late may result in inconsistent or inaccurate behaviour. Figure 2 illustrates the three different phases that subsume to the overall time bound on the *adaptation time* (a_t). The *scheduling time* (s_t) is defined as the amount of time needed to determine which adaptation actions to execute in terms of code modules to install or replace. It's triggering is denoted by the *arrival time* (arr_t), for example when a vehicle is in communication range of the tollgate. The *download time* (d_t) defines the actual time needed to download all code modules. The *integration time* (i_t) then is defined as the actual execution time of the adaptation actions. The *waiting time* (w_t) is the duration between a completed adaptation process and the adaptation deadline, e.g., a *departure time*

(Dep_t) for the vehicle entering the motorway.

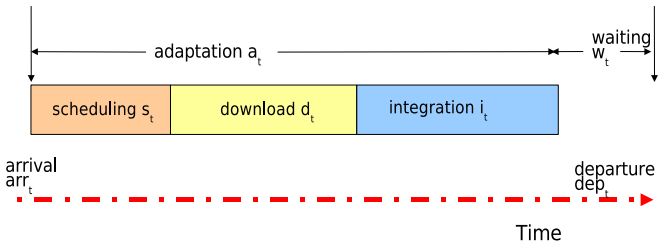

Figure 2: Time Constraints of Software Adaptation

3.2 Time-bounded Basic Scheduling Algorithm

The scheduling algorithm is located on the service provider side and maximises the amount of code modules that can be downloaded and adapted within the specified time bounds for a specific consumer's configuration. The algorithm uses a greedy-approach [7] by iterating through an ordered list of all available code modules and scheduling each module that fits within the time bounds.

The time bounds are determined for each consumer service configuration respectively. For example in the managed highway scenario, an entering vehicle sends alongside its current configuration the adaptation deadline when the complete adaptation process needs to be completed. Additionally, a worst-case estimated download time is calculated based on parameters, like distance and number of waiting vehicles. The remaining scheduling time is then calculated as the difference of the download time from the adaptation time. All code modules that are scheduled within this time are ensured to be downloaded and integrated before the deadline.

The scheduling of code modules involves the determination of which adaptation actions to take, dependent on the consumer's current configuration. If a code module is not present on the consumer's platform, the action results in an integration of this code module on the consumer's software platform. Other actions include (1) the update, i.e., exchange of a code module if there is a newer version available, (2) the downgrade of a code module or the (3) deletion of a code module.

Code modules are ordered according to an evaluation function based on the multi-attribute utility theory (MAUT) [19]. MAUT is a technique for the evaluation of objects based on multiple dimensions. For example, code modules are evaluated based on dimensions such as priorities, number of dependencies, size and integration time. The theory defines an overall evaluation function $v(x)$ that is defined as the sum of all weighted additions of all the dimensions that are relevant to an evaluation.

The basic algorithm orders code modules after their importance, i.e., high-priority modules should always be scheduled or at least tried to be scheduled as they might be safety-critical. Therefore, we consider the two dimensions *priority* and *number of dependencies* and have defined two weighted functions for these dimensions, F_P and F_D. Priorities can have the value range from 1 to 10, where 1 denotes the highest priority. The number of direct dependencies is assigned like the priorities by the code module developer. However, this number increases due to indirect dependencies that are obtained by a transitive dependency relation. For example, if code module "A" is dependent from code module "B" and "B" itself is dependent from code module "C", the number of dependencies of A is two.

The overall score of a code module then is determined by the subtraction of F_D from F_P. The score denotes the relative importance of a specific code modules, i.e., the higher the score, the more important a module is and the earlier it will be scheduled. Table 1 gives an overview of the functions used and their according value range. The priority of a code module m is denoted with $m.p$, the amount of dependencies is denoted with $m.d$. Example score values are illustrated in Table 2

Dimension	Weighted Function	Value Range
Priority	$F_P(m) = 100 - 10 * (m.p - 1)$	$F_P \epsilon$ [10, 100]
Dependency	$F_D(m) = m \bmod 10$	$F_D \epsilon$ [0, 9]
Score Value	$S = F_P - F_D$	$S \epsilon$ [100, 1]

Table 1: Weighted Functions

Dimension	Module 1	Module 2	Module 3
Priority	1	1	2
Dependencies	0	1	0
Score	100	99	90

Table 2: Example Score Values

3.3 Scheduling Examples

In the following, we discuss two example schedules for the consumer and service provider configuration illustrated below. In this example we consider an estimated download time (d_t) of 1 ms.

Scenario 1 has an overall worst-case adaptation time (a_t) of 1500 ms, scenario 2 an overall worst-case adaptation time (a_t) of 900 ms, i.e., after 1500 and 900 ms respectively the vehicle will enter the motorway and all adaptations necessary must have been executed until then. The code modules are ordered according to their score value, i.e., code module "A" will be scheduled before code module "B" and code module "C". Table 3 summarises the time constraints for

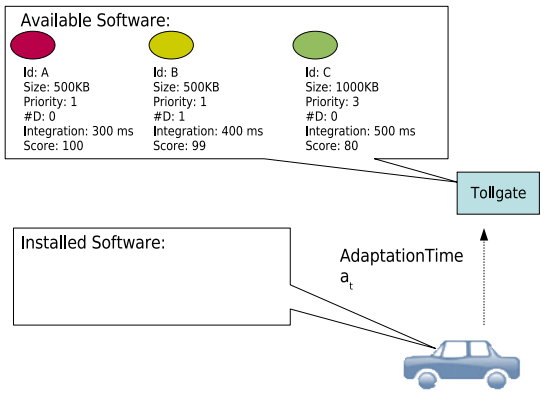

Figure 3: Configuration Example

Time Constraints	Scenario 1	Scenario 2
adaptation time	1500	900
download time	1	1
integration time	1200	1200

Table 3: Time Constraints for Scenarios

each scenario. All values are given in ms.

The following equation must hold true to schedule a set of code modules 1 ...j within the given time constraints: $\sum_{i=0}^{n-1} s_t(i) + \sum_{i=0}^{j} d_t(i) + \sum_{i=0}^{j} i_t(i) < a_t$

This equation states that the sum of the integration times of all modules scheduled plus the overall scheduling and download time for each module must be smaller than the available adaptation time.

In Scenario 1, this equation is fulfilled as the the sum of the integration times of all modules and the given download time does not exceed the overall adaptation time. Hence, all code modules can be scheduled and integrated at the consumer's side. As the vehicle does not contain any code modules, the adaptation actions would result in an integration of all three modules.

Scenario 2 cannot fulfil the above equation as the the sum of all integration times exceeds the worst-case adaptation time (1200 ms > 900 ms). Hence, it is not possible to schedule all code modules for adaptation. The scheduler in this case would linearly schedule code modules until the time bounds are exceeded. In this case, the first two modules would be scheduled. Their overall integration time plus the download time is still smaller than the available adaptation time. Likewise to scenario 1, the two code modules would be scheduled for integration.

3.4 Adaptive Scheduling Algorithm

Our algorithm can be seen as a static scheduling approaches because the ordering of the code modules is determined statically based on the module's importance, e.g., at deployment time of a repository [13]. However, some scenarios might require a different ordering of the code modules based on (dynamic) conditions of the environment. For example, code modules may need to be scheduled according to their size when dealing with consumers with very limited memory capabilities. Other scenarios may require the ordering of code modules after integration times, e.g., a code module with a very high or low integration time needs to be scheduled first.

One solution is to adapt the actual scheduling mechanisms to better reflect the current conditions by adjusting the weights on its evaluation function. [11]. Different weighted functions are provided that emphasise various aspects of the system. For example, a weighted function for memory space, would favour smaller code modules, whereas a weighted function for integration times would favour modules with higher integration times. The overall evaluation function is then chosen depending on the current situation of the consumer.

