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

ABSTRACT

Component-based software increasingly needs dynamic adaptation to support applications in domains such as automotive, avionics or robotic systems. Dynamic software adaptation involves both the integration of new, previously unanticipated features and the update of existing features without requiring system downtime. Software adaptations must often be time-bounded, *e.g.*, due to mobility constraints. Inconsistent or inaccurate behaviour may result from an adaptation that does not complete within specified time constraints. Service providers must therefore take time constraints into account when scheduling adaptation actions. This paper describes an algorithm for time-bounded scheduling of adaptation actions and demonstrates the validity of its results. [1]

Categories and Subject Descriptors

D11 [**Software/Software Engineering**]: Software Architectures

General Terms

Algorithms

Keywords

Dynamic Software Adaptation, Timeliness, Scheduling algorithm

1. INTRODUCTION

Emerging trends and challenges. Next-generation embedded systems in domains such as automotive, avionics or robotics need to adapt swiftly to changing environmental conditions [12]. In prior work [10] we have shown that these systems require varying levels of support for dynamic adaptation, ranging from 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 runtime adaptation of software in component-based driver information systems.

In this paper, we focus on one type of software adaptation called *compositional adaptation*, which allows the dynamic integration and exchange of features and resulting application behaviour at runtime [14]. Previously unanticipated or updated features can be integrated into running systems in response to external triggers, making these systems more flexible and maintainable [15].

Software adaptations are often triggered when a server that provides features is in close range. Since a server may potentially handle thousands of requests concurrently—and has knowledge of all feature characteristics it supports—it can decide which adaptation actions to execute based on the requester's current configuration [19]. Adaptation actions include the installation or upgrade of features.

Additionally, software adaptations must often be time-bounded, *e.g.*, due to mobility constraints. For example, a plugin to the driver information system can only be downloaded when the vehicle is in the vicinity of a server providing this plugin. Since the vehicle is moving, however, this download and integration must 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 a newer feature indicating gas stations on the managed highway should be installed before the vehicle actually enters the highway. The decision process that schedules the adaptations is also affected by time constraints as the more time is spent on adaptation scheduling, the less time is left to perform the adaptations.

Adaptations can also be constrained by other factors, such as priorities, dependencies and code features versions. For example, high-priority features, such as security patches, should be adapted before low-priority features, such as movie clips. Likewise, dependencies between features can affect adaptation since adapting one feature can require an update or installation of other features, e.g., adding a gas station feature might require an update of a map feature. The ver-

[1] ©ACM, (2008). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), ICSE 2008 http://doi.acm.org/10.1145/1370018.1370035"

SEAMS'08, May 12–13, 2008, Leipzig, Germany.

sion of a feature might require the update of other features, and some platforms may support only specific versions of features. Also, it might be not possible to download and integrate all necessary plugins because of a vehicle's memory limitations.

Solution approach → Constraint-based Scheduling of Adaptation Actions This paper describes a constraint-based algorithm that schedules adaptation actions that can be executed by a service provider. This algorithm takes time constraints into account, by maximising the amount of features that can be adapted within a given time on a software platform. Moreover, the algorithm considers constraints, such as limited memory and feature importance.

The algorithm works as follows:

1. It assumes features are ranked in an ordered list via certain criteria, such as priorities and amount of dependencies or memory size.

2. The ordered list of features is obtained by applying weighted functions on the features' properties, *e.g.*, priority and size. The ordering is not statically defined but dynamically assessed to better reflect the current requester's constraints.

3. The algorithm iterates linearly through the ordered list of code features and schedules all code features for adaptation possible within a given time bound.

Paper organisation. The remainder of this paper is organised as follows: Section 2 motivates our work with a scenario from the automotive domain; Section 3 describes the algorithm in detail and introduces our system model; Section 4 discusses key design challenges and their solutions; Section 5 shows some evaluation results of our approach; Section 6 compares our approach with related work on scheduling and code distribution; and Section 7 presents concluding remarks.

