# EMWF: A Middleware for Flexible Automation and Assistive Devices

Ting-Shuo Chou, Yu Chi Huang, Yung Chun Wang, Wai-Chi Chen, Chi-Sheng Shih, Jane W. S. Liu

▶ **To cite this version:**

Ting-Shuo Chou, Yu Chi Huang, Yung Chun Wang, Wai-Chi Chen, Chi-Sheng Shih, et al.. EMWF: A Middleware for Flexible Automation and Assistive Devices. 8th IFIP WG 10.2 International Workshop on Software Technologies for Embedded and Ubiquitous Systems (SEUS), Oct 2010, Waidhofen/Ybbs, Austria. pp.191-203, 10.1007/978-3-642-16256-5_19 . hal-01055386

**HAL Id: hal-01055386**

**https://inria.hal.science/hal-01055386v1**

Submitted on 12 Aug 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# EMWF: A Middleware for Flexible Automation and Assistive Devices

Ting-Shuo Chou[2], Yu Chi Huang[2], Yung Chun Wang[1], Wai-Chi Chen[3], Chi-Sheng Shih[3], and Jane. W. S. Liu[1]

[1] Institute of Information Science, Academia Sinica, Taipei, Taiwan
{wych, janeliu}@iis.sinica.edu.tw
[2] Computer Science Department, National Tsing-Hua University, Taiwan
[3] Computer Science and Information Engineering Department, National Taiwan University, Taipei, Taiwan

**Abstract.** EMWF (Embedded Workflow Framework) is an open source middleware for flexible (i.e., configurable, customizable and adaptable), user-centric automation and assistive devices and systems. EMWF 1.0 provides a light-weight workflow manager and engines on Windows CE, Windows XP Embedded, and Linux. It is for small embedded automation devices. EMWF 2.0 also provides basic message passing and real-time scheduling mechanisms and workflow communication facility. This paper describes EMWF 1.0 and extensions in EMWF 2.0, as well as case studies on workflow-based design and implementation as motivations for EMWF and the extensions.

**Keywords:** Workflow-based architecture, embedded automation advices, workflow management and engine

## 1 Introduction

This paper describes a middleware designed to easy the effort in building high-quality, low-cost personal and home automation and assistive devices and systems. We refer to these devices (and systems) collectively as UCAADS (User-Centric Automation and Assistive Device and Systems). Smart medication dispensers, autonomous appliances, service robots and robotic helpers described in [1-7] are examples. These devices aim to improve the quality of life and self-reliance of their users, including elderly or functionally limited individuals. Other examples of UCAADS are automation tools (e.g., [8, 10]) for use in care-providing institutions for enhancing the quality and reducing the costs of medical and health care.

Despite vast differences in their purposes and functions, UCAADS have many common requirements. First and foremost is *flexibility*; by that, we mean configurability, customizability and adaptability. A flexible device can be configured to support different processes and rely on different support infrastructures. It can be easily customized to suit different users. The device should also be able to adapt to serve the user well over time as the user's needs change.

We have adopted the workflow paradigm [11, 12] as a means to achieve flexibility. This paradigm has been widely used in enterprise systems for automation of business

processes. Using this paradigm, the developer decomposes work to be done by the application into basic building blocks called *activities.* An activity may be done by executing a software procedure. We call such an activity a *software activity*. An activity may also be an operation by a hardware device, a message delivery by a network, and so on. Some activities are actions of human users. We call all of these activities *external activities*. Activities are composed into module-level components called *workflows*. The order and conditions under which activities in a workflow are executed and the resources needed for their execution are defined by the developer of the workflow. The definition can be in terms of some programming language (e.g., C# in [11]), a process definition language (e.g., XPDL, WfMC standard XML Process Definition Language [12, 13]), or an execution language (e.g., BPEL, Business Process Execution Language [14]). Workflows can also be defined graphically [11, 15]: In a *workflow graph*, nodes represent activities in (or states of) workflows and directed edges represent transitions between activities (or states).

A key component of a platform for workflow-based applications is the *workflow engine* (or *engine* for short). The engine sequences activities in each workflow, coordinate activities that share resources, schedules and manages activities that are ready to run and in case of software activities, execute them. The engine also provides the applications served by it with *built-in activities*: They are activities that start and stop workflows, and alter the timing and flow paths within workflows during executions. To design and implement a workflow-based application, the developer only needs to define the workflows in the application and provide the resources required by the activities in them and then leaves the engine to integrate workflow components dynamically at runtime as specified.

Today's matured engines and tools for defining, building and executing workflows (e.g., [11-18]) are primarily for applications that automate business processes on enterprise computers and mobile devices. They are not well suited for UCAADS that have embedded components running at high rates and interacting closely with hardware devices. EMWF (Embedded Workflow Framework) presented here is a workflow management system designed specifically for such devices. Typical UCAADS are not as severely power and size constrained as cell phones and PDA's. Consequently, energy consumption and memory footprint requirements of EMWF engine and workflow applications are not as stringent as the requirements of engines and applications for mobile web-based workflow applications (e.g., [17, 18]).

Following this introduction, Section 2 provides illustrative examples to further elaborate workflow-based design and rationales behind EMWF. Section 3 provides an overview of EMWF version 1.0 [19]. Section 4 describes extensions designed to provide EMWF 2.0 (i.e., the next version of EMWF) with communication and real-time capabilities. Section 5 summarizes the paper.

## 2 Motivations and Rationales

EMWF is written in C. It provides a workflow manager and engines on Linux and Microsoft Windows CE and XP Embedded. We focus here on the relatively mature Windows versions. Hereafter, by EMWF, we mean these versions and call them EMWF 1.0. A description of the Linux version can be found in [19, 20].

## 2.1 Simple Workflow-Based Devices

Specifically, EMWF 1.0 [19] is for relatively simple embedded devices and system components that run on a processor. An example is an automatic vacuum cleaner. Its workflow-based structure is shown in Fig. 1(a). Most parts of the devices are built from workflows. We omit drivers and components that are hardwired. In the figure, the rectangular boxes represent activities. The middle dotted box encircles software activities executed by the workflow engine on a CPU. External activities are in dotted boxes labeled environment interaction and robot components. These activities are carried out by sensor devices, microcontroller and mechanical parts of the device.

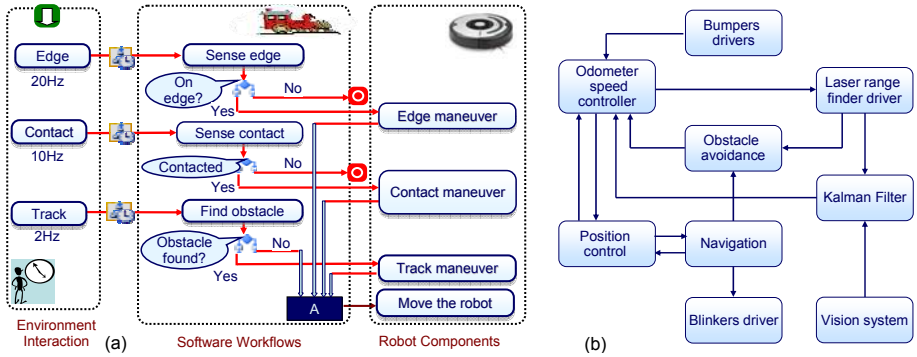



**Fig.** 1. Parts (a) Example of workflow-based device and (b) tasks in a mobile robot [21]

Table 1 lists built-in activities provided by EMWF. The symbols used to represent some of them are from Microsoft Windows Workflow foundation (WF) GUI editor [11]. Top five rows lists examples of generic built-ins, including start, stop, if else, and wait-for built-ins that are used in the device shown in Fig. 1(a). All applications need some generic built-ins. The bottom row of Table 1 lists built-ins for robotic behavior coordination (BC), including arbiter, the built-in that performs fixed-priority arbitration among edge, contact and track maneuvers in the device in Fig. 1(a). Arbiter is implemented in EMWF 1.0 with generic built-ins as described in [19]. Table 1 also lists superposition and voter activities for behavior coordination in robotic applications and push data, pull data and mode change activities for all embedded applications. These built-ins are provided by EMWF 2.0.

Table 1 Built-in activities

| | | | |
|---|---|---|---|
| Generic built-ins | Route | If else (2-way XOR split) | |
| | Split | Merge | |
| | Throw | Exception | |
| | Invoke workflow | Execute workflow | |
| | Delay /timeout | Set events / timers | |
| | Wait for events / timers / workflow triggers | | |
| | Start | Stop | While |
| Built-ins for BC | Superposition | Arbiter | Voter |
| | Mode change | Push data | Pull data |

As pointed out by [19], we can modify a workflow-based device by changing the workflows in it and/or by using different resources for activities. As an example, we can turn the vacuum cleaner in Fig. 1(a) into a navigation component by changing its workflow graph and into a toy sumo by replacing the "back and random move" activity with a "move back and hit" activity. We will return shortly to give another example to illustrate the merit of the workflow approach in this respect.

### 2.2 Messaging and Real-Time Capabilities

Many UCAADS applications rely on multiple computers, wire and wireless networks, local and remote sensors and control devices, and mobile and fixed user interfaces. Fig. 1(b) shows an example. The block diagram is from [21]. It contains some of the tasks in an experimental mobile robot called Pygmalion. If the robot were workflow based, these tasks would be implemented by workflows. According to [21], the bumpers drivers and speed controller run at 1 KHz, and the position controller runs at 50 Hz. Obstacle avoidance makes use of the vision system and laser range finder. Such a robot may also have a speech recognition system that can capture and interpret voice commands and a speaker location system that can pinpoint the speaker. Navigation and obstacle avoidance tasks may run on a processor while the vision system and speech related tasks run on a separate processor or processors, and speed and position control tasks run on yet another processor.

To support applications exemplified by Pygmalion, we extended EMWF 1.0 with a low-level message passing mechanism that implements push data and pull data built-ins in Table 1 and provides the low-level support essential for end-to-end distributed workflow management. Section 4 will describe the mechanism.

Many tasks shown in Fig. 1(b) have real-time requirements: Obstacle avoidance and position control are examples. They must complete on a timely basis for the robot to move smoothly at a required speed without bumping into obstacles. For such devices, a defect of existing workflow management systems, including EMWF 1.0, as well as existing middleware for robotic applications (e.g., [22]), is their lack of adequate real-time support. This is why we extended EMWF 1.0 with an end-to-end scheduler at the higher level and a message scheduler at the lower-level. Together with the priority-based CPU scheduler provided by the workflow manager in EMWF 1.0, the two-level scheduler enables the extended EMWF to support many well known end-to-end real-time scheduling strategies.

### 2.3 Workflow Communication

EMWF 1.0 supports event-driven, sequential workflows, but not state machine workflows. More seriously, it lacks flexible, easy-to-use facilities for invocation of workflows by workflows and data exchanges between workflows. These limitations prevent us from using it for complex devices such as iNuC (intelligent nursing cart) [8]. iNuC is a mobile system of medication administration and record keeping tools for nurses. It is self-contained: During network and hospital-wide server outage, the cart can operate stand-alone. The major component tools and system modules include a graphical user interface (GUI), authentication and authorization module (AaA),

work-time manager (WTM), intelligent monitor, alert and notification (iMAN), locker interlock mechanism (LIM), record keeper (RK), data refresher (DR) and cart user event log (CUEL). The beta version iNuC 1.0 has the traditional hardwired structure, is written in C, and runs on Microsoft Windows XP embedded.

A hospital typically needs not only full-service carts like iNuC, but also basic mobile units (BaMU). A BaMU works collaboratively with a per-patient-ward mulit-user medication station (MUMS) server [23] and relies on the server for many functions, including AaA, WTM, and iMAN functions. Building a BaMU by modifying the way modules are integrated in iNuC 1.0 would take considerably more effort than building it from the workflow-base version, called iNuC 1.5.

iNuC 1.5 runs on Microsoft Windows Workflow Foundation (WF) in .NET [11]. The left half of Fig. 2 depicts its structure. Details in the diagram are not important for our discussion here. It suffices to note that the behavior of the cart and its interaction with the user depend almost solely on the iNuC state machine workflow. We can change an iNuC into a BaMU by replacing the iNuC state machine workflow with a BaMU state machine workflow. On WF, such a reconfiguration can be done dynamically without having to restart the cart.

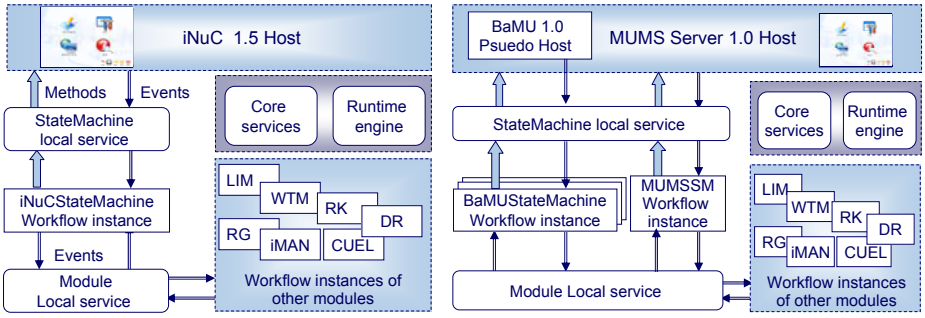



Fig.2. Workflow-based architecture of intelligent nursing cart and station software

The right half of Fig. 2 shows the workflow-based structure of a multi-user medication station server that works with BaMU to enforce bar-code controlled medication dispensing. The server can support a few remote sessions via BaMU concurrently. It may have multiple BaMU state machine workflow instances interacting via a local service (i.e., a communication facility) with BaMU pseudo host and through it with GUI's running remotely on basic mobile units. Because the workflow runtime engine does the difficult work of managing their execution, we can easily reuse component modules to built different systems.

Before moving on, we note that another advantage of workflow-based design is that the definitions of the workflows and service interfaces of a device specify clearly the device behavior and make the dependencies between components explicit. We can use the definitions as executable behavior specifications to simulate the device and user-device interactions for the purpose of assessing the usability and performance of the device as soon as the specification of the device is available. The simulation environment described in [24] is for this purpose.

WF has many shortcomings for applications such as iNuC, BaMU and Pygmalion, however. Running in .NET environment means not only large system resource demands, but also less than ideal response times for time critical functions. To

provide real-time scheduling capabilities requires replacing the default scheduling service by a custom one. This can be done in principle, but a custom scheduler in .NET environment is unlikely to give the developer control on scheduling to the degree necessary for many real-time applications. By providing services for workflow communications, EMWF 2.0 aims to provide the advantages of WF for embedded applications without its shortcomings.

## 3. Overview of EMWF 1.0

The embedded workflow definition language supported by EMWF 1.0 is called SISARL-XPDL [19]. It is an extended subset of the WfMC standard XPDL 2.0 [13]. The extension includes the built-ins activities listed in the last row of Table 1. It also includes Period and ExtendedAttributes. These elements enable the developer to specify which workflows are real-time, what their execution rates are, and what custom scheduling policy is to be used if the workflows are not to be scheduled by default on rate-monotonic basis. The SISARL-XPDL parser first translates extension elements into standard XPDL 2.0 elements and then compiles XPDL 2.0 definitions into executable workflow scripts. Workflow scripts are stored in .wfs files.

### 3.1 Engine Manager, Workflow Manager and WLA and ALA Engines

Fig. 3(a) shows the general structures of a workflow-based embedded device running on EMWF 1.0. The application components are shown as workflow instances. Major components of EMWF 1.0 are engine manager, workflow manager and workflow processor. The engine manager manages the configurations of the engine and application workflows. The workflow manager processes the workflow scripts, and the workflow processor executes activities according to the scripts.

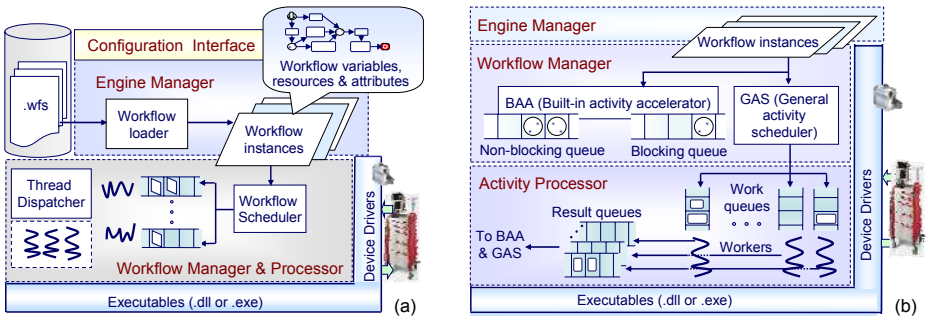



Fig. 3 Parts (a) Workflow-based device with WLA engine and (b) structure of ALA engine

The engine manager is responsible for handling user requests and managing their accesses to workflow-related definitions and optional contextual information. The developer can tune the engine via the configuration interface. Configuration parameters include the maximum numbers of threads and priority levels, and the finest resolution of timers. During initialization, the engine manager initializes and configures the engine. It then loads the .wfs files needed by all the applications for all operation modes and adaptation into memory. This enables the workflow manager to

allocate memory dynamically for all instances of activities and workflows during initialization in order to keep the memory footprint of the applications small. A negative consequence is that when .wfs files are added and removed, the engine must be restarted for the configuration changes to take effect.

EMWF 1.0 offers two different multi-threaded workflow processors on Microsoft Windows CE and XP Embedded. They are the WLA (workflow-level assignment) engine and the ALA (activity-level assignment) engine. The lower half of Fig. 3(a) depicts the structure of the WLA engine. In this case, each thread is dedicated to execute a workflow: The workflow manager attaches a thread to each workflow instance when it initializes the instance and schedules the thread to execute all activities in it. The thread inherits the priority of the workflow instance.

Fig. 3(b) depicts the structure of an ALA engine. The term general activity in the figure refers to activities provided by the developer, i.e., not built-ins of the engine. An ALA engine uses worker threads to execute activities as work items: The workflow manager maintains a FIFO queue per priority for queuing work items and assigns at least a thread per queue. The thread (or threads) serving a queue executes at the priority of the queue. Threads in the workflow manager and processor interact in more or less the leader/followers pattern. Worker threads are followers. For a device that has no blocking built-in activities, the workflow manager may have just one leader thread. The leader processes workflow scripts, wraps ready (i.e., enabled) general activities as work items and inserts them in priority queues according to their priorities to be executed by follower threads, and supervises their completion. With a few exceptions, built-ins are simple. The leader executes itself built-ins as they become enabled, which in turn leads to more general activities be ready and queued. These functions of the leader are depicted as general activity scheduler and built-in activity accelerator in Fig. 3(b).

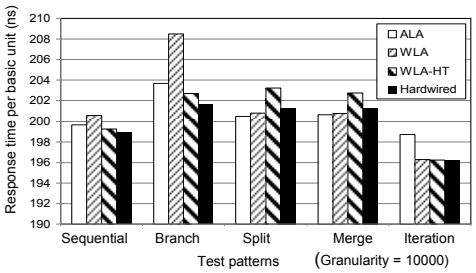
### 3.2 Relative Merits

Since a thread assigned to a workflow in a WLA engine executes both general activities and built-ins in it, most of the transitions between activities incur no context switch. In an ALA engine, however, every transition from one general activity to another incurs at least one context switch.

The current versions of WLA engines do not support varying priority within workflows, just like Microsoft .Net WF, which also uses the WLA strategy. Priority increment of an activity with respect to the base priority of the workflow containing it is ignored by the engine. In contrast ALA engine executes activities at their own priorities and thus supports varying workflow priority naturally. Coherent time order for all tasks within a device is often expensive to achieve. Using an ALA engine, this can be accomplished with no additional cost by having the engine use a single thread to handle all timing events.

We have measured the runtime performance of WLA and ALA engines using several benchmark workflow-based workloads running on a 3.4 GHz Pentium 4 and Windows CE 6.0 platform. Each workflow-based workload is characterized by the number of workflows, a test pattern and the granularity of activities. For each workflow-based load, we also ran equivalent hardwired code on Windows CE 6.0 without the engine. The difference between the response times for the two versions

gives us an estimate of the runtime overhead introduced by the engines. The chart in the left half of Fig. 4 shows the kind of performance data obtained from this study. In this case, all activities are software activities with the same granularity. (The function for each activity calls a random number generator 10,000 times.) The WLA-HT engine is an enhanced WLA engine which uses a helper thread to create workflow instances. We can see that when activities are of sufficiently large granularity, as in the case shown here, runtime overheads of ALA and WLA-HT engines are acceptable. The disadvantage of ALA engine in terms of context switches becomes evident when activities are so small that the number of extra context switches is in order of 1000 per workflow. Similar measurements performed on Windows XP Embedded points to similar conclusions. Details on this study can be found in [19].




Performance under Hartstone PH-series load

| | Hardwired | WLA | ALA |
|---|---|---|---|
| Task 1 | 26 (2, 33) | 31 (2, 29) | 27 (9, 18) |
| Task 2 | 51 (8, 43) | 62 (4, 58) | 59 (17, 42) |
| Task 3 | 93 (44, 49) | 125 (8, 117) | 116 (28, 88) |

Task routine: Array processing

| | | | |
|---|---|---|---|
| Task 1 | 35 (2, 33) | 31 (1, 30) | 31 (3, 28) |
| Task 2 | 69 (4, 65) | 62 (1, 61) | 62 (7, 55) |

Task routine: Standard function calls

Fig. 4 Performance data from [19] and from a test using Hartstone workload

The tables in the right half of Fig. 4 list data on engine performance taken by running Hartstone PH-series [25] workload on a 1.73 GHz Genuine Intel CPU and Windows Standard XP SP3 with 1-ms timer resolution. The workload contains 5 periodic, harmonic tasks, and they were scheduled rate-monotonically. Again, we ran an equivalent hardwired code without the engine: In the hardwired version, each task is a single thread that executes the task routine periodically. In the workflow version, the routine is executed as an activity by the engine. We used two task routines: One processes a small integer array and the other makes standard function calls. The former is shorter (i.e., gives us a smaller granularity) than the latter.

We started a series of tests by setting the highest rate (i.e., the rate of Task 5) at 32 Hz and for subsequent tests, increased the rate until it reaches the highest possible rate of 500 Hz. For each test, we ran the tasks for 10 seconds and recorded for each task the number $M$ of misses (i.e., the number of periods in which the task ran but completed late) and number $S$ of skips (i.e., the number periods in which the task was not scheduled). The number $T$ of timing faults is the sum $M + S$. We use the notation $T$ ($M$, $S$) to capture these numbers.

When the highest rate was 32Hz and task routine was standard function calls, Task 5 with rate 32Hz had 4(2, 2) and 75(9, 66) timing faults in 320 periods when the tasks were executed by the WLA engine and ALA engine, respectively. On the ALA engine, each task incurs a context switch each period, while there is no context switch on the WLA engine and for the hardwired code. This is a reason that Task 5 incurred a larger number of timing faults when executed on the ALA engine. All other tasks incurred no timing faults. When the task routine was array processing, all tasks, including Task 5, had no timing fault. When the highest rate was 500Hz, the system was overloaded. The tables in Fig. 4 list the numbers of timing faults of lower priority tasks, showing that the engines performed comparably under such condition.

## 4 Extensions in EMWF 2.0

We presented earlier rationales for extending EMWF 1.0 with two kinds of extensions. First, messaging passing and end-to-end scheduling mechanisms are essential parts of workflow management for time-critical distributed and networked applications. These mechanisms have already been added to EMWF 1.0 on Windows XP Embedded. The second kinds of extensions include capabilities and tools that are needed to make EMWF 2.0 not only a middleware ideally suited for a wide range of user-centric automation and assistive devices from simple devices like automatic vacuum cleansers to complex ones like iNuC and Pygmalion, but also an excellent design, development and evaluation environment for them.

### 4.1 Messaging and End-to-End Scheduling Mechanisms

Fig. 5 (a) illustrates how push data and pull data built-in activities are implemented. Without loss of generality, suppose that there are two workflows: Workflow 1 contains a push data built-in while Workflow 2 contains two pull data built-ins. When encountering a push data activity, the workflow manager dispatches a Send operation to move the data to be sent into a send buffer and queues a Send work item in one of the prioritized work queues according to the priority of the Send operation. The work item is executed by a worker thread at the priority of the queue. Depending on whether the receiving workflow runs locally or remotely, the Send work item either invokes an IPC or a Winsock send API function to move the data from the send buffer to the receive buffer. This and other data transfers are depicted by dashed arrows. Once arrived, the data waits in the receive buffer until the workflow manager of the receiving workflow encounters a pull data activity, depicted as a box with a block arrow pointing into the box. The manager then queues a Receive work item to move the data from the receive buffer to the space of the receiving workflow. Specific work to be done by Receive is application dependent. The developer can specify how it is to be done via a parameter of Receive.

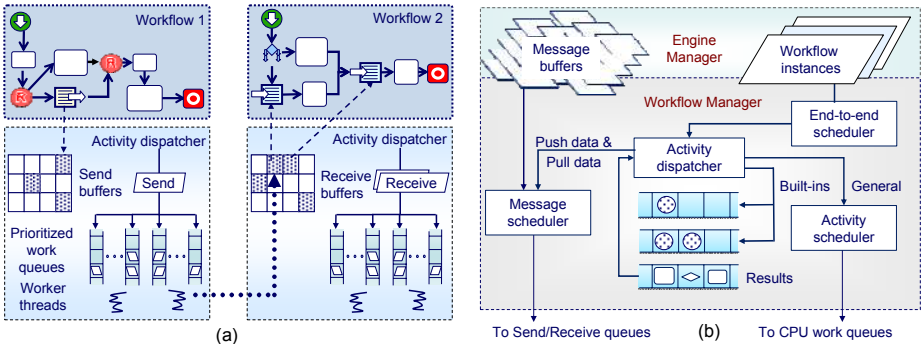



Fig. 5. Real-time extensions (a) Send/Receive mechanism and (b) two-level scheduler

Rather than a library of schedulers for different types of applications, EMWF provides a general two-level, end-to-end scheduling mechanism with which many well-known and commonly used real-time scheduling schemes can be easily

implemented. Fig. 5(b) shows the structure of the mechanism for ALA engine. The description for it is by and large applicable to the case of the WLA engine.

The assumption here is that the system is statically configured: In other words, components of each application are partitioned and assigned to processors (and other types of resources) at initialization time. They are migrated among processors only when reconfiguration becomes necessary. The extended attributes of each workflow instance contains information on the CPU used for each software activity and resources required by external activities. If the workflow has real-time requirements, the extended attributes also specify a finite end-to-end deadline, estimated worst-case execution time of each activity and so on. The end-to-end slack time of a workflow instance is the difference between its end-to-end deadline and the total worst-case execution time of the longest chain of activities in the workflow.

The workflow manager calls the high-level end-to-end scheduler when a new workflow instance becomes ready for execution. Currently, the only task performed by the scheduler is to distribute the available end-to-end slack time of each workflow (or chain of workflows) to activities or chains of activities according to specified algorithm(s). Both low-level schedulers (i.e., activity scheduler for CPU scheduling and message scheduler for network traffic) support fixed priority scheduling.

EMWF 1.0 requires the developer to partition the application(s), assign processors (and resources) to individual partitions and provide algorithms for distributing end-to-end slack and priority assignments for CPU and network scheduling. EMWF 2.0 will provide admission control and resource management tools to support this work.

### 4.2 Service Interfaces

Earlier in Fig. 2, we introduced the term local service without explaining what it means and what the components bearing this name do and why EWMF 2.0 needs to provide similar support. Simply put, from the point of view of a workflow-based application, a local service (e.g., StateMachineService or ModuleService in Fig. 2) of Window .NET WF [11] is a set of interface functions that are defined and implemented by the developer of the application. To illustrate, the right half of Fig. 6 shows the interface functions of the local services in Fig. 2. Workflow instances in the application communicate with non-workflow component(s) and with each other using these interface functions.

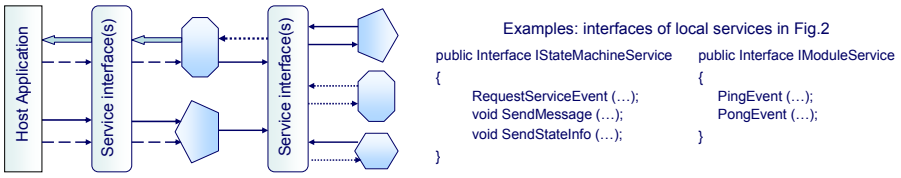



Fig. 6 Interaction via service interfaces

EMWF service interfaces resemble closely to local services of WF. The kind of assistance EMWF 2.0 will provide to service interfaces is similar to what WF does for local services: In particular, each service interface is identified by its type. After the developer has declared a service interface type, implemented a service interface of the type and have the application created and registered the service interface with the

workflow management runtime during initialization time, workflow instances in the application can query the workflow manager for functions provided by the service and make use of the functions for communication and invocation.

The diagram in Fig. 6 illustrates ways workflow instances and non-workflow components communicate and interact, making use of service interfaces. Rectangular shapes represent service interfaces, host applications and non-workflow components. Polygons represent workflow instances of the same or different type. Bold arrows represent callbacks and simple arrows represent raises and deliveries of events.

Specifically, workflow instances invoke each other and deliver results to each other by raising events. In essence, the system of service interfaces serves as a router, routing each event raised by a workflow instance to one or more workflow instances as specified by the parameters of the raise-event interface function. In a distributed system, the workflow manager helps to track the locations of all workflow instances and thus relieves the developer from the burden of this work. In addition to events, workflow instances can communicate with the host application and other non-workflow components via callbacks.

We note that when a caller raises an event to invoke a workflow instance, the event handler is executed by the thread dispatched to execute the instance. When the workflow instance responds to a caller via a callback function, the function is also executed by this thread. It is important to follow the principle of this pattern when the caller must be responsive. This is the case of the host in iNuC. The host application contains a single thread. After initialization, it becomes the GUI thread. Using the communication pattern depicted by Fig. 6, the GUI thread is never tied up doing time consuming work. It is almost always free and ready to respond to user action.

## 5 Summary

This paper describes the current status and future plans for EMWF. The workflow management system enables diverse embedded devices to be built from activities and workflows components. Its light-weight engine integrates the components at runtime by executing the components in manners specified by the developer. Configurability is one of primary factors that motivated us to build the embedded workflow framework. The proof-of-concept version EMWF 1.0 provides light-weight engines on Microsoft Windows CE and XP Embedded platforms. They are released under GPL license. This version has many limitations. We are working to remove them while adding the features and capabilities described in the previous section and maturing the middleware into EMWF 2.0.

**Acknowledgments** This work was partially supported by the Taiwan Academia Sinica thematic project SISARL.

## References

1. Tsai, P. H., C. Y. Yu, W. Y. Wang, J. K. Zao, H. C. Yeh, C. S. Shih, and J. W. S. Liu, "iMAT: Intelligent Medication Administration Tools," *Proc of IEEE Healthcom*, July 2010.

2. Wang, W. Y., J. K. Zao, P. H. Tsai, and J. W. S. Liu, "Wedjat: A Mobile Phone Based Medication Reminder and Monitor," *Proceedings of the 9th IEEE International Conference on Bioinformatics and Bioengineering*, June 2009.
3. Tsai, P. H., C. Y. Yu, C. S. Shih and J. W. S. Liu, "Smart Medication Dispenser: Architecture, Design and Implementation," Technical Report No. TR-IIS-008-010, Institute of Information Science, Academia Sinica, 2008.
4. Chou, T. S. and J. W. S. Liu, "Design and Implementation of RFID-Based Object Locator," *Proceedings of IEEE 2007 International Conference on RFID Technology*, March 2007.
5. Hsu, C. F., H. Y. M. Liao, P. C. Hsiu, C. S. Shih, T. W. Kuo, and J. W. S. Liu, "Smart Pantries for Homes," *Proceedings of IEEE International Conference on SMC*, Sept. 2006
6. Forizzi, J. and C. DiSalvo, "Service Robots in Domestic Environment: a Study of Roomba Vacuum in the Home," *Proc. of ACM/IEEE International Conference on HRI,* March 2006.
7. Kaneshige,Y., M. Nihei, and M. G. Fujie, "Development of New Mobility Assistive Robot for Elderly People with Body Functional Control," *Proceedings of IEEE/RAS-EMBS*, February 2006.
8. Tsai, P. H., Y. T. Chuang, T. S. Chou, C. S. Shih, and J. W. S. Liu, "iNuC: An Intelligent Mobile Medication Cart," *Proceedings of the 2nd International Conference on Biomedical Engineering and Informatics*, October 2009.
9. SpeciMinder hospital delivery robot, http://www.youtube.com/watch?v=lJ7RnTAYZ-8
10. TUG, pharmacy delivery robot, http://hfrp.umm.edu/tug/tug_main.htm
11. Bukovics, B. *Pro WF: Windows Workflow Foundation* in .Net 4.0, Apress 2009.
12. WfMC: Workflow Management Coalition, http://www.wfmc.org/
13. XPDL (XML Process Definition Language) Document, http://www.wfmc.org/standards/docs/TC-1025_xpdl.2.2005-10-03.pdf, October 2005
14. BPEL (Business Process Execution Language), http://en.wikipedia.org/wiki/BPEL
15. Open Source Java XPDL editor, http://www.enhydra.org/workflow/jawe/index.html.
16. Enhydra Shark, http://forge.objectweb.org/projects/shark
17. Pajunen, L. and S. Chande, "Developing workflow engine for mobile devices," *Proc. of IEEE International Enterprise Distributed Object Computing Conference,* 2007.
18. Hackmann, G., M. Haitjema, C. Gill, and G. C. Roman, "Silver: A BPEL workflow process execution engine for mobile devices," in A. Dan and W. Lamersdorf, Ed., ICSOC 2006.
19. Chou, T. S., S. Y. Chang, Y. F. Lu, Y. C. Wang, M. K. Ouyang, C. S. Shih, T. W. Kuo, J. S. Hu and J. W. S. Liu, "EMWF for Flexible Automation and Assistive Devices," *Proceedings of IEEE RTAS*, April 2009.
20. Chang, S. Y., Y.-F. Lu, T. W. Kuo, and J. W. S. Liu, "The Design of a Light-Weight Workflow Engine for Embedded Systems," *Proceedings of RTSS Workshop on Software and Systems for Medical Devices and Services*, December 2007.
21. Brega, R., N. Tomatis, K. O. Arras, "The Need for Autonomy and Real-Time in Mobile Robotics: A Case Study for X0/2 and Pygmalion," *Proceedings of IEEE/RSJ Internationa*l *Conference on Intelligent Robots and Systems*, October 2000.
22. Robot Standards and Reference Architecture, http://wiki.robot-standards.org/index.php/Middleware
23. J. W. S. Liu, C. S. Shih, C. T. Tan and V. J. S. Wu, "MeMDAS: Medication Management, Dispensing and Administration System," m-Health Workshop, IEEE HealthCom 2010.
24. T. Y. Chen, C. H. Chen, C. S. Shih, J. W. S. Liu, "A Simulation Environment for the Development of Smart Devices for the Elderly," *Proceedings of IEEE International Conference on Systems, Man and Cybernetics,* October 2008.
25. N. H. Weiderman1 and N. I. Kamenoff, "Hartstone Uniprocessor Benchmark," *Journal of Real-Time Systems*, December 1992.