4. PROTOTYPE OF THE TIME-BOUNDED SCHEDULING APPROACH IN FUNAMBOL

This section illustrates the prototype implementation of our time-bounded scheduling approach. We have implemented our approach on top of the open-source mobile application server Funambol [2]. Funambol provides data and binary synchronisation leveraging the standard protocol SyncML [5]. In the following, we briefly discuss the three main design challenges (1) Determination of necessary adaptation actions (2) Representation of code modules and (3) Realisation of constraint-based scheduling algorithm

4.1 Determination of Necessary Adaptation Actions

Problem. The adaptation actions to execute are dependent on a consumer's current configuration. We need a mechanism that can send a description of a consumer's currently contained code modules. Based on this description, a decision can be made which adaptations to execute.

Solution approach → Leverage Funambol's built-in synchronisation strategies. The Funambol platform supports two synchronisation modes, namely partial and full synchronisation. With partial synchronisation, only modules that have been changed since the last timestamp are compared against the service provider, whereas in a full synchronisation, a complete comparison of a consumer's and a service provider's code modules is made. The actual decision which adaptation actions to execute is built into the synchronisation strategy of Funambol and depends on the time stamps of code modules. If the code module is not contained on the consumer's side, the resulting adaptation action is an integration of this code module. Otherwise, if the service provider contains a newer version of the code module, the resulting adaptation action is an update of the code module on the consumer's side. We have extended these synchronisation algorithms to support our time-bounded scheduling algorithm, taking also into account the priority and number of dependencies between code modules.

4.2 Representation of Code Modules

Problem. Code modules are the entities that are adapted. We need a representation format that allows to add additional non-functional properties, like priorities, memory size and amount of dependencies. These properties are meant to be set by the code module developer.

Solution approach → Use SyncItems with additional properties in form of meta-data. SyncItems represent the smallest binary or textual information that can be synchronised in the Funambol platform. Code modules are realised in our approach as SyncItems with priorities, memory size, amount of dependencies and integration times as non-functional properties. We have extended the basic class of SyncItems to include these additional properties and also have defined initialisation methods, e.g., a distribution function for the priority.

4.3 Realisation of Constraint-based Scheduling Algorithm

Problem. The constraint-based scheduling algorithm tries to maximise the amount of adaptation actions and thus code modules within a bounded time. We need to order the code modules after some criteria, that should be adaptable to a consumer's current limitations.

Solution approach → Use Weighted Functions for the ordering of the code modules. In our approach, code modules are ordered according to the weighted func-

tions, described in Section 3.2. The default ordering is with regards to the priority and amount of dependencies of a code module, however due to a consumer's limitations, the ordering can be recalculated on a different criteria, for example, an ordering of code modules according to their memory size.

An outline of the implementation of the constraint-based scheduling algorithm is illustrated in Listing 1.

```
double adaptationTime = Consumer.getAdaptationTime
    ();
double downloadTime= Consumer.getDownloadTime();
double schedulingTime = adaptationTime -
    downloadTime;
double integrationTimeModule;
double currentTime = 0;
for (int i = 0; i < moduleList.lenght; i++) {
  integrationTimeModule = m(i).getIntegrationTime
      ();
  if(m(i).hasDependencies()){
    dependentList = m(i).getDependencies();
    for (j=0; j < dependentList.length; j++){
        integrationTimeModule +=dependentList(j).
            getIntegrationTime();
    }

    if((integrationTimeModule + currentTime) <
        schedulingTime)
      currentTime= integrationTimeModule;
      schedule(m(i));
      schedule(dependentList);

  }


}
```

Listing 1: Implemented Mechanism

The download time and adaptation time are given by the consumer at startup of the scheduling process. A remaining *scheduling time* is then calculated. The ordered list of code modules is traversed linearly and for each code module it is determined whether the remaining scheduling time is smaller than the integration time of the code module itself and its dependent modules. The overall runtime complexity of this mechanism is $O(n)$ with n denoting the amount of code modules in the scheduling list.

5. EXPERIMENTAL EVALUATION

To assess our scheduling algorithm, we conducted three experiments that explored the following three hypotheses:

1. **Hypothesis 1: Effectiveness of the Time-constrained algorithm**: The time-constrained scheduling algorithm works correctly according to the specifications described in Section 3. The algorithm schedules higher priority modules first and takes dependencies between modules into account (basic scheduling algorithm). All scheduling is done within the given time bounds.

