EMWF for Flexible Automation and Assistive Devices

T. S. Chou, Y. C. Wang and J. W. S. Liu
Institute of Information Science
Academia Sinica, Taiwan
{tchou, wych, janeliu}@iis.sinica.edu.tw

Dept. of Computer Science and Information Engineering
National Taiwan University, Taiwan
{ktw, cshih}@csie.ntu.edu.tw

J. S. Hu
Department of Electrical and Control Engineering
National Chiao-Tung University, Taiwan
jshu@cn.nctu.edu.tw

Abstract—This paper describes an embedded workflow framework (EMWF) that enables flexible personal and home automation and assistive devices and service and social robots (collectively referred to as SISARL) to be built on workflow architecture. The process definition language supported by EMWF is called SISARL-XPDL. It consists of a subset of the WfMC standard XML Process Definition Language (XPDL) 2.0, together with elements that implement common mechanisms for robot behavior coordination. EMWF provides workflow engines for Linux and Windows CE platforms. The engines are written in C in order to keep their memory footprint and runtime overhead small. Performance data show that the overheads introduced by the engine and workflow data are tolerable for most SISARL devices.

Keywords—embedded Middleware; workflow architecture, component-based design, automation and assistive devices

I. INTRODUCTION

Thanks to steady advances in embedded systems and robotic technologies over the years, a diverse spectrum of smart devices targeted for elderly individuals and functionally limited users have become economically feasible today. Examples include personal and home automation and assistive devices (e.g., automatic medication dispenser, smart storage pantry, robotic housekeeping aids, and mobility assistants [1-11]) and service and social robots (e.g., object fetchers and delivery robots, robotic pets, and intelligent physical-therapy companions [12-17]). We call them collectively SISARL (Sensor Information Systems for Active Retirees and Assistive Living) [18]. Such devices are increasingly more essential for an increasingly larger aging segment of the world population.

As tools designed to help their users stay well and live independently or to improve the quality and reduce the cost of medical and long-term care, SISARL must be flexible: A flexible device can be easily configured to work with a variety of sensors and controllers, rely on different support infrastructures and operate in different environments. It can be easily customized to suit its user. Furthermore, the device can adapt to changes in its user’s needs and skills over the years while it is in use.

This paper describes an embedded workflow framework (EMWF) as architectural foundation for the design and implementation of flexible SISARL. Even today, all but the simplest SISARL devices are handcrafted, and handcrafted devices are difficult and costly to configure and customize. EMWF was motivated by this fact. Its primary purpose is to reduce the level of expertise and effort required to design and implement from reusable components flexible SISARL devices in general and behavior-based robots in particular.

As its name implies, EMWF is based on the try-and-true workflow approach [19] that is widely used for automation of business processes of enterprises worldwide. Components of an application based on this approach are workflows. Each workflow is composed from elementary steps called general activities (or simply activities). In workflow-based embedded devices, some activities are done by executable code running on a CPU; others are by hardware components. A semi-automatic device also contains user activities carried out manually. The orders and conditions under which activities in a workflow are executed, the resources used for their execution, and interactions and communications among activities are specified either by textual workflow definitions or graphically by one or more workflow graphs. Some workflow graphs resemble task graphs used to represent workloads in real-time systems. Each node represents an activity. Each directed edge represents a transition from the source activity to the sink activity, and consequently a precedence relation between the activities. Each workflow has a start and one or more stop. They are special activities that have no predecessor and successor, respectively, in the workflow. Many workflows in embedded devices are event-driven. Such a workflow is often defined as a state machine in which device and user activities cause the machine to transition from state to state.

A key element of a platform for workflow applications is the workflow engine (or engine for short). The engine executes activities implemented by software components and commands and coordinates activities carried out by the user(s) and hardware components. In addition, the engine provides and executes built-in activities that start and stop workflows, sequence and synchronize general activities in each running workflow and facilitate their communications as specified by the definition of the workflow. By managing control and data flows among general activities, scheduling and allocating resources among competing workflows and enforcing rules and policies on behalf of the applications, the engine dynamically integrates the application components that are implemented by workflows.
A reason for the wide adoption of the workflow approach for automation of business processes (e.g., [20-31]) is the relative ease with which workflow-based applications can be designed and implemented. A developer can design and implement a new application or configure and customize an existing one by supplying or modifying definitions of workflows in it and components for activities. Existing standard process definition and execution languages (e.g., [20-22]), together with tools (e.g., [23, 24, 27]) for editing workflow definitions and for parsing and building them, significantly reduce the effort to do these tasks.

