中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-07-007

# Component Model for SISARL Devices and Systems

T. Y. Chen, T. S. Chou, P. H. Tsai, A. Thamizhmani, T. W. Kuo,
C. S. Shih, and J. W. S. Liu

**April 18, 2007 || Technical Report No. TR-IIS-07-007**

http://www.iis.sinica.edu.tw/LIB/TechReport/tr2007/tr07.html

Institute of Information Science, Academia Sinica

Technical Report TR-IIS-07-007

# Component Model for SISARL Devices and Systems

T. Y. Chen, T. S. Chou, P. H. Tsai, A. Thamizhmani, T. W. Kuo, C. S. Shih, and J. W. S. Liu

## Abstract

This paper describes a component model, called SISARL model. It is the basis of component-based approach to building families of smart devices and systems that enhance life quality and well being of elderly individuals. In addition to providing the traditional view of hardware, firmware and software components, as do existing component models, SISARL model also provides developers with an operational view. The view enables the developer to specify device-user interactions as executable workflows and allows the device operations and user actions to be experimented with and their correctness ascertained throughout the design, development and assessment process. After describing the elements and usages of the model, the paper presents a simulation environment for this purpose.



*Submitted to ACM EMSOFT 2007*

# Component Model for SISARL Devices and Systems

T. Y. Chen, P. H. Tsai, T. S. Chou
National Tsing-Hua University
Hsinchu, Taiwan
+886-2-2788-3799
{yen, peipei}@iis.sinica.edu.tw

J. W. S. Liu, A. Thamizhmani
Academia Sinica
Box 128, Academia Road, Sec. 2
Nankang, Taipei, Taiwan
+886-919-36-4433
{janeliu}@iis.sinica.edu.tw

C. S. Shih, T. W. Kuo
National Taiwan University
Taipei, Taiwan
+886-2-2362-5336
{cshih, ktw}@csie.ntu.edu.tw

## ABSTRACT
This paper describes a component model, called SISARL model. It is the basis of component-based approach to building families of smart devices and systems that enhance life quality and well being of elderly individuals. In addition to providing the traditional view of hardware, firmware and software components, as do existing component models, SISARL model also provides developers with an operational view. The view enables the developer to specify device-user interactions as executable workflows and allows the device operations and user actions to be experimented with and their correctness ascertained throughout the design, development and assessment process. After describing the elements and usages of the model, the paper presents a simulation environment for this purpose.

## Categories and Subject Descriptors
J.7 [**Computers in Other Systems**] D.2.2 [**Design Tools and Techniques**]:

## General Terms
Design, Experimentation, Human Factors

## Keywords
Component-based design, Modules and interfaces, Activities and workflows, Operational specifications, Simulation environment

## 1. INTRODUCTION
Since its advent decades ago, the merits of component-based approach to constructing systems from configurable building blocks with well-defined interfaces have been demonstrated undisputedly. Over time, the approach has been adopted for an increasingly broader spectrum of devices, systems and services. Today, one can find component-based networked sensors [1, 2], embedded software and systems (e.g., [3-5]), robotic software (e.g., [6-8]) and multimedia and general DSP (e.g., [9, 10]), as well as large enterprise systems and applications built on component technologies such as EJB, .NET and CORBA.

This paper describes a component model that serves as the basis for design, development and quality assurance of SISARL (Sensor Information Systems for Active Retirees and Assisted Living) devices and systems. We are building a component library, an integration framework and supporting tools based on the model. The term SISARL refers to consumer electronics designed to enhance life quality of elderly individuals and help them live independently [11-16]. The advantages of component-based approach are particularly important for these devices. Being consumer electronics, they must be affordable and easy to use, customize, and maintain. All SISARL devices provide their users with essential services while the users are well. When the need arises, some also serve as point-of-care and assistive devices. They must function dependably and fail in a safe manner even when misused by their untrained users.

The SISARL model provides us with two views: resource view and operational view. In the *resource view*, a device or system is composed of hardware, firmware and software building blocks, called *resource components* or simply *components*. Existing embedded system component models typically support this view and this view solely. For SISARL devices, however, resource view is insufficient. Full automation is not always economically feasible and, often, is not desirable. Like other semiautomatic devices, many SISARL devices rely on the user(s) to perform some essential functions. The *operational view* of a device specifies the activities of the user and device and their collaborations: It defines the required functions provided and used by the user and thus defines the user as a resource component. At the same time, it constrains the user behavior and defines conditions for correct device operations.