2. **Hypothesis 2: Efficiency of the Time-Constrained Basic Scheduling Algorithm**: Our time-bounded basic scheduling algorithm performs better compared to other approaches under the same environmental conditions and configurations.

3. **Hypothesis 3: Efficiency of the Time-Constrained Adaptive Scheduling Algorithm**: This hypothesis deals with variable weighted functions that can schedule modules according to their memory size, integration times, priorities and dependencies. The variability points of our algorithm can improve its behaviour under particular environmental conditions and configurations.

5.1 Experiment 1: Measuring the Effectiveness of the Time-Constrained Algorithm

For this subsection, we demonstrate the effectiveness of our algorithm, i.e., that the scheduling algorithm works correctly. Firstly, the algorithm always needs to schedule more important modules first. Secondly, the algorithm needs to handle dependencies in a correct way, i.e., if a code module that is currently scheduled depends on other code modules, these code modules need to be scheduled beforehand.

Handling of priorities. First, we want to evaluate how priorities are handled. For this we define a scenario in which all code modules have the same configuration (c.f., Table 4) except for their priority. We distinguish three different simulation runs: (1) Run1: All modules are of high-priority, i.e., priority 1 (2) Run2: Priorities are equally distributed between the modules (3) Run3: Service provider contains only two high-priority code modules, the remaining code modules are of lower-priority.

Constraint	Value
Dependencies	0
Module Size	100kB
Integration_Time	1ms
Download_time	1 ms

Table 4: Evaluation Configuration

Figure 4 shows the percentage of high-priority modules (i.e., modules with priority 1) received by the consumer for an increasing adaptation time. The resulting sample values are calculated as an average over 10 simulation runs. Run 1 and Run 2 show similar increasing behaviour, however the percentage of high-priority modules is smaller in Run 1 and increases slower as it contains the most high-priority modules. Hence the algorithm tries to schedule all of them within the time bounds. Due to the fact that Run 3 has to schedule only two high-priority modules, the point in time when both modules can be scheduled within the time bounds is reached much faster than in the other two runs.

Handling of dependencies. Secondly, we want to evaluate the handling of dependencies in our algorithm. For this, we have slightly changed the configuration of code modules. All code modules have the same high priority (priority 1). We distinguish three different scenarios: (1) No Dependencies: All code modules are independent from each other (2) Flat Dependencies: Code modules contain a maximum dependency degree of one (3) Deep Dependencies: Code modules can contain a maximum dependency degree of more than one.

We first illustrate the handling of dependencies by means of an example code module set, that contains 10 code modules. Table 5 shows the scheduling results, i.e., the order in which the code modules arrive at a consumer's side for the different scenarios under the assumption that the adaptation time is large enough. In the "No Dependency" scenario, all code modules have the same overall score value (c.f., Section

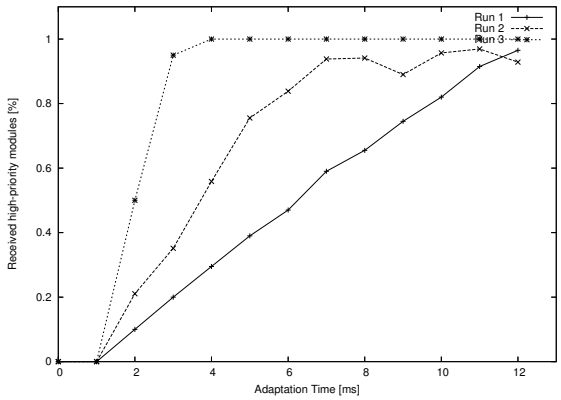

Figure 4: Priority Runs

3.2) and will be scheduled linearly. In the "Flat Dependency" scenario, all code modules that do not have any dependencies are scheduled first, as their score value is higher than the score value of code modules with dependencies. In the "Deep Dependency" scenario, module A is scheduled last, as it has the highest amount of dependencies and hence the least score value.