EMWF enables us to adopt the workflow-based architecture for SISARL by providing a language for the specification of embedded workflow applications and choices of light-weight engines for executing and managing them on embedded platforms. Specifically, the workflow process definition language supported by EMWF is called SISARL-XPDL. It contains a subset of the WiMC (Workflow Management Coalition) standard XML Process Definition Language (XPDL) 2.0 [20]. The subset is augmented with built-in activities specifically for coordination of robot behaviors. Workflow definitions can be parsed either first into intermediate scripts in C or compiled directly into binary workflow scripts (wfs) for execution. Currently, EMWF provides versions of workflow engines for Linux and Windows CE. The engines are written in C in order to keep the memory footprint and runtime overhead introduced by the engine and scripts small. We are using EMWF as a test bed for the experimentation with the workflow approach to building SISARL devices and for evaluation of the merits and shortcomings of the approach.

Following this introduction, Section II provides an overview of closely related work. Section III describes the major components of EMWF and a general structure of embedded devices built within the framework. Section IV describes the design, architecture and implementation of the engine software. Section V describes the elements of SISARL-XPDL and provides illustrative examples of workflow scripts. Section VI describes data on runtime overheads for different versions of EMWF engines. Section VII summarizes the paper and presents future work.

II. RELATED WORK

As stated earlier, SISARL-XPDL is a variant of the WiMC standard XPDL 2.0 [20]. The fact that XPDL process definitions are interchangeable among modeling tools and execution engines is important for us since we also use workflows to define device and user behavior models and device-user collaborations for simulation and evaluation purposes [1, 2]. Modern workflow engines and management environments (e.g., [23 - 31]) can handle automated and manual activities in an integrated way. EMWF is built on the foundation established by them.

Unlike engines designed to run in J2EE or .NET environments, EMWF engines are scaled down to fit embedded platforms. In this respect, they resemble workflow engines for mobile devices such as the ones described in [29-31]. Existing mobile workflow management systems are for web-based applications: They enable business processes (e.g., loan approval and insurance claim processing) defined in BPEL [22] to run on handheld devices. In contrast, EMWF engines are designed solely for embedded applications containing workflows that open and close drawers in an intelligent medication cart, guide a service robot around obstacles, and so on. Many embedded workflows run at high rates and interact closely with hardware devices. BPEL and other business process execution languages are not suitable for them. Typical SISARL devices are not as severely power and size constrained as cell phones and PDA’s. Consequently, energy consumption and memory footprint requirements for EMWF engines and workflow applications are not as stringent as the requirements of engines and applications for mobile workflow management.

EMWF has the same goal as many efforts by robotic community aiming at making the creation and modification of robotic device software and systems easier. As examples, building blocks and development environments provided by Microsoft Robotic Studio and LEGO®MINDSTORMS™ reduce the skills and efforts that are required to design and build some robots [32, 33]. Research projects such as CARMENO, CLARA™, MARIE, MIRO, ORCA, OROCOS, and Player [34-40] aim to provide component-based software architectures and tools for building robotic software systems from reusable modules. Embedded Software Architecture for Intelligent Robots (ESAIR) [41] is an object-oriented environment that supports component-based design and implementation of embedded software of behavior-based robots. By providing a pluggable component interface and a discovery mechanism, ESAIR enables devices and software modules to be plugged in and removed from the system without having to redesign and implement the robot. Many design choices of EMWF were influenced by ESAIR.

III. ARCHITECTURE AND MAJOR COMPONENTS

This section first describes architecture of workflow-based embedded devices focusing on the applications. It then describes the devices from the perspective of the engine.

A. Application Structure

In a workflow-based SISARL device, most parts are built from activities and workflows. In general, a device contains both non-embedded and embedded workflows. Hereafter, we focus solely on embedded workflows. In such a workflow, some activities are executed on CPU(s) by the engine. We call them software activities when we need to be specific. An embedded workflow also has external activities: Some external activities are user activities carried out manually. Others are by sensor devices, special-purpose hardware, and mechanical parts.

As a rule, functions and executables for general software activities must not make blocking calls, and each has a single entry and a single exit. In other words, all activities that alter the flow path and timing of workflows are built-ins. This requirement is not unduly restrictive for typical embedded workflows. In behavior-based robotic devices such as the ones targeted by ESAIR [41], activities are behaviors. With
one exception, they meet this requirement naturally. The exception is when behaviors share sensor devices; a behavior needs to send the data it reads to receiving behavior(s). Having the sender blocked, waiting for the receiver to be ready is not acceptable. ESAIR provides a middleware component, called (behavior) supervisor, to facilitate their communication: A behavior can send data asynchronously to the supervisor. The supervisor holds the data and delivers the data to the receiver when the receiver is ready. The EMWF workflow engines also provide this kind of service.

Table 1 lists examples of built-in activities. Some of the symbols here are from Windows Workflow Framework [24]. A block arrow directed to or from an activity represents multiple transitions (edges) in or out of the activity.

<table>
<thead>
<tr>
<th>TABLE I. EXAMPLE OF BUILT-IN ACTIVITIES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Generic built-ins</strong></td>
</tr>
<tr>
<td>Route</td>
</tr>
<tr>
<td>Split</td>
</tr>
<tr>
<td>Merge</td>
</tr>
<tr>
<td>If-else (2-way XOR split)</td>
</tr>
<tr>
<td>Exception</td>
</tr>
<tr>
<td>Invoke workflow</td>
</tr>
<tr>
<td>Execute workflow</td>
</tr>
<tr>
<td>Delay /timeout</td>
</tr>
<tr>
<td>Set events / timers</td>
</tr>
<tr>
<td>Wait for events / timers / workflow triggers</td>
</tr>
<tr>
<td>Start</td>
</tr>
<tr>
<td>Stop</td>
</tr>
<tr>
<td>While</td>
</tr>
<tr>
<td><strong>Built-ins for BC</strong></td>
</tr>
<tr>
<td>Superposition</td>
</tr>
<tr>
<td>Arbiter</td>
</tr>
<tr>
<td>Voter</td>
</tr>
<tr>
<td>Mode change</td>
</tr>
<tr>
<td>Push data</td>
</tr>
<tr>
<td>Pull data</td>
</tr>
</tbody>
</table>

The top half of the table lists generic built-ins required by typical embedded applications. The names of some of them (e.g., if-else and wait) more or less tell what they do. The more common names of built-in activities Split and Merge listed in the first row are branch and join, respectively. The former branches an activity into multiple successor activities, and latter joins multiple predecessor activities into a single successor. They are special cases of the Route activity. A Route can have arbitrary numbers of incoming and outgoing transitions. When used with transition restrictions, Route can implement arbitrary complex flow logic, including combinations of conditional XOR and AND merges of incoming transitions and splits of outgoing transitions. We will return to provide an illustrative example in Section V and discuss the behavior coordination (BC) activities listed in the bottom half of the table. BC activities are specifically for behavior-based robotic applications.

As an example, Fig. 1 shows the workflow-based structure of an intelligent medication cart, highlighting its application components. The device is designed to assist its user in administration of medications to patients under the user’s care. The cart has microcontrollers and motors for propulsion, an ultrasound range finder and other sensors for guidance and navigation, a RFID reader and a bar-code scanner for medication and patient identification, and so on. Some device drivers, like the workflow engine itself, are not built from workflows; these parts are shown in dark color in the figure. The other functions are provided by workflows components.

![Figure 1. Structure of a workflow-based device](image)

Fig. 2 shows the guidance and propulsion module of the medication cart in its entirety to highlight the distinguishing characteristics of embedded workflows. In this and subsequent workflow graphs, we use rectangular boxes to represent general activities of the application. When there is no need to be specific, we use dotted circles to represent all built-in activities. External activities are in dotted boxes labeled environment interaction and cart components. This module operates automatically; it has no user activities.

![Figure 2. Guidance and propulsion workflows](image)

**B. Engine Structure**

Fig. 3 shows a different view of the workflow-based structure of a device. Focusing on the engine, the figure depicts applications simply as workflow scripts. We will provide further details on EMWF internal representation of workflow definitions and attributes in Section V. The XPDL term participant in the callout at upper right corner refers to a resource that is managed and allocated to workflows by the engine. Participants required by workflows are declared in
workflow definitions. A participant may be a hardware device (e.g., bar-code scanner, motor, sensor and even a CPU), a person, and so on. When assigned to do the work, such participants carry out external activities. Participants required by software activities include library functions, interfaces and shared resources.

The engine has three major components: engine manager, workflow manager, and workflow processor. We focus on the engine manager here and defer the descriptions of workflow manager and processor to the next section. The engine manager manages the configurations of the engine and application workflows. It is also responsible for handling user requests and managing their accesses to workflow-related definitions and optional contextual information.

Specifically, the engine manager in Fig. 3 is used for development within EMWF. A developer can tune the engine via the configuration interface by changing its configuration parameters, which include the maximum numbers of threads and priority levels, and the finest resolution of timers. The engine manager here has SISARL-XPDL preprocessor and XPDL parser locally, as shown in the figure, or can access the tools remotely. These tools make it possible for the developer to add new workflows defined in terms of SISARL-XPDL, translate them into standard XPDL, and parse XPDL definitions into workflow scripts. For now, the term workflow script specifically refers to binary executable workflow definitions. They are stored in .wfs files. We show these tools and .s_xpdl and .xpdl files of workflow definitions in dotted shapes to indicate the fact that a typical device, even one as complex as an intelligent medication cart, do not have these tools. By requiring XPDL definitions to be parsed off-line, the engine does not incur the substantial memory space and power required to run the tools.

During initialization, the engine manager initializes and configures the engine. It then loads the .wfs files of the applications into memory to be processed by the workflow manager. The current versions of EMWF engines allow .wfs files to be added and removed, but the engine must be restarted for the configuration changes to take effect. This means that all the workflows a device needs to operate in multiple modes or to adapt while running must be in the system and initialized before the device starts to run.

C. Runtime Data

Fig. 4 shows the structures of runtime data on activity, workflow and package types and instances maintained by the engine. Most fields of the structures are self-explanatory. We omit data types in order to save space when there is no need to be specific. The XPDL term package means a container for grouping entities common in multiple workflow definitions to avoid duplicate definitions.

![Figure 3. Major engine components](image)

The workflow manager processes the workflow scripts, and the workflow processor executes activities according to
the scripts. These components form the core of the engine and contribute the bulk of memory footprint and context switch overhead of the engine. Needless to say that it is important to keep these overheads small.

A. Design Rationales

We ruled out having single-threaded workflow processor by default even through having a single thread execute all software and built-in activities is a way to minimize context switch and synchronization overheads. Rather, the maximum number of threads is a configuration parameter of EMWF engines, allowing the developer of a device to choose the configuration best for the device.

1) Alternative Structures: There are two strategies to structure a multi-threaded workflow processor:

- Workflow-level assignment (WLA): Each thread is dedicated to a workflow.
- Activity-level assignment (ALA): Activities from multiple workflows are queued as work items and executed by worker threads serving the queues.

We call engines using these strategies WLA engine and ALA engine, respectively.

Existing workflow engines typically used the WLA strategy. The Linux EMWF engine, called LIWWE (Light-Weight Workflow Engine) in [42], uses the WLA strategy as it suits the Linux thread model and APIs.

We implemented both the WLA strategy and the ALA strategy on Windows CE. The ALA engine can take better advantage of the Windows thread model and APIs. We implemented the WLA strategy on Windows CE so we can make fair comparison of the alternatives strategies.

2) Pros and Cons: We note that in a WLA engine, a thread assigned to a workflow executes both general and built-in activities in it. Hence, most of the transitions between activities incur no context switch. Another advantage of this strategy is that the workflow manager does not need to handle blocking built-in activities (e.g., delay and wait in Table 1) specially. It can simply let the thread wait when a workflow processing thread executes such a built-in. It is expensive for threads to change priority, however. This is why the current versions of WLA engines do not support varying priority within workflows: Priority increments of activities are ignored.

As we will see shortly that in an ALA engine, varying priority in a workflow is accomplished at no cost by having threads of differing priorities execute activities in it. Similarly, coherent time order for all tasks within a device is accomplished naturally with no additional cost by having all timing events handled by a single thread. On the other hand, every transition from one general activity to another incurs at least one context switch. As we will see in Section VI, this is a disadvantage when compared with WLA engines.

3) Memory Allocation: We note that many low-level embedded components (e.g., the workflows in Fig. 2) have stringent timing requirements. This is why we try to keep runtime overheads low, sometimes at the expense of memory footprint. As stated earlier, the engine manager loads .wfs files of all workflows needed for all modes and adaptation during initialization. This allows the workflow manager to dynamically allocate memory for all instances of activities and workflows during initialization.

B. WLA Engines

Again, the workflow manager creates and initializes threads needed to execute software and built-in activities when the engine is initialized, a group of POSIX threads in the Linux engine and user-mode threads in the Windows CE versions. All threads execute at fixed priorities. Fig. 5(a) depicts the structure of LIWWE and the WLA version(s) of the Windows CE engine. The dotted box below the engine manager encircles the workflow manager and processor.

![Figure 5. WLA and ALA engine structures](image)

1) Basic Version: The workflow manager attaches a thread to each workflow when it initializes the workflow and schedules the thread to execute both general and built-in activities in the workflow. The thread inherits the priority of the workflow. A workflow may contain Split and Merge activities. When a split occurs, the manager selectively attaches additional threads to execute the successors. When multiple threads join for a merge activity, the last thread that reaches the merge executes the merge operation.

When a thread executes an end activity, the manager detaches it from the workflow and returns it to the thread pool. On the other hand, after a thread executes the EndTrigger of a workflow, the workflow manager may terminate the thread and other threads attached to the workflow since the activity signals the termination of the workflow process.

2) Enhanced Version: Before performance data became available, we conjectured that since the number of context
The work for that flow path, it continues to check for completion by the completion of the built-in activity. After it completes the built-in activity and queues the activity or activities enabled upon the completion of each general activity by setting a priority order, starting from the highest priority. For each completed activity it finds, the leader executes the successor priority event. When awakened by the event, the leader processes the item in the queue at the priority of the activity, which is the sum of the priority of the workflow and priority increment of the activity. In other words, ready activities are executed according to their priorities with ties among equal priority activities broken in FIFO order.

Threads in the workflow manager and processor interact in more or less the leader/followers pattern [43]. For a device that has no blocking built-in activities, the workflow manager may have just one worker thread dedicated to execute work items in each queue at the priority of the queue. When a general activity is ready to be executed, the workflow manager wraps it in a work item and places the item in the queue at the priority of the activity, which is the equal to the sum of the priority of the workflow and priority increment of the activity. In other words, ready activities are executed according to their priorities with ties among equal priority activities broken in FIFO order.

As we will see shortly, in an ALA engine, only one thread needs to access instance data. To give WLA engines the same advantage, we implemented an enhanced version of WLA engines, called WLA with helper thread (WLA-HT) engine. The workflow manager uses a helper thread to create instances on behalf of workflow processing threads. With this enhancement, the overall runtime overhead of the WLA strategy is indeed smaller in most cases.

C. ALA Engine

Fig. 5(b) shows the structures of the workflow manager and processor in the ALA engine. A configuration parameter is the number of priorities the engine supports. The workflow manager maintains a FIFO queue per priority for queuing work items, and has at least one worker thread dedicated to execute work items in each queue at the priority of the queue. When a general software activity is ready to be executed, the workflow manager wraps it in a work item and places the item in the queue at the priority of the activity, which is the equal to the sum of the priority of the workflow and priority increment of the activity. In other words, ready activities are executed according to their priorities with ties among equal priority activities broken in FIFO order.

Threads in the workflow manager and processor interact in more or less the leader/followers pattern [43]. For a device that has no blocking built-in activities, the workflow manager may have just one leader thread and let the leader execute at the highest priority. The leader processes workflow scripts, queues ready general activities as work items to be executed by follower threads (i.e., worker threads), supervises their completion, and executes built-in activities, which in turn leads to more general activities be queued. With a few exceptions, built-in activities are simple.

As depicted by the figure, they are not queued as work items as a rule. The leader executes them itself; this allows most built-in activity functions to be in-line. These functions of the leader are depicted as general activity scheduler and built-in activity accelerator in Fig. 5(b).

Specifically, each worker thread signals the leader thread upon the completion of each general activity by setting a completion event. When awakened by the event, the leader checks the result queues for completed general activities in priority order, starting from the highest priority. For each completed activity it finds, the leader executes the successor activity and queues the activity or activities enabled by the completion of the built-in activity. After it completes the work for that flow path, it continues to check for completed general activities and serves them until it finds no more completed general activities waiting for its attention. It then resets the completion event and returns to wait.

The simple pattern with a single leader does not work because many generic built-ins (e.g., delay, wait) are blocking. The workflow manager in the current ALA engine takes advantage of the fact that workflows are known before the engine starts and that a thread can wait simultaneously for multiple objects to reduce the number of threads needed to help the leader wait for them. The notification API of Windows CE allows the workflow manager to handle timers, events and other workflow triggers in the same way: For time management, it uses a helper thread to assist the leader by setting and waiting for all timer events in all running workflows. Depending on the device, the manager may use the same helper thread or a separate helper thread to monitor events, rules, result tokens, etc. that are standard XPDL triggers for which workflows may wait.

V. SISARL-XPDL

The WfMC standard XPDL 2.0 [16] has been widely used for not only business applications but also many industrial applications with embedded components. We chose to include in SISARL-XPDL a minimal subset of XPDL 2.0. By including only elements essential for SISARL, we keep the SISARL-XPDL parser and resultant workflow scripts small.

A. Standard XPDL Elements

Table 2 lists examples of standard XPDL elements included in SISARL-XPDL. The table divides elements for workflow definitions and attributes into three parts: workflow data, structure, and control. How some of the elements are to be used can be deduced from their names. We have talked about some of the elements in passing in earlier sections.

| Workflow Data | TypeDeclaration, DataType, BasicType, SchemaType, DeclaredType, RecordType, DataField, InitialValue |
| Workflow Structure | Package, Pool, Lane, WorkflowProcess, BlockActivity, Activity, SubFlow, Task, Application, Implementation |
| Workflow Control | Transitions, TransitionRef, TransitionRestriction, Route, Merge, Split, StartEvent, EndEvent, Condition, IntermediateEventTrigger, Deadline, Priority |
| Resources | ParticipantType, Participant, Application, FormalParameter, ActualParameter |
| Extensions | Period, ExtendedAttributes, BCA |

We now use the example in Fig. 6 to help us explain a few more. The figure depicts a simplified diagram of a system consisting of a smart medication dispenser and its user. The outermost rectangle represents a package. The package here contains two workflow processes. The top rectangle encircles the workflows of the dispenser process: They are timer_workflow, notify_workflow, and schedule_workflow. The bottom rectangle encircles the manual workflow process of user activities. Associated with
The term SubFlow in Table 2 refers to a workflow that is called synchronously by another workflow. In the example in Fig. 6, schedule_workflow is a subflow of notify_workflow. A workflow may also be called asynchronously; the interaction between notify workflow and timer workflow is an example.

The example has only a two-way XOR split (i.e., an if-else built-in activity) depicted as a circle labeled R. XPDLEvent activity is a general primitive for alternating the courses and timing of a workflow process. An event trigger can be time, timer, rule, results, result errors, and so on. The manual process illustrates the use of StartEvent. In this example, it is an alarm event. When sounded, it triggers the start of the manual workflow: The user reports to the dispenser, which in turn sets the immediate event (trg_1) waited for by notify_workflow.

Finally, XPDL schema provides a standard way for introducing user specific extensions. We have not yet fully exploited this aspect to add workflow attributes, such as rate and latency, which can be used to help the engine better service workflows with rate and deadline constraints.

B. Built-Ins for Behavior Coordination

EMWF supports large grain workflow-based design of behavior robots: In a robot with this architecture, behaviors are implemented as activities. Once starts, each behavior runs to completion without requiring engine attention.

Some transitions between activities are for coordination of behaviors. As Table 1 indicates, built-in behavior-coordination (BC) activities provided by SISARL-XPDL are arbiter for fixed-priority arbitration and superposition and voter for command fusion. We choose to support these simple mechanisms mostly because they are commonly used. One may argue that the advantages of more general mechanisms (e.g., variable priority, multiple objective coordination and fuzzy fusion) cannot offset their disadvantages in higher complexity and variability [44, 45].

Another reason is that we can implement these special-purpose built-ins from standard XPDL elements together with simple library functions.

As an example, Fig. 7 shows the workflow that implements a 3-way fixed-priority arbiter. The input I_1 has the highest priority, and I_3 the lowest. In other words, the output O is the command generated by activity (behavior) I_1 if data are present at the output of the activity when it completes. Otherwise, it is from I_2 if data are present on I_2. The output is the command generated by I_3 only when both activities I_1 and I_2 generated no data.

The 3-way JoinAND_SplitXOR and JoinXOR routers in the figure are standard XPDL elements. The workflow also contains activities implemented by Enable and Move functions. For sake of legibility, we omit all edge labels that specify the conditions under which the corresponding transitions are enabled. An Enable activity (say Enable_i) sets the condition of its outgoing transition to TRUE and thus enables the transition, when it finds data generated by its predecessor; otherwise, it disables its outgoing transition. Fixed priority arbitration is accomplished by the router JoinAND_SplitXOR. Starting from the instant when one of its incoming transitions is enabled, the router scans in fixed order all of its incoming transitions for an interval of length specified by deadline (e.g., 10 ms as shown in Fig. 7). Upon finding the first enabled transition (say from Enable_k), the condition (I_k == TRUE) of the corresponding outgoing transition of the router becomes TRUE. This causes the successor Move_k activity to be executed, moving the data from I_k to the output and enabling the transition from Move_k to the JoinXOR router. Being an exclusive-OR merge activity, the outgoing transition of the router is enabled and its successor becomes ready for execution. For example, suppose that the JoinAND_SplitXOR router finds the transition from Enable_2 enabled, but the transition from Enable_1 remains disabled during its 10 ms scan. Because the transition from Enable_2 is scanned before the transition from Enable_3, the condition of the outgoing transition to Move_2 becomes TRUE and the transition enabled regardless whether the transition from Enable_3 is enabled. The command sent to the robot is from activity I_2.

C. Workflow Scripts

For relative simple applications, it suffices for us to parse SISARL-XPDL definitions of workflows directly to binary scripts. We anticipate, however, the need for intermediate scripts in C. They can offer developers of large complex workflow applications added convenience during debugging and for experimentation and evaluation purposes.
Fig. 8 shows the C data structures used by EMWF for intermediate workflow scripts. GENERAL and BUILTIN in the left column give the structures of the general and built-in script fields in the ACTIVITY structure shown in Fig. 4.

We note that the type of a built-in activity, together with flag and numbers of incoming and outgoing transitions specify the activity exactly. Take JoinAND_SplitXOR activity in Fig. 7 as an example. It is of BuiltInRoute type with RouteFlag IS_TXIN_AND | IS_TXOUT_XOR. Similarly, by using a combination of flag values, we can specify an event-type built-in activity as a wait for events, timers, and/or other types of triggers in different combinations of ways.

<p>| enum BUILTINTYPE { |
| BuiltInRoute, BuiltInEvent, |
| BuiltInCallWorkflow, BuiltInBCA; |</p>
<table>
<thead>
<tr>
<th>}</th>
</tr>
</thead>
<tbody>
<tr>
<td>#define IS_TXIN_AND 0x01</td>
</tr>
<tr>
<td>#define IS_TXIN_XOR 0x02</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>#define IS_WAIT_EVENT_ALL 0x02</td>
</tr>
<tr>
<td>#define IS_WAIT_EVENT 0x10</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>#define IS_TXIN_AND 0x01</td>
</tr>
<tr>
<td>#define IS_TXOUT_XOR 0x02</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>typedef struct general_activity_script_extension_t {</td>
</tr>
<tr>
<td>ProcedureNameSizeOrID;</td>
</tr>
<tr>
<td>NumberInputParameters;</td>
</tr>
<tr>
<td>NumberOutputParameters;</td>
</tr>
<tr>
<td>... } GENERAL;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>typedef struct builtin_activity_script_extension_t {</td>
</tr>
<tr>
<td>... } GENERAL;</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Fig. 8. C structures for intermediate workflow script

VI. EVALUATION

As a part of experimentation with workflow-based design and implementation, we have built parts of representative SISARL devices from workflows, including a motion control and floor covering module for Roomba [12] and parts of smart pantry controller [7]. We are also using workflows extensively as executable specifications of device operations, user actions and device-user interactions [1] for simulation and evaluation purposes. These studies have convinced us that componentization comes naturally with workflow-based design of embedded devices, as advocates of workflows say about enterprise applications. Indeed, workflow engines provide an easy-to-use, flexible platform for dynamic integration of reusable components.

As a parallel effort, we have taken measurements of runtime performance of several benchmark workflow-based workloads running on ALA, WLA and WLA-HT engines. The engines ran on a 3.4 GHz Pentium4 and Windows CE 6.0 platform. Through this series of measurements, we want to determine whether the runtime overhead introduced by the workflow engine is tolerable for representative SISARL devices. In addition, we want to assess the relative merits of different engine structures in this respect.

Each workload is characterized by the number of workflows in the load and the test pattern and granularity of the workflows. Fig. 9 shows the test patterns used in this experiment. Boxes labeled D in the workflow graphs represent general activities. Granularity of a workflow gives a measure of the size of the dummy code called by the general activities. Specifically, the dummy code consists of repeated calls to a basic (code) unit. We measure granularity in terms of the number of times the basic unit is called. All the data shown here were taken with the basic unit being the Windows CE 6.0 API function Random().

![Test patterns](image_url)

The performance measure we recorded is the response time per basic unit (RTBU). For a given workflow, RTBU is equal to the total amount of time the engine spends to complete the workload divided by the product of the number and the granularity of workflows in the load. In essence, RTBU is the average execution time of a basic unit, including the total runtime overhead introduced by the engine.

During the experiment, we varied the number of workflow processing threads and the number of workflows in each workload. We found that the relative performance of the engines is not sensitive to these parameters, and neither is the trend in RTBU as a function of granularity. So, we include below only plots for the case where the number of workflow processing threads is equal to 8 and number of workflows in each load is equal to 100.

Fig. 10 gives an overview of relative performance of the ALA, WLA and WLA-HT engines for workloads with different test patterns. We can see that at the granularity of 10,000, the runtime overheads of ALA and WLA-HT engines are acceptably small. The overhead of WLA engine is high for some test patterns (e.g., Branch), and this was what motivated us to develop the WLA-HT engine. From the figure, one can clearly see that split and merge workloads on ALA and WLA engines have smaller RTBU than hardwire code. This is because the hardwired code has to create threads at runtime. ALA engine creates all the worker threads before it starts. It out performs not only hardwired code but also WLA and WLA-HT engines. Workflow-level assignment has poorer performance because the thread dispatcher becomes a bottleneck as it handles requests for threads to process split flow paths.
Parts (a), (b) and (c) of Fig. 11 show how RTBU changes with granularity for branch, split and iteration patterns. We leave out plots for sequential and merge patterns because they exhibit similar trends as branch and split, respectively. The most noteworthy plot is the one on iteration pattern. For this pattern, we were able to measure the RTBU for loads with very small granularity. The data validate what we expected: The disadvantage of ALA engine in terms of context switches can be serious sometimes. In this instance, the iteration pattern has 2000 transitions; each transition incurs a context switch on the ALA engine. In contrast, no context switch incur on WLA and WLA-HT engines and for hardwired code.

Finally, to be sure that memory footprint is not an issue for majority of SISARL devices, we ran Roomba workflows on the ALA engine. The engine consumes approximately 524 KB after it starts. It consumes 1114 KB after Roomba .wfs file is loaded. Footprint of this order is but a small fraction of available memory for typical service robots.

VII. SUMMARY AND FUTURE WORK

We described in earlier sections the design and implementation of the embedded workflow framework EMWF. EMWF provides light-weight engines for Linux and Windows CE platforms. It also provides a small but extensible language, called SISARL-XPDL, for defining workflow processes in embedded automation and assistive devices and service robots. The SISARL-XPDL preprocessor translates special-purpose built-in activities into standard XPDL. EMWF engines cannot execute XPDL directly. This is why EMWF parses XPDL workflow process definitions either into intermediate C scripts or directly into binary workflow scripts for execution by the engine.

EMWF engines are available under GPL license at http://of.openfoundry.org/projects/emwf. Runtime overheads introduced by all the engines are tolerable for most types of workloads. An exception is when workflows have small granularities. For that kind of workloads, ALA engine performs poorly. WLA-HT or WLA engine should be used. Current versions require that all workflow scripts a device needs to operate in different modes be in the system and initialized before the device starts to run. This clearly limits the ability of the device to adapt while running. Removing this limitation is one of our near term goal.

As middleware, a workflow engine duplicates many functions (e.g., scheduling and synchronization) of the underlying operating system. The extra runtime and memory overheads thus consumed can be further reduced (and eliminated all together in some cases) by having the engine runs on a thin hardware abstraction layer, replacing the OS. This is our goal beyond the immediate work needed to make EMWF ready for industrial use.

ACKNOWLEDGMENT

This work was partially supported by the Taiwan Academia Sinica thematic project SISARL, Sensor Information Systems for Active Retirees and Assisted Living.
REFERENCES


[18] SISARL (Sensor Information Systems for Active Retirees and Assisted Living), http://sisarl.org