SISARL resource components and their interfaces are described in a nesC-like language [1, 2]. Like nesC model, SISARL resource view also supports event-driven execution, bidirectional interfaces and flexible hardware-software boundaries. This is where the resemblance between the models ends. nesC model allows only static memory allocation, nesC events and tasks run to completion and tasks do not preempt each other. These and similar restrictions of nesC and other models (e.g., [17]) make the devices based on them easier to verify and test but can unduly constrain architecture and implementation of SISARL product families. SISARL devices are typically not entirely reactive; many are complex and most are not severely size and power limited. Rather than making such restrictions an intrinsic part of the model, SISARL model offers them as options imposed when some requirements are best met with them. We will return to discuss specific differences between nesC and SISARL models when elements of resource view are described

Following the lead of programming languages such as C# and Real-Time Java, SISARL resource view uses attributes to provide information on components. The values of some attributes specify restrictions. Attributes which instruct the preprocessor, compiler and run-time environment of static linkage scope, memory allocation and scheduling restrictions are example. Some attributes (e.g., platform and version) guide component selection. Other attributes declare capabilities and limitations. For example,

we can use values of quality of service attribute to specify the input quality a component requires and output quality the component can produce.

In the operational view, the work by a device consists of one or more workflows [18]. Each *workflow* is defined by a workflow graph. Roughly, each node in a *workflow graph* represents an *activity,* which is a job, a task, or some other granule for work carried out by the system or user(s). *Edges* specify the conditions under which activities are carried out. In this sense, workflow graphs resemble task graphs commonly used to characterize real-time and embedded workloads. Workflow graphs are widely used in enterprise systems to define automated business processes. While sequencing and synchronization between jobs in a task graph are hardwired in the process code, they are done for workflow graphs by a workflow engine. In essence, the *workflow engine* integrates dynamically the components that implement activities. Indeed, we use a workflow engine as a component of SISARL integration middleware in some devices [19]. This is a reason for using workflows to describe operational view.

Extensive works on man-machine interactions and automation surprises can be found in literature and on Internet. These works typically assume that users (e.g., pilots, drivers) are trained. We cannot readily apply their assumptions and user models. SISARL users are untrained and may even decline in general skills and alertness over the years while the devices are in use.

The remainder of the paper is organized as follows. Section 2 provides background on SISARL devices in general and briefly describes two devices that are used as illustrative examples in later sections. Section 3 gives an overview of SISARL resource components and operational view. Section 4 presents and illustrates the elements of the resource view. Section 5 discusses operational view. Section 6 describes our simulation environment. Section 7 summarizes the paper and future work.

## 2. SAMPLE SISARL DEVICES

SISARL families of devices and systems range from small gadgets to sizable home appliances, from consumer electronic products to assistive and home-care equipment, and from passive monitors to robotic helpers. Walker's buddy [13] and object locator [16] are examples of gadgets. When worn by a walker or jogger, the former can detect and warn the wearer of uneven pavement ahead and thus help preventing falls. A RFID-based object locator uses smart phones and PC for user interface and provides its users with the capability of querying for locations of misplaced household and personal objects (e.g., key) by names. Examples of point-of-care devices include vital sign monitors. Examples of assistive devices include semiautomatic robotic helpers. Medication dispensers for medical and assistive care institutions [15] also contain robotic components.

Figure 1 shows two representative SISARL devices: a storage pantry [12] on the left and a personal medication dispenser [14] on the right. Both are for use by untrained users at home over years. We use these simple devices for illustrative purposes and to put issues in user-device interactions in context.

### 2.1 Smart Pantry

A smart pantry is for storage of non-perishable household supplies. The pantry is smart because when the last unit of any supply is removed from it, the pantry will contact a specified supplier, place an order and arrange payment on user's behalf to have replenishment delivered within a specified time.

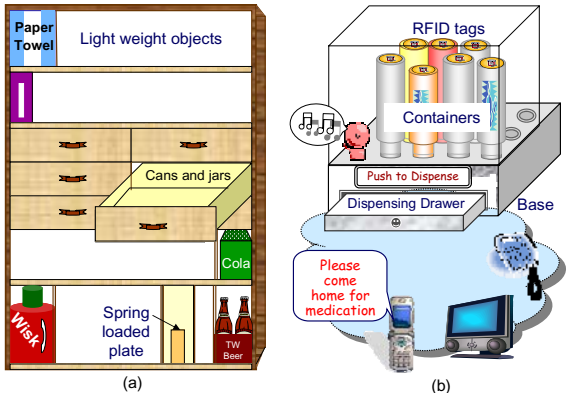


**Figure 1. Two sample SISARL devices**

A smart pantry must have some means to identify and monitor its content. A pantry using RFID for this purpose is fully automatic and easy to use, but not economically feasible: Even a cent per tag is too costly to replace bar codes on bottles of shampoo, rolls of papers, etc. We have built and experimented with a pantry that uses a digital camera for content capture and pictures for object identification. It is also fully automatic and easy to use for the pantry owner, but its usability is poor for suppliers. Each picture in purchase orders from the pantry must be processed to identify the brand and size of the object in it. Because picture quality is not ideal and object search space is large, most object identification methods cannot attend the required error rate.