Scenario	Scheduling Order
No Dependencies	A B C D E F G H I J
Flat Dependencies	B D F H J A C E G I
Deep Dependencies	J I H G F E D C B A

Table 5: Scheduling Order

Figure 5 illustrates the duration of the scheduling time results for the handling of dependencies in the three scenarios for an increasing number of code modules.

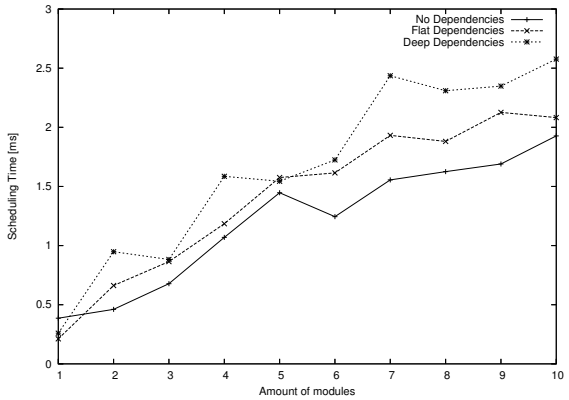

Figure 5: Dependency Runs

In the "No Dependency" scenario, code modules are scheduled the fastest, as they do not contain any dependencies. The "Flat Dependencies" scenario schedules code modules faster than than the "Deep Dependencies" scenario, because the maximum dependency degree of a code module, i.e., the amount of code modules to check, is less. In the worst case, the "Deep Dependencies" scenario has to check each code module whether it is already scheduled. To summarise, both figures show that the algorithm works correctly and hence proves our first hypothesis.

5.2 Experiment 2: Measuring the Efficiency of the Time-Constrained Basic Scheduling Algorithm

In this section, we want to show that our scheduling algorithm (TB) performs better than other scheduling approaches under the same environmental conditions and configurations. We hereby consider three existing approaches that differ in the ordering of the code modules: (1) Ordering of code modules in first-in first-out (FIFO) manner. (2) Ordering of code modules according to priorities (P) (3) Ordering of code modules according to dependencies (D).

For this evaluation, we propose the usage of a success metric. This metric indicates the amount of successful deadlines of code modules reached for each approach for an increasing adaptation time. For each sample run, i.e., a given approach and a given adaptation time, a success rate function $F(x)$ is calculated as defined below with x denoting the scheduling approach:

```
F(x) =  amount of high-priority modules scheduled
```

We apply this success rate function on an example code module set configuration for each of the four approaches. We expect our scheduling approach to perform better than the existing FIFO strategy as our approach orders the modules after importance and hence schedules higher-priority modules first. In particular, when the available adaptation time is small, a high percentage of code modules cannot be scheduled within time and hence the ordering of code modules becomes important. That implies that the amount of missed high-priority modules not being scheduled, should be less in our mechanism than using the FIFO strategy. The mechanism should show a similar behaviour when compared to the scheduling approaches that schedule code modules after priority and dependencies. However, our approach would outperform both approaches when specific code module configurations are valid, e.g., when low-priority modules have lesser dependencies than high-priority modules.

Table 6 illustrates the code module set configuration. For this example configuration, we assume an integration time of 1 ms for each code module and an available adaptation time of 3 ms. Table 7 illustrates the ordering of the example module set for each of the approaches and Table 8 shows the code modules that can be scheduled before the deadline and and the score values of F(x).

Module ID	A	B	C	D	E	F
Priority	3	3	2	1	1	1
Dependencies	0	2	0	1	1	0

Table 6: Example Module Set Configuration

Approach	Module Set
FIFO	A B C D E F
P	D E F C A B
D	A C F D E B
TB	F D E C A B

Table 7: Code Module Set Ordering

Approach	Module Set	F Score Value
FIFO	A	0
PA	D, E	0
DA	A, C	0
TB	F, D	2

Table 8: Scheduled Code Modules

The FIFO approach would schedule only module A as module B cannot be integrated due to unresolved dependencies. The resulting order for modules ordered after priority is *D* and *E*, however, both these modules are low-priority modules and the value of F would be 0. Likewise, the resulting code module set for modules ordered according to dependencies, is *A* and *C*. Both code modules also low-priority modules. Our approach would result in the scheduling of adaptation actions for code modules *F* and *D*, both high-priority modules, assigning the value 2 to the success metric.