2. MOTIVATING SCENARIO

To motivate the need for time-bounded scheduling of adaptation actions, this section describes a scenario from the automotive domain. Figure 1 shows a intelligent lane reservation system in a next-generation managed highway that aims to reduce traffic congestion and control traffic flow, *e.g.*, by allowing emergency vehicles to arrive safely and faster at accidents [16]. One way to schedule and enforce vehicle QoS

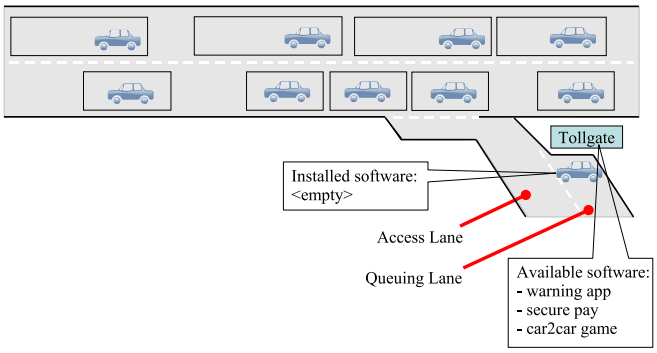

Figure 1: Managed Highway Scenario

on a managed highway is to allow drivers to reserve lanes "slots".

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. For example, adaptations can involve 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. Other examples of software adaptations include the downgrade or deinstallation of software due to memory limitations of the vehicle software platform or expirations of licences [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 relevant software is downloaded to the vehicle, the adaptations are executed on the vehicle's software platform. The overall adaption process itself is time-bounded since deciding which adaptations to perform and then downloading and adapting the software must be executed before the vehicle can enter the highway. This decision process can also be influenced by (1) the available memory on a vehicle platform, (2) software interdependencies, (3) software priorities and (4) software versioning constraints.

3. CONSTRAINT-BASED ADAPTIVE SCHEDULING

This section introduces our system model and defines the different time bounds that are used in our constraint-based scheduling algorithm. It also explains how the algorithm schedules adaptation actions according to constraints (such as time bounds) that are input as parameters at the start of the adaptation process.

3.1 Software Adaptation

Software adaptation is traditionally performed on systems composed of binary software components with specified interfaces and explicit dependencies called modules [17]. Dynamic adaptation actions include (1) integration of code modules, (2) deintegration of code modules and (3) exchange with existing code modules for up- or down-grades. Our approach assumes 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, such as 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 and (2) a *service consumer*, *e.g.*, a vehicle, that receives code modules and associated adaptation actions from the service provider. An adaptation may

occur when a consumer is within communication range of a service provider. The service provider then determines the adaptation actions and the order to execute these actions based on the consumer's current configuration, *e.g.*, the code modules already integrated on the consumer's software platform and the hardware platform. After the service provider determines the scheduling order of adaptation actions, it sends all the scheduled code modules and associated adaptation actions to the consumer, which then executes the actions 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 executes 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)*.

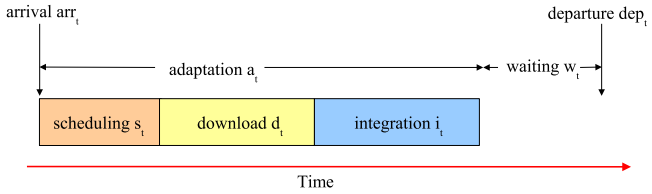

Figure 2: Time Constraints of Software Adaptation

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. Its triggering is denoted by the *arrival time (arr_t)*, *e.g.*, 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)* 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.

3.2 Time-bounded Basic Scheduling Algorithm

Our scheduling algorithm runs on the service provider side and maximises the 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's configuration.

For example, in our managed highway scenario from Section 2, an entering vehicle sends its current configuration and its departure time. A worst-case estimated download time is calculated at the tollgate based on parameters, such as distance and number of waiting vehicles. The scheduling time, *i.e.*, the time available for determining the code modules to integrate, is calculated as the difference of the download time from the adaptation time. Within the scheduling time, the algorithm iterates over the ordered list of code modules to determine which ones can be scheduled for download and integration. The determination depends on the integration time of a code module and is defined by the following equation, with j denoting the amount of scheduled code modules and n denoting the amount of available code modules.

$$\sum_{i=0}^{n-1} s_t(i) + \sum_{i=0}^{j} d_t(i) + \sum_{i=0}^{j} i_t(i) < a_t \quad (1)$$

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

The scheduling of code modules involves determining which adaptation actions to perform, depending on the consumer's current configuration. If a code module is not present on the consumer's platform, the action results in the download and integration of this code module on the consumer's platform. Other adaptation actions include (1) the upgrade, *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) [18]. MAUT defines a family of methods that are a means to analyse situations and create an evaluation process when prospective alternatives must be evaluated to determine which alternative performs best. 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 the dimensions of an object x that are relevant to its evaluation.