BAC (bar code) pantry [12], which identifies supplies by their bar codes, has the best cost versus usability tradeoff today. As shown in Figure 1(a), each shelf is logically divided into compartments: Each compartment is marked by a pair of switch and spring loaded plate on the shelf. Each compartment is used to hold only one kind of supply. The pantry controller determines whether the compartment is empty by sensing the state of the corresponding switch: When a compartment is nonempty, its switch is pushed open by object(s) in front of the plate. When it is empty, the plate snaps forward and closes the switch.

BAC pantry is ideal for suppliers, but cannot work without user's help. The work required of the user (i.e., the owner) is actually simple: when placing an object into an empty compartment, scan its bar code. The scan-place activity of the user triggers the pantry to generate and maintain a compartment-id-bar-code association. When a compartment becomes empty, the pantry inserts the associated bar code in the purchase order. It then deletes the association and thus frees the compartment for new supplies. The process of acquiring bar codes is error prone, however. A busy user may dump supplies in the pantry without scanning their bar codes. Multiple users may put away supplies and remove supplies at the same time. We cannot restrict user-pantry interaction patterns but must make sure that the pantry works satisfactorily regardless.

### 2.2 Medication Dispenser

We are building a fully automatic personal medication dispenser for use by naïve users [14, 15]. It is illustrated in Figure 1(b).

Each dispenser has a *medication scheduler* that computes administration schedule (i.e., time and size of every dose) of every medication managed by the dispenser, a *dispensing unit* that releases doses of medications to the user according to the schedule, and a *compliance monitor* that detects non-compliance to medication directions and takes appropriate actions. All are under the control of the *dispenser controller*. It also has a small non-volatile memory for storing a machine readable *medication schedule specification* (*MSS*). The scheduler computes schedules following the directions given by the specification. A minimal dispenser uses a local alarm to remind the user at times when one or more doses is due and a dial up connection for sending appropriate notifications when non-compliance arises. A high-end dispenser may use a variety of devices and network links for these purposes and provide a local database for keeping records on medication history and user preference and behavior.

In order for medication dispensers to be effective in prevention of medication errors, all medications taken by each user are managed by a single dispenser. Moreover, some professional has verified that the directions of user's medications are right for the user. A fair assumption is that someone is a pharmacist. Whenever the user comes to get medication supplies, the pharmacist verifies the user's directions with the help of a prescription authoring tool [15], generates a new MSS for the user's dispenser, and gives the user the MSS in a flash memory, along with new supplies, each medication in a separate container tagged with the RF id of the medication. To put new supplies of medications under the care of the dispenser, the user loads the new MSS into the dispenser and plugs the new containers into sockets on top of the dispenser base, one container at a time and one per socket. Each plug-in action causes the dispenser controller to read RFID tags and socket statuses and thus discover the location of the new container and the id of the medication in it. The controller creates an association between the medication and its socket. It needs this information to operate the dispensing mechanism.

The dispenser controller uses Schedule( ) and GetNextDose( ) functions provided by the medication scheduler to determine when to give the user medication(s). When called, the former produces a *schedule*, which is a list of {time, doseList} structures. time in each entry gives the absolute time to dispense some medications and doseList specifies the id and dose size of each medication to be dispensed at the time. GetNextDose( ) returns the entry with the earliest time in the schedule and an absolute deadline for the user to retrieve the dose(s) due at the time.

The dispenser sends a reminder to the user a short time before each scheduled time for the user to take medication(s). The length of the time is an estimate of the time the user typically takes to respond to a reminder. If the user responds to the reminder and comes to retrieve the medications in time, the controller commands the dispensing mechanism to put the doses of the specified sizes due at the time in the dispensing drawer, open the drawer and present the doses to the user. If the user responds late by a specified threshold, the dispenser controller calls the scheduler to adjust the size and time for the current dose(s) in case that the MSS indicates such changes are required. If the user fails to respond by the associated deadline, the controller informs the compliance monitor of the missed dose(s) so the monitor can take appropriate action.

# 3. COMPONENT-BASