DETERMINISTIC RUNTIME ADAPTATION FOR HARD REAL-TIME EMBEDDED SYSTEMS WITH PROCESSING CONSTRAINTS

Fahad Bin Tariq
Design of Distributed Embedded Systems, University of Paderborn, Fuerstenallee 11, Paderborn, Germany

Keywords: Runtime Adaptation, Determinism, Hard Real-time, Processing Constraints, Middleware, Embedded Systems.

Abstract: Increasing the connectivity of systems at various levels gives rise to potential benefits that is addressed by trends such as the Internet of Things, Cyber-Physical Systems and Cyber-Biosphere. It is inevitable that the vast global network will consist of computationally constrained units. The ability of such systems to adapt while being connected to a global network presents new opportunities. Qualities such as fault tolerance and load sharing may be extended to nodes in the network that initially were devoid of them. On the other hand, an *open* adaptable system that does not limit its adaptation to pre-defined and pre-containing *states* needs to be connected to one or more sources that may provide the necessary behavioral units to switch to the new state. Extending computationally constrained nodes with the ability to adapt is in itself challenging, which increases when timely behavior is required due to real-time requirements. The system must then guarantee all deadlines implying the need for deterministic adaptation. The question of *when* to adapt becomes an important one with real-time deadlines involved. We present our work on a framework that aims to achieve deterministic runtime adaptation on single processor systems thereby enabling such computationally constrained systems to extend their functionality and non-functional qualities.

1 INTRODUCTION

Embedded systems are typically designed to have long life cycles. Consider the systems found in modern vehicles such as airplanes and cars, or those running industrial systems. Unmanned devices such as autonomous submarines or mobile robots(Herbrechtsmeier et al., 2009) fully rely on the embedded systems within them to achieve their tasks. Once deployed, the system will be expected to run for periods ranging from months to decades (Koopman, 1996). It is inevitable that such systems come across changes in their context. This may include changes to other components working with the embedded system such as hardware replacement due to a failure. System objectives and requirements may evolve over time with changes in the environment. QoS requirements may vary over time as well as resource consumption. To cater to these changes, the system needs to adapt while maintaining a safe level of operation.

The idea of runtime adaptation in general is not new as can be seen by work done in (Fabry, 1976) as early as 1976. Its application to embedded real-time systems was only a matter of time and recent years have shown an increased interest by the embedded systems community in harnessing the benefits associated with it. Runtime adaptation is the ability of a system to change its behavior and/or structure while remaining in a desirable state of operation. A desirable state of operation implies the system remains on course to meet certain functional and/or QoS objectives.

There exist various reasons for having runtime adaptation as a characteristic property of the system. As the complexity of systems and their development process has increased over time, so have the costs and efforts to maintain them. An architecture enabling adaptation inherently supports change during the runtime phase of the system lifecycle making maintenance less costly. Resource requirements may change overtime depending on various factors such as computational load and communication requirements. The occurrence of failure on one node in a distributed environment may force topological restructuring such as the sharing of processing load or the creation of new connections. Hence the ability to adapt enables systems to acquire other properties, for example, fault tolerance.

Bin Tariq F. (2012).
DETERMINISTIC RUNTIME ADAPTATION FOR HARD REAL-TIME EMBEDDED SYSTEMS WITH PROCESSING CONSTRAINTS.
In *Proceedings of the 2nd International Conference on Pervasive Embedded Computing and Communication Systems*, pages 469-477
DOI: 10.5220/0003943504690477

The increasing complexity of embedded systems is due, but not limited, to the increasing quality and quantity of the resources involved (e.g. Multi-core processing units). It is interesting to note that such resources are there in the first place to tackle the increasing complexity brought onto the system by ever-increasing requirements and functionality (realized by software). Another approach to handle such complexity and, as a result give rise to different complexity, is to distribute the system or to increase its connectivity. Increasing the connectivity of systems at various levels gives rise to potential benefits that is addressed by trends such as the Internet of Things, Cyber-Physical System (Lee, 2008) and Cyber Biosphere (Rammig, 2008). Hence, simple computational and communication units connected with similar units give rise to complex behavior which is also the basic concept of swarms (Dorigo and Birattari, 2007). An example where units (or nodes) that are relatively simple at the individual level but combine to create more complex behavior, is a group of communicating mobile robots called bebots (Herbrechtsmeier et al., 2009). The Bebots consist of a relatively simple architecture and communicate via an ad-hoc network. Another more common example is a sensor network, which relies on very simple but large number of stationary sensing nodes, possibly distributed over a vast area to gather information. Thus it is inevitable that the vast global network mentioned above will consist of computationally constrained units.

The ability of systems to adapt while being connected to a global network presents new opportunities. Qualities such as fault tolerance and load sharing may be extended to nodes in the network that initially were devoid of them. Conversely, an *open* adaptable system that does not limit its adaptation to pre-defined and pre-containing *states* needs to be connected to one or more sources that may provide the necessary components to switch to the new state. Extending computationally constrained nodes with the ability to adapt is in itself challenging, which increases when timely behavior is required due to real-time requirements. The system must then guarantee all deadlines implying the need for deterministic adaptation. The question *when* to adapt becomes an important one with real-time deadlines involved. In literature we find two common techniques namely (a) adapt at once and (b) adapt on demand. In the former, all the required adaptation steps are scheduled in a single block. The duration of this block is not fixed and depends on the amount of adaptation. In the latter, adaptation takes place only when the functionality, to be directly effected by the adaptation, is accessed. In the first case, the duration of the adaptation is unpredictable, whereas in the second case, access times become unpredictable since an access may trigger adaptation.

In this paper we present an approach to tackle the issue of runtime adaptation on computationally constrained systems with real-time constraints. Our focus is on systems with only a single general purpose processor, with the responsibility to run the system and application software. In section 2, we introduce our Framework followed by the architecture and evaluation in sections 3 and 4 respectively. Related work is then found in section 5 followed by the conclusion in section 6.

2 FRAMEWORK

The inevitable presence of computationally constrained systems in emerging paradigms such as the Internet of Things, has already been emphasized above. This section introduces a framework that enables such systems to adapt during runtime and partake in the dynamics of the global network thus created. To achieve this the framework, viewed from the bottom up, provides a supporting mechanism to (a) enable the behavior to execute and (b) enable the system to deterministically adapt. Adaptation may include changes to the current behavior of the system, as well as addition of new behavior. These tasks are the responsibility of the *Middleware*. To facilitate the work of the Middleware and pave way for efficient behavioral reconfiguration, a *behavioral model* is provided. The model lays forth the anatomy of the application designed for the embedded system enabling the behavior to be captured in a desired structure. Further, the model lays down rules of communication that allow for efficient connection reconfiguration between the behavioral units. The idea is to support the adaptation process with applications having features making them inherently adaptive. Hence in order to understand the tasks of the Middleware, a brief description of the behavioral model is first provided.

2.1 Behavioral Model

Embedded systems tend to be application specific, the question *how* an application should be implemented, instead of *what* it implements, is addressed by the behavioral model. The result is an application behaving desirably from a structural point of view, and communicating between its constituent components without any explicit connections. This is achieved without the need to dictate or even know its functional behavior. In our model, the behavior is seen as a composition of

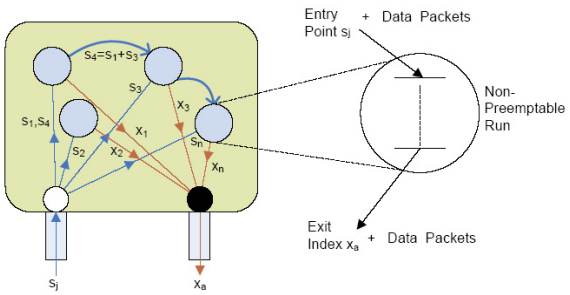

Figure 1: Behavioral unit with entry points and exit indices.

behavioral units, as is done in *programming-in-the-large*. This is in contrast to *programming-in-the-small* where granularity of models is at the level of classes, objects and methods. The behavioral units are developed independently and appear to each other and the underlying system as black boxes. If the behavior consists of k behavioral units, then the composition of behavioral units $C_i, \forall i : 1 \leq i \leq k$ making up the behavior is denoted by $\hat{C}$.

Typically, a behavioral unit has a required and provided interface, where the former specifies the prerequisites of the behavioral unit before it can fulfill its functionality. The provided interface then, manifests the different results the behavioral unit produces on completing its function. In this model, the required interface consists of many *entry points*, as can be seen in Fig. 1. Each entry point s_j can be seen as providing an access point to the functionality a behavioral unit has to offer. Hence, different entry points allow for access to different functionality of the behavioral unit. The set of entry points of a behavioral unit C is denoted by EP_C.