The basic algorithm orders code modules after their importance, *i.e.*, high-priority modules should always be scheduled (or at least attempted to schedule) since they might be safety-critical. Two modules with the same importance can differ, however, in their dependencies on other code modules. A module with less dependencies is preferred over a module that depends on a significant number of modules. In the following, we focus our discussion on how priorities and dependencies are handled for space limitations.

We have defined two weighted functions for the dimensions *priority* and *number of dependencies*: F_P and F_D. Priorities are fixed by the code module developer. Their value can vary from 1 to 10, where 1 denotes the highest priority. Dependency values range from 0 to the number of code modules -1 and can be automatically obtained by software tools using the transitive dependency relationship. For example, if code module "A" depends on code module "B"—and "B" itself is dependent from code module "C"—the number of dependencies of A is two.

The score denotes the relative importance of specific code modules, *i.e.*, the higher the score, the more important a module is and the earlier it will be scheduled. The overall score of a code module then is determined by subtracting F_D from F_P. Table 1 gives an overview of the weighted functions used and their according value range. The priority

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

Table 1: Weighted Functions

of a code module m is denoted with $m.p$ and the number

of dependencies is denoted with $m.d$. Note that the value ranges of the two functions do not overlap. Example score values are shown in Table 2.

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

We now discuss two example schedules for the consumer and service provider configuration shown in Figure 3. This

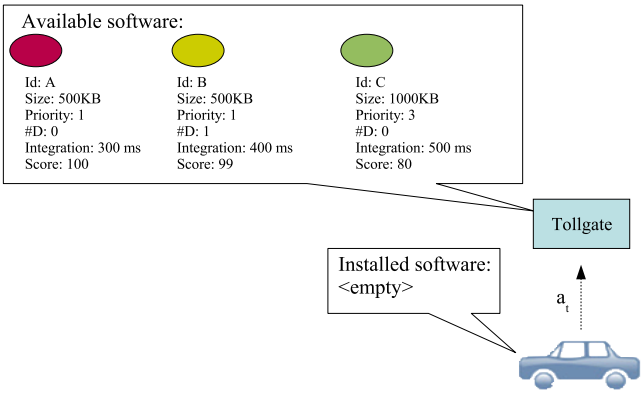

Figure 3: Configuration Example

example considers an estimated download time (d_t) of 1 ms. Scenario 1 has an overall worst-case adaptation time (a_t) of 1,500 ms, scenario 2 an overall worst-case adaptation time (a_t) of 900 ms, *i.e.*, after 1,500 and 900 ms respectively, the vehicle will enter the motorway and all necessary adaptations must be executed by 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 each scenario. All values are given in ms.

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

Table 3: Time Constraints for Scenarios

In Scenario 1, equation 1 is fulfilled since 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 the integration of the three modules.

In scenario 2, equation 1 is *not* fulfilled since 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 linearly schedules code modules until the time bounds are exceeded. In this example, the first two modules are scheduled. Their overall integration time plus the download time is still smaller than the available adaptation time. As with scenario 1, the two code modules are scheduled for integration.

3.4 Adaptive Scheduling Algorithm

As described above, our algorithm can be seen as a static scheduling approach because it orders code modules statically based on importance, *e.g.*, at deployment time of a repository

[19]. Some scenarios, however, 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 limited memory capabilities. Other scenarios may require the ordering of code modules after integration time, *e.g.*, a code module with a very high or low integration time should be scheduled first.

Our 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 favours smaller code modules, whereas a weighted function for integration times favours modules with higher integration times. The overall evaluation function is then chosen depending on the current configuration of the consumer.

4. PROTOTYPE OF THE TIME-BOUNDED SCHEDULING APPROACH

This section describes the prototype implementation of our time-bounded scheduling approach. We implemented our approach on top of the open-source mobile application server Funambol [2]. Funambol provides data and binary synchronisation based on the standard *SyncML* data synchronisation protocol [5]. Below, we discuss solutions to the three main design challenges of our prototype: (1) determining necessary adaptation actions, (2) representing code modules, and (3) realising a constraint-based scheduling algorithm.

4.1 Determining Necessary Adaptation Actions

Problem. The adaptation actions to execute depend on the consumer's current configuration. A mechanism is needed to send a description of the code modules that are currently installed on the consumer's platform. Based on this description, a decision can be made on which adaptations to execute.