5.3 Experiment 3: Measuring the Efficiency of Time-Constrained Adaptive Scheduling Algorithm

In this subsection, we want to show that the variability points of the algorithm can improve its behaviour under certain environmental conditions and configurations. We consider the following two approaches that differ in the weighted function applied on code modules and hence their initial ordering: (1) Ordering of code modules according to priorities and dependencies (*PD*) (2) Ordering of code modules according to memory size (*M*) We suggest to use the same success metric as defined in 5.2, that calculates the amount of high-priority modules received at the consumer.

The success metric is applied on an example code module set configuration for the two approaches. Table 9 illustrates the code module set configuration, with priorities and amount of dependencies as well as the memory size given in kilobytes. For this example, we assume an integration time of 1ms for each code module and an adaptation time of 6ms. Additionally, the consumer has a memory limitation of 15Kb.

Module ID	A	B	C	D
Priority	1	1	1	1
Dependencies	0	0	0	0
Memory Size	10	10	5	9

Table 9: Example Module Set Configuration

Table 10 shows the scheduling result of each approach. When code modules are ordered after priority, only the first code module "A" can be integrated on the consumer, as then the memory of the consumer is full. In case of an ordering after memory sizes, the code modules "C" and "D' could be scheduled, leading to a better function score value for this approach.

Approach	Module Set	F Score Value
PD	A	1
M	C, D	2

Table 10: Scheduled Code Modules

However, the performance of the adaptive approach depends on the configuration of the code modules and specific environmental conditions. For example, when all high-priority modules are relatively big in size, the outcome of this approach is expected to be under the performance of the standard approach, as high-priority modules cannot be scheduled first and hence more deadlines are missed.

6. RELATED WORK

The related work is two-fold. In the first part, we investigate scheduling approaches for data and binary features in real-time and in grid computing systems as these systems have similar requirements to our approach, e.g., timeliness of scheduled data. In the second part, we briefly discuss two approaches for the distribution of code: code package managers and over-the-air programming.

Scheduling algorithms. Jobs in real-time systems have points in times by which their execution is required to be completed, the so-called deadlines. A scheduling algorithm in a real-time system tries to allocate the resources and processors of a system in a way that all jobs can be finished before their deadline. A common approach for scheduling jobs in real-time systems is a priority-driven algorithm, e.g., earliest-deadline first (EDF) or first-in-first-out (FIFO) [14]. Priority here refers to the deadline of jobs, for example the earliest deadline first algorithm schedules first the jobs that have the closest deadline. However, these approaches schedule jobs and do not consider features per se. Also they mainly use a single dimension, like the importance or weight of the job. Our constraint-based scheduling algorithm differs from these approaches, as it considers multiple dimensions for the ordering of features, c.f., Section 3.2.

Scheduling is also an issue in data-intensive grid-based applications where data items must be efficiently allocated and transferred over intermediate nodes to their destination to meet the assigned deadline. For example, real-time tracking of storm data for avionic control has stringent time constraints and non trivial data scheduling issues due to the amount of flights a single avionic system controls [9].

The authors of [9] propose a scheduling algorithm that schedules the requests for data items based on a path selection heuristic. Multiple data items are transferred at the same time to different destination. The approach tries to maximise the amount of satisfied requests. However, their approach only considers the scheduling of data items based on the location. It does not address the scheduling of actions associated with the data items as our algorithm does, c.f. Section 3.2.

Code distribution approaches. Code package managers, like the Debian based advanced packaging tool (APT) [1] or Redhats package manager (RPM) [4], allow the automatic download and integration of software modules into a running system. They provide integrated dependency detection and resolution, i.e., software modules will be downloaded and installed together with all their dependent modules. These systems schedule the code modules as well as their associated actions, like installation and upgrades. However, unlike our algorithm, they do not take any time constraints into account

Over-the air-programming (OTA) is a technique for distributing software updates to mobile phones [3]. The software is delivered to a mobile phone's hardware platform by either explicit user action or performed automatically. How-

ever, often after a software update a mobile phone has to be restarted to take over the changes. Hence, this approach does not fall into the category of dynamic adaptation/reconfiguration.

7. CONCLUDING REMARKS

In a previous paper, we have identified the need for dynamic software adaptation in next-generation embedded systems, like automotive systems [10]. This paper presented a constraint-based scheduling algorithm that maximises the available adaptation actions that can be executed on features within given time bounds. The algorithm schedules features in a greedy manner from an ordered list. Weighted functions are applied on properties of features, like their priority, to calculate the rank in the list.

The algorithm is illustrated by means of an example taken from the domain of managed highways. Early evaluation results show that the algorithm works correctly and a success metric was used to compare our algorithm with existing approaches.

The lessons we learned developing and designing our constraint-based adaptive scheduling algorithm are:

- Current synchronisation frameworks can be leveraged to provide the basic functionality of determining the adaptation actions to execute based on a client's current configuration. Our algorithm is implemented on top of a synchronisation framework and provides the time-bounded scheduling of adaptation actions on code modules that are ordered according to some ranking.

- The ranking of code modules can itself be adaptive based on a client's limitations. The default scheduling algorithm assumes code modules to be ordered based on importance, however some scenarios require a different ordering of the code modules, e.g., taking into account memory size.

- The current implementation assumes a stable bandwidth between the service provider and a client. However, in mobile environments, unstable and rapidly changing network conditions are the norm rather than the exception. Part of our future work is to determine the time bounds, particularly the download time (d_t) for each client in a way that reflects the current reality.

- The algorithm is only concerned with the decision-process relating to which adaptation actions to execute and which code modules are affected. In our current work, we are developing a platform that supports the actual execution of the adaptations within time constraints.

8. REFERENCES

[1] Advanced packaging tool (apt). http://www.debian.org/doc/manuals/apt-howto/.

[2] Funambol. http://www.funambol.com.

[3] Ota. http://www.openmobilealliance.com.

[4] Redhat package manager (rpm). http://www.rpm.org/.

[5] Syncml protocol specification. http://www.openmobilealliance.com.

[6] R. Anthony and C. Ekeling. Policy-driven self-management for an automotive middleware. In *PBAC '07: First International Workshop on Policy-Based Autonomic Computing*, 2007.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGrawHill, 2002.

[8] I. Crnkovic. Component-based approach for embedded systems. In *Ninth International Workshop on Component-Oriented Programming (WCOP)*, 2004.

[9] M. Eltayeb, A. Dogan, and F. Ozguner. A data scheduling algorithm for autonomous distributed real-time applications in grid computing. In *ICPP '04: Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)*, 2004.

[10] S. Fritsch, A. Senart, D. C. Schmidt, and S. Clarke. Time-bounded dynamic adaptation for automotive system software. In *Proceedings of the 30th International Conference on Software Engineering (ICSE), Experience Track on Automotive Systems*, 2008.

[11] M.T. Gervasio, W. Iba, and P. Langley. Learning user evaluation functions for adaptive scheduling assistance. In *Proceedings of the Sixteenth International Conference on Machine Learning (ICML)*, 1999.

[12] C. Gill, R. Cytron, and D.C. Schmidt. Middleware scheduling optimization techniques for distributed real-time and embedded systems. In *WORDS '02: Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, 2002.

[13] C. Gill, D. Schmidt, and R. Cytron. Multi-paradigm scheduling for distributed real-time embedded computing. In *IEEE Proceedings Special Issue on Modeling and Design of Embedded Software*, 2002.

[14] J. W. S. Liu. *Real-Time System*. Prentice Hall, 2000.

[15] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[16] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[17] N. Ravi, S. Smaldone, L. Iftode, and M. Gerla. Lane reservation for highways (position paper). In *ITSC '07: Proceedings of the 10th International IEEE Conference on Intelligent Transportation Systems*, 2007.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[19] D. von Winterfeld and W. Edwards. *Decision Analysis and Behavioral Research*. Cambridge University Press, 1986.