A behavioral unit may, for example, offer two compression techniques. A client wishing to access one of these must then provide the data to be compressed along with the associated entry state. Further, a behavioral unit may combine two or more internal functionalities to fulfill a composite function. For example, a third compression technique may be provided by combining the first two. This composite function will again have only one associated entry point instead of requiring the client to provide an entry point for every function involved. Thus from the client perspective, access to a simple or composite functionality of the behavioral unit is transparent. An entry point triggers an atomic run that, upon completion, releases an *exit index*. An exit index x_i denotes the state that the behavioral unit is in, from a set of possible and predefined states. Further, an exit index may be accompanied by data generated by the behavioral unit during execution. The underlying system (section 2.2) notifies other behavioral units about the received exit index based on behavioral unit connections registered in the *connection list*.

A connection is simply the association of an exit index of a behavioral unit to an entry point of another behavioral unit. In other words, the generation of an exit index by a behavioral unit may trigger the entry point of another behavioral unit. One exit index may be associated with one or more entry points and vice versa. Thus the connection list is a mathematical relation between all the exit indices and their triggered entry points, as given below:

$$CL = \{(x_i, s_j, o) | x_i \in XI_{Cl}, (s_j \in EP_{Ck} \vee s_j \in EP_{C\emptyset}), \\ \forall C_k, C_l \in \hat{C}, \forall o \in \mathbb{N}\}$$

$\hat{C}$ is the set of behavioral units in the system, and C_k and C_l are two such behavioral units. XI_{Cl} are the exit indices belonging to behavioral unit C_l, whereas EP_{Ck} are the entry points of behavioral unit C_k. $EP_{C\emptyset}$ specifies the possibility of an exit index not being associated with the entry point of any behavioral unit. An exit index may be associated with more than one entry point of the same behavioral unit, hence a turn based approach is used to disambiguate between the entry points. The ordering o specifies which entry point to choose at a given instance.

A behavioral unit has certain platform independent and platform specific properties associated with it. These properties are packaged along with the behavioral unit and called the meta-data. The meta-data consists of, for every behavioral unit, the platform independent relative deadline, the platform specific worst case execution time and worst case memory usage. Besides these, pointers to the starting point of behavioral unit code execution and communication interface are also provided. These are used by the underlying system to transfer execution control and send/receieve data to/from the behavioral unit via the Middleware-Application Interface.

2.2 Middleware

The mechanism to enable the behavior to execute and the system to adapt, is provided by the middleware. From the application behavioral units perspective(section 2.1), the middleware acts as a execution engine on top of which the behavioral units execute and communicate. From a node perspective, in a networked environment with different nodes running their own instances of the execution engine, the engine is the middleware through which nodes communicate. Even on a single node, the execution engine is the middleware that separates the application behav-

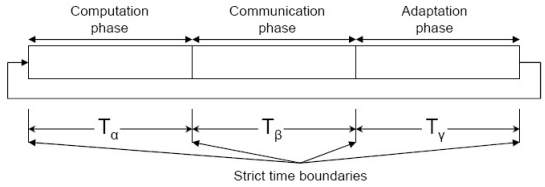

Figure 2: Timed phases.

ioral units from the underlying operating system and other behavioral units.

The middleware consists of two modes namely, the *initialization mode* and the *running mode*. In the initialization mode, all the behavioral units that currently makeup the application are *plugged in*. Plugging in the behavioral units consists of allocating resources such as memory space, registering the metadata and linking the execution and communication interface pointers with the middleware. The initial connection list is then used to register the connections between the behavioral units.

The drawbacks of two approaches to adaptation namely *adapt at once* and *adapt on demand* were mentioned in section 1. The middleware employs a highly deterministic approach called *timed phases* to achieve the intended objectives of behavioral execution and runtime adaptation. As already mentioned, the target embedded systems have processing constraints with only one processing unit, therefore the timed phases approach divides the processing timeline of the single processor into phases for computation, communication and adaptation, as shown in Fig. 2. Effectively, processing time is allocated for the three separate concerns. Adaptation is not treated by the system as a low priority, optional activity but as a main concern. The approach not only aims to guarantee deterministic behavior from the functionality perspective but also guarantees processing time for adaptation activities.