Solution approach → Leverage Funambol's built-in synchronisation strategies. Figure 4 illustrates the overall synchronisation process of Funambol. The Funambol platform supports two synchronisation modes: partial and full synchronisation. With partial synchronisation, only modules that have been changed since the last timestamp are compared against the service provider. With full synchronisation, a complete comparison of a consumer's and a service provider's code modules can be performed. The consumer triggers the adaptation by sending the desired synchronisa-

tion mode, its currently contained code modules and additional capabilities, like memory constraints, to the service provider. The actual decision of which adaptation actions to execute is built into Funambol's synchronisation strategy and depends on the timestamps of code modules. If the code module is not contained on the consumer's platform, 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 platform. The affected code modules and their operations are then sent back to the consumer.

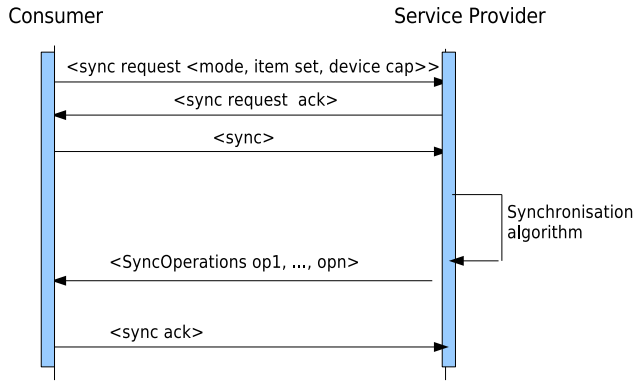

Figure 4: Funambol Synchronisation Process

In our implementation we always use the full synchronisation approach as all code modules contained at the consumer are sent to the service provider. Additionally, we sent the maximum allowed overall memory consumption. We extended Funambol's synchronisation algorithm to support our time-bounded scheduling algorithm, *e.g.*, including handling the priority and number of dependencies between code modules. Hence, we leverage on Funambol's synchronisation strategy to decide which adaptation actions to execute.

4.2 Representating Code Modules

Problem. A representation format is needed that supports the additional of new non-functional properties, such as priorities, memory size and amount of dependencies. These properties can be set by code module developers.

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 have also defined initialisation methods, *e.g.*, a distribution function for the priority.

4.3 Realising Constraint-based Scheduling Algorithm

Problem. Our constraint-based scheduling algorithm maximises the amount of adaptation actions and thus code modules within a fixed time bound. The code modules must be ordered according to criteria specific to a consumer.

Solution approach → Use weighted functions for the ordering of code modules. In our approach, code modules are ordered according to the weighted functions described in Section 3.2. The default ordering is based on the priority and amount of dependencies of a code module. The ordering can be adjusted according to different criteria, *e.g.*, an ordering of code modules according to their memory size.

An outline of the implementation of the constraint-based scheduling algorithm is shown 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 when the scheduling process begins. A remaining *scheduling time* is then calculated. The ordered list of code modules is traversed linearly and for each code module the algorithm determines 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

We conducted experiments to evaluate the effectiveness of our scheduling algorithm, *i.e.*, determine whether the algorithm works correctly. First, the algorithm always needs to schedule more important modules before less important ones. Second, 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 must be scheduled first.

5.1 Handling of priorities

We first want to evaluate how our algorithm handles priorities. We therefore define an experiment where 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, and (3) *Run3* – the service provider contains only two high-priority code modules, the remaining code modules are of lower priority.

Figure 5 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

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

Table 4: Evaluation Configuration

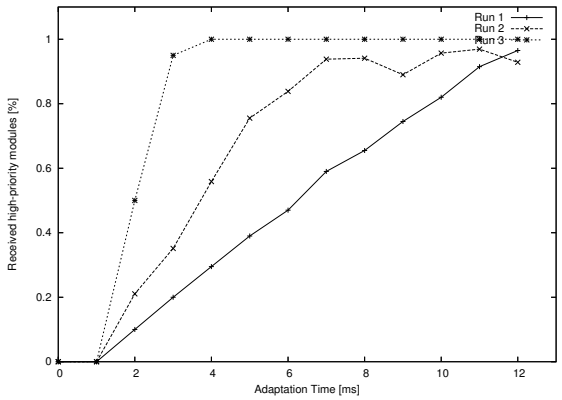

Figure 5: Priority Runs

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

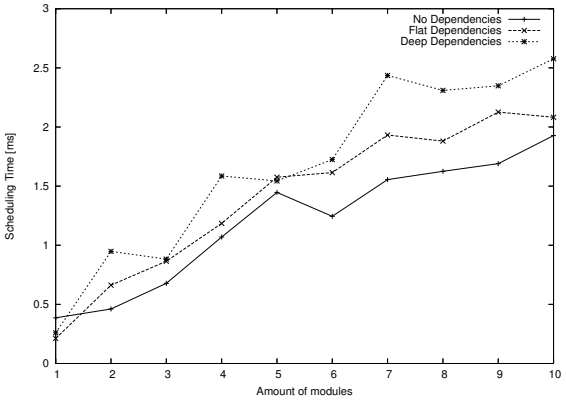

Figure 6: Dependency Runs

are calculated as an average over 10 simulation runs. Although Run 1 and Run 2 show similar increasing behaviour, the percentage of high-priority modules is smaller in Run 1 and increases slower since it contains the most high-priority modules. Our algorithm thus schedules all of them within the time bounds. Since Run 3 must 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.

5.2 Handling of dependencies

We next want to evaluate the handling of dependencies in our algorithm. We therefore changed the configuration of code modules so 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, and (3) *Deep Dependencies* – Code modules can contain a maximum dependency degree of more than one.

We first show 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 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 since it has the highest amount of dependencies and hence the lowest score value.

Figure 6 shows the duration of the scheduling time results for the handling of dependencies in the three scenarios for an increasing number of code modules. 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 lower. In the worst case, the "Deep Dependencies" scenario has to check each code module whether it is already scheduled. As a conclusion, our experiments show that our algorithm works correctly.

6. RELATED WORK

This section compares our work on scheduling time-bounded dynamic software adaptation with two areas of related work. First, we compare our work with scheduling approaches for data and binary modules in real-time and in grid computing systems since these systems have similar requirements, *e.g.*, timeliness of scheduled data. Second, 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 (*i.e.*, deadlines) by which their execution must complete. A scheduling algorithm in a real-time system tries to allocate the resources and processors of a system in a way that all jobs are 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) [13]. Priority here refers to the deadline of jobs, *e.g.*, the earliest deadline first algorithm schedules first the jobs that have the closest deadline. These approaches consider job scheduling and not module adaptation scheduling per se and therefore they focus mainly on a single dimension, such as the job importance or weight. Our constraint-based scheduling algorithm differs from these approaches by considering multiple dimensions for the ordering of modules, *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 predefined deadlines. For example, real-time tracking of storm data for air traffic management has stringent time constraints and non-trivial data scheduling issues due to the amount of flights a single avionics 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 destinations. The approach maximises the amount of satisfied requests but only considers the scheduling of data items based on the location. In contrast, our algorithm addresses the scheduling of adaptation actions associated with the data items, *c.f.*, Section 3.2.

Code distribution approaches. Code package managers, such as the Debian Advanced Packaging Tool (APT) [1] or Redhat 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 are 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. Unlike our algorithm, however, 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 either automatically or by explicit user action. Often after a software update, however, a mobile phone must be restarted to take over the changes. This approach therefore does not address dynamic adaptation/reconfiguration, unlike our approach, *c.f.*, Section 3.1.

7. CONCLUDING REMARKS

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

We showcased our algorithm via an example from the domain of managed highways. Early evaluation results show that the algorithm works correctly. The lessons that we have learned designing and developing our constraint-based adaptive scheduling algorithm so far include:

- 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 the Funambol synchronisation framework and provides time-bounded scheduling of adaptation actions on code modules that are ordered according to a configurable ranking.

- The default scheduling algorithm orders code modules based on importance, but the ranking of code modules can also be dynamically adapted according to client's limitations. Some scenarios, for example, require a dynamic ordering of the code modules, *e.g.*, taking into account current available memory or dynamic constraints versioning of code modules.

- Our current prototype assumes stable bandwidth between the service provider and a client. In mobile environments, however, unstable and rapidly changing network conditions are common. Our future work will therefore take into account the varying download time (d_t) for each client, in a way that reflects the actual bandwidth availability.

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

Acknowledgements

The work described in this paper is funded by Science Foundation Ireland under the Research Frontiers Program and Lero - the Irish Software Engineering Research Centre.

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

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

[15] 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.

[16] 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.

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

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

[19] J. White, A. Nechypurenko, E. Wuchner, and D. C. Schmidt. Optimizing and Automating Product-Line Variant Selection for Mobile Devices. In *11th International Software Product Line Conference*, Kyoto, Japan, September 2007.