After the behavioral units have been plugged in and connections registered, the middleware initializes the *timed phases* with the help of services offered by the underlying real-time operating system (section 3), such as the timer and interrupt services. The lengths of the computation, communication and adaptation phases, denoted by T_α, T_β and T_γ respectively, may differ from one another and are predetermined based on the current application. Once set, the three phase lengths remain constant throughout the system lifetime.

The commencement of the timed phases signals the switch of the middleware to the *running mode*. In the running mode, the three phases are executed one after the other. The end of the adaptation phase triggers the start of another computation phase and the phases are repeated in cyclic manner, as is shown in Fig. 2.

2.2.1 Computation Phase

The behavioral units are scheduled for execution during the *computation phase*. Every behavioral unit flagged for release is given the processor based on the scheduling policy being used. The scheduling policy, for instance the Earliest Deadline First (EDF) or Fixed Priority (FP), is chosen at design time and remains constant throughout the system lifetime. Upon release, a behavioral unit receives an entry point and a block of data (mail) registered under it in the middleware. This mail contains the data required by the behavioral unit to proceed with the functionality. As mentioned earlier in section 2.1, upon completion the behavioral unit releases an exit index and data. Further, a behavioral unit may have data that it uses over a period of time, for example the average temperature of a device over a given duration. Such data is classified as the *state* of the behavioral unit and is recorded by the middleware at the end of the computation phase.

2.2.2 Communication Phase

In the *communication phase* the middleware executes the *communication protocol*. The connection list is used to map the exit indices to entry points. This process employs the publish-subscribe pattern by making a behavioral unit have the task of only publishing an exit index without having any knowledge of subscriptions to the exit index. The subscribing behavioral unit(s) receive an entry point from the middleware based on the connection list. Data is transferred by providing a buffer to the publisher where standard size packets are created. This buffer along with the packets, is then routed to the subscribing behavioral units. The data is consumed during the computation phase. All inter-behavioral unit communication including data transfer is done indirectly via the middleware. Due to this, communication between geographically distant nodes connected via a network appears transparent to all behavioral units on these nodes.

2.2.3 Adaptation Phase

The *adaptation phase* runs in two modes namely the normal mode and the adapt mode. The middleware checks at the beginning of the phase for an adaptation request. If none is found, the normal mode is executed where soft real-time messages are exchanged for system diagnostics and the system is monitored to check

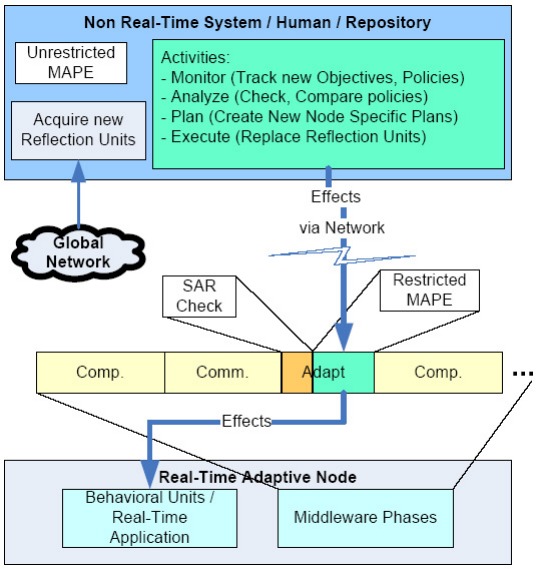

Figure 3: Hierarchial adaptation.

if adaptation is required. An adaptation request may be triggered locally or remotely. A local request is sent from a *reflection monitor* responsible to reflect upon the system and ensure certain constraints are met. These constraints vary depending on the adaptation policy. For example, the system may have a policy of executing a certain load during the computation phase, which may decrease depending on the energy utilization of the system. It is possible that at a certain energy level, the monitor decides to decrease the load and hence send an adaptation request. Terminology introduced by (Ibm, 2006) and commonly used for adaptive systems namely the Monitor, Analyze, Plan and Execute cycle (MAPE) may be applied to the activities of the reflection monitor. The MAPE activities of the reflection monitor are limited to the ones present within the monitor resulting in a restricted-MAPE. Since the monitor itself is a behavioral unit, it may be replaced during runtime *in effect making the local adaptation policy replaceable*, as can be seen in Figure 3. A separate system (e.g. a server), automated or human controlled, and connected via network, monitors the adaptable node (current system) as well as for the presence of new adaptation policies and node objectives. New node objectives and/or adaptation policies may require new reflection monitors not present with the server. These reflection monitors are then acquired from third-parties via the global network. Since there is no restriction to the change of objectives or adaptation policies that can be created locally or acquired by the server, the activities can be termed as unrestricted-MAPE. A request to replace the reflection monitor can be sent to the adaptive node.

Having dealt with local adaption requests triggered by the reflection monitor, remote requests may also be received from an external source, such as a human controller or a monitor in another node, in a networked system. If an adaptation request is found, the middleware enters the adapt mode and executes the *adaptation protocol*. A Standard Adaptation Request (SAR) specifies the *type* of adaptation required which can either be the addition/removal of behavioral units and/or a reconfiguration of behavioral unit connections. Reconfiguring connections during adaptation is the simple task of changing the connection list, instead of needing to go through every behavioral unit and changing the connections, since no direct connections exist and all inter-behavioral unit communication is handled by the middleware. The new connection configuration is read from the SAR. In case where whole behavioral units need to be added, the SAR contains the quantity of the behavioral units and their meta-data, mentioned in section 2.1. This meta-data is used for admission control where, based on the scheduling strategy used, the middleware checks the processor utilization to conclude whether the timing and memory requirements of existing and new behavioral units can be met. In case they are met, an approval is sent to the source of the request to start transfer of the behavioral unit binaries.

The length of the adaptation phase T_γ is not fixed since no optimal value, applicable to all scenarios, exists. This follows from the work done by Gupta which proves that *"determining appropriate constraints for the timing of an update is undecidable in general"*(Gupta, 1994). An appropriate value for T_γ can be determined case by case, considering certain parameters such as the expected adaptation frequency and application performance requirements.

2.3 Performance in Highly Adaptive Environment

The presence of multiple processing units gives rise to the possibility of delegating adaptation to a dedicated processor or one with less computation load. When only a single processing unit is available, computation time must be allocated to adaptation while guaranteeing deadlines of existing hard real-time computation load. Any guarantee depends on the system remaining deterministic while accommodating adaptation.

Having already outlined the benefits of adaptation made possible in an open and networked environment, the assumption that systems will be working in highly adaptive environments is realistic. Efforts in areas such as Cloud Computing outline a path towards

leaner systems that acquire functionality on demand. This results in many design time decisions to be forwarded to system runtime enabling an increase in decision accuracy.

We assume an adaptive system deployed in such a scenario with a high frequency of adaptation. In resource constrained embedded systems, besides having timed phases, other possibilities are to have adapt-at-once or adapt-on-demand approaches, mentioned in section 1. We compare the adapt at once approach with the timed phases one. Without loss of generalization, let cp denote the time allotted for computation, cm for communication, ac the time needed to only check if an adaptation request has arrived and ap the time allotted for adaptation in a single phase. Let T_a denote the cycle of a system Sys_a using the timed phases approach and T_b the cycle of a system Sys_b using the adapt-at-once approach. We assume a cycle to have finished in in Sys_b once cp,cm and ac have been allotted once. Hence:

$$T_a = cp + cm + ac + ap$$
$$T_b = cp + cm + ac$$

To get a comparable term, we acquire a common multiple (c.m.) of T_a and T_b.

$$\widetilde{T} = T_b \times T_a$$

$\widetilde{T}$ implies that in Sys_a, there are T_b cycles of T_a and similarly in Sys_b there are T_a cycles of T_b. Let $cp + cm + ac$ be denoted by t_{cca}, which is also equal to T_b. So,

$$T_a = t_{cca} + ap$$

In $\widetilde{T}$ there are T_b cycles of t_{cca}:

$$\widetilde{T} = T_b(t_{cca}) + T_b(ap) \quad (1)$$

From Sys_a perspective in $\widetilde{T}$, $T_b(ap)$ time was given for adaptation. We assume an adaptation request took that much time to successfully complete up till time $\widetilde{T}$. From Sys_b perspective there exist T_a cycles of computation and communication up till time $\widetilde{T}$:

$$\widetilde{T} = T_a(t_{cca})$$

As already mentioned, we need $T_b(ap)$ amount of time to adapt, hence in Sys_b adaptation needs to start at T_{start}:

$$T_{start} = \widetilde{T} - T_b(ap)$$
$$T_{start} = T_b(t_{cca} + ap) - T_b(ap) \quad \text{(from (1))}$$
$$T_{start} = T_b(t_{cca})$$

Thus Sys_b also provides T_b computation and communication cycles as Sys_a, after which adaptation will need to start to be successfully completed. Hence having an adaptation phase in every cycle will not decrease system performance in a highly adaptive environment. On the contrary, Sys_a performs better when it comes to managing adaptation with hard real-time tasks already scheduled. Assume there is a high priority periodic task with a period P_{task}, with $P_{task} \approx T_a$. In Sys_b, a stretch of $T_b(ap)$ after T_{start} for adaptation will either not start an instance of the periodic task or not complete adaptation till $\widetilde{T}$. Hence another feature of the approach employed by the middleware, is besides guaranteeing determinism for the hard real-time behavior, a guarantee that adaptation will take place and not denied processing time by a high priority activity, is also provided.

3 ARCHITECTURE

The behavioral model and middleware described in section 2 gave an idea of the architecture of a system following the frameowork. The application behavior is vertically partitioned into a set of loosely coupled behavioral units that interact indirectly via the middleware below. Hence taking a horizontally partitioned view, the top layer consists of the application layer followed by the middleware layer, as can be seen in Fig. 4. To achieve low level functionality such as allocating storage space to incoming behavioral units, garbage collection after behavioral unit removal, scheduling behavioral units for execution on the single processor, timer interrupt handling etc., the middleware uses the services of a Real-Time Operating System (RTOS). The RTOS is also responsible to act as an interface between the processing unit and software layers running above. To make the middleware independent of any specific RTOS, an Operating System Adaptor Layer (OSAL) is present between the middleware and the RTOS. The middleware does not make any direct calls to any specific RTOS services but to generic services provided by the OSAL. The middleware can be ported to any RTOS by porting the OSAL to the desired RTOS. The scope of the framework concepts encapsulates the top three layers and is called Framework for Deterministic Runtime Adaptation of Real-time Embedded systems (F-DRARE).

The system is a node in a network with each node having the same software architecture with the possibility of heterogeneous hardware platforms, or may even be a standalone system such as a single robot. A node may even have many application specific processing units (e.g. DSP) with the assumption that only a single processing unit is provided for the system software. In the networked case (Fig.4), each node runs its own independent copy of the middle-

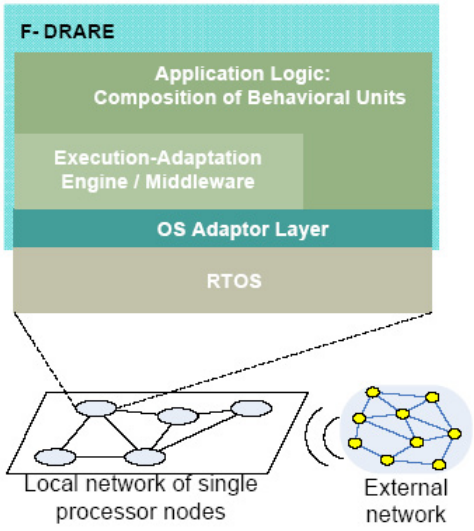

Figure 4: Architecture of system on a node in a network.

ware. The application running at the top layer may be distributed into many behavioral units running on different nodes. Keeping with the idea of Cyber Physical Systems, these nodes may be connected to external networks with or without the same architecture. A server used for external monitoring, as mentioned in section 2.2, is connected and also acts a behavioral unit repository.

A node may send/receive an SAR and behavioral units to/from either the repository, other nodes in the local network or an external network. The nodes in the local network synchronize by using any bounded delay network protocol. This enables behavioral units of a distributed application present on different nodes to synchronize via the middleware and also fulfill adaptation timing requirements.

4 EVALUATION

The concepts presented earlier have been implemented and an evaluation is shown in this section. The middleware or execution/adaptation engine of F-DRARE has been implemented in C++, as well as the OSAL. Following the behavioral model enabled the creation of different behavioral units loosely coupled by mapping exit indices to entry points in the connection list. The resulting composition became the application. Similar to the middleware, the behavioral units were also implemented in C++. The TrueTime Kernel(Cervin et al., 2003) has been used as the underlying RTOS, which runs on Matlab/Simulink.

The example used consists of two categories of uni-processor Bebots. The first category consists of lighter and cheaper worker-bots with a small camera,

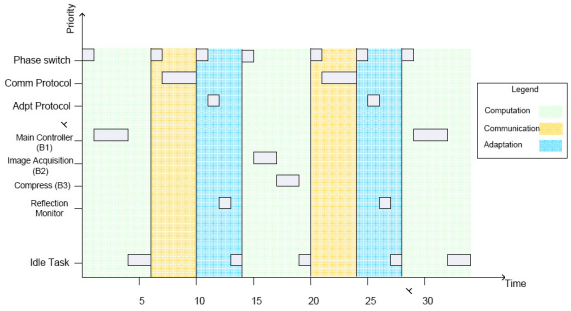

Figure 5: Normal mode execution.

a general purpose processor, wireless communication device, a motor for movement and a limited power supply. The second category consists of server-bots equipped with an additional dedicated visual processing unit (Digital Signal Processor). All the worker-bots have a server-bot within communication range at all times. The Bebots move around an area in search of a chemical found in plants growing in the designated area. The presence of the chemical is identified by analyzing the shade of the leaves of the plant in question. The application on a worker-bot consists of behavioral units B1, the main controller also responsible for the direction of movement; B2, responsible to acquire the image using the on-board camera; and B3, responsible to compress the image and send it to the server-bot which in-turn analyzes it and returns the boolean result (chemical present or not). B2 and B3 are registered to listen to the exit index of B1 resulting in respective entry points. The entry point for B2 instructs it acquire an image whereas the entry point for B3 instructs it to compress and send the last image to the server-bot. In turn, B1 is registered to listen for the exit indices of both B2 and B3, which effectively delivers the results to B1.

A normal mode simulation with the granularity of time in milliseconds, is shown in Fig 5. The end of the computation phase is triggered by the start of the communication phase. Similarly the end of the communication phase is triggered by the start of the adaptation phase. The phase durations T_α, T_β and T_γ, correspond to current simulation values of 0.6 ms, 0.4 ms and 0.4 ms respectively . The small duration of the *phase switch* triggering the start of a phase specifies the handover activity carried out by the middleware when switching from one phase to the next. The behavioral units are released for execution during the computation phase. The communication and adaptation protocols described in section 2.2, and denoted here by *Comm protocol* and *Adpt protocol* are executed in their respective phases. A reflection monitor reflects upon the system during the adaptation phase. In this example, the reflection monitor tracks the en-

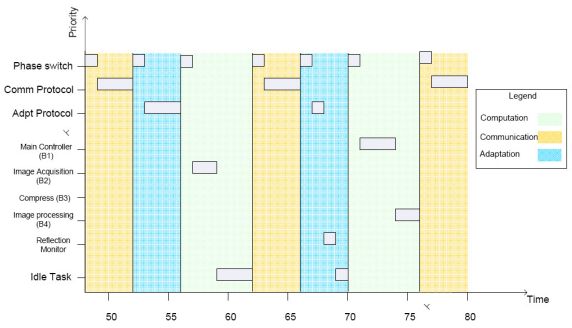

Figure 6: Reconfiguration of behavioral units and connections.

ergy consumption of the robot to check that enough samples are analyzed before the battery runs out. During the course of the search it is realized that enough samples will not be analyzed with the current energy consumption. A local adaptation request is triggered and according to the current adaptation policy, the behavioral unit B3, responsible to compress and communicate with the server-bot, is replaced with a new behavioral unit B4, with a software image processing algorithm. This results in the worker-bot saving energy by not communicating with the server-bot and instead relying on a less accurate software analysis.

Simulation results in Fig. 6 depict the handling of the adaptation request. At time 53 ms an SAR is processed resulting in the replacement of behavioral unit B3 with the new behavioral unit B4. Connection reconfiguration also occurs with B4 registering to the exit index of B2, resulting in the image taken by B2 to be sent to B4 for analysis. Connection entries related to B3 are removed. The following computation phase executes the already scheduled B2 followed by execution of the new behavioral unit B4 in the computation phases starting at time 70 ms.

5 RELATED WORK

The introduction of this paper already stated that the concept of runtime adaptation has been present with some variation as far back as the pioneering work done by Fabry in 1976 (Fabry, 1976). A great variation exists from naming (e.g. online update, runtime reconfiguration etc.), as well as the target system characteristics, to approaches for achieving it. For instance, Fabry in his work strives to change Abstract Data Types with the scope of the change already provided by the programmer and realized by using privileged instructions. A good survey on these variations can be found by Vandewoude et. al. in (Vandewoude, 2002). The same authors present their work on a Java-based component system SEESCOA (Vandewoude and Berbers, 2004) that employs the *adapt at once* technique, mentioned in section 1. Ritzau et.al. also present a Java based approach where the Java Virtual machine (JVM) is extended (Ritzau and Andersson, 2000). This work is notable since it uses the other approach mentioned in section 1, namely *adapt on demand*. We differ from these and similar approaches with respect to the timing of the adaptation by employing the timed phases to achieve determinism, a top priority in hard real-time systems. Another difference is the special focus on processing constrained systems.

The Flexible Resource Manager tries to optimize system behavior using a heuristic based on latest requirements (Hojenski and Oberthür, 2006). This is achieved by toggling between a set of pre-defined profiles to reach the goal of resource optimization. Besides resource optimization, our work enables the acquisition of new behavior un-defined at system design time. Heavier adaptive platforms such as the Simplex Architecture (Sha et al., 1995), besides having other differences, are not suited to the target embedded system in our focus.

In (Andersson et al., 2009), we find a reference model for adaptive systems which enables a qualitative evaluation of systems with the ability to reflect upon themselves, a necessary quality for self-adaptive systems. This is done by analyzing a system via the given reflection prism.

6 CONCLUSIONS

In this paper we have presented a framework that provides a behavioral model and an underlying support mechanism to achieve deterministic runtime adaptation for processing constrained embedded systems. With the concepts of Cyber Physical Systems and the Internet of Things changing the stance from closed to open and connected embedded systems, the potential to harness new abilities such as the acquisition of system qualities as well as functionality not present at design time, becomes a possibility. To achieve this, systems must have the ability to adapt while simultaneously fulfilling their timing guarantees. It is inevitable that the vast global network will consist of nodes with processing constraints, making adaptation a problem. The work in this paper moves towards solving this problem. A realization of the concepts and evaluation was done by presenting F-DRARE.

REFERENCES

Andersson, J., de Lemos, R., Malek, S., and Weyns, D. (2009). Reflecting on self-adaptive software systems. In *SEAMS*, pages 38–47.

Cervin, A., Henriksson, D., Lincoln, B., Eker, J., and rzn, K.-E. (2003). How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime. In *IEEE Control Systems Magazine*.

Dorigo, M. and Birattari, M. (2007). Swarm intelligence. *Scholarpedia*, 2(9):1462.

Fabry, R. S. (1976). How to design systems in which modules can be changed on the fly. In *Intl. Conf. on Software Engineering*.

Gupta, D. (1994). On-line software version change.

Herbrechtsmeier, S., Witkowski, U., and Rückert, U. (2009). Bebot: A modular mobile miniature robot platform supporting hardware reconfiguration and multi-standard communication. In *Progress in Robotics, Communications in Computer and Information Science. Proceedings of the FIRA RoboWorld Congress 2009*, volume 44, pages 346–356, Incheon, Korea. Springer.

Hojenski, K. and Oberthür, S. (2006). Towards self-optimizing distributed resource management. In *Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS 06)*, Kassel, Germany.

Ibm (2006). An architectural blueprint for autonomic computing. *Quality*, 36(June):34.

Koopman, P. (1996). Embedded system design issues (the rest of the story). In *Proceedings of the 1996 International Conference on Computer Design, VLSI in Computers and Processors*, ICCD '96, pages 310–, Washington, DC, USA. IEEE Computer Society.

Lee, E. A. (2008). Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*. Invited Paper.

Rammig, F. J. (2008). Cyber biosphere for future embedded systems. In *Proceedings of the 6th IFIP WG 10.2 international workshop on Software Technologies for Embedded and Ubiquitous Systems*, SEUS '08, pages 245–255, Berlin, Heidelberg. Springer-Verlag.

Ritzau, T. and Andersson, J. (2000). Dynamic deployment of java applications. In *IN JAVA FOR EMBEDDED SYSTEMS WORKSHOP*.

Sha, L., Rajkumar, R., and Gagliardi, M. (1995). Evolving dependable real-time systems. In *IEEE Aerospace Applications Conference*, pages 335–346.

Vandewoude, Y. (2002). Run-time evolution for embedded component-oriented systems. In *in Proceedings of the International Conference on Software Maintenance*, pages 242–245. IEEE Computer Society.

Vandewoude, Y. and Berbers, Y. (2004). Supporting runtime evolution in seescoa. *J. Integr. Des. Process Sci.*, 8:77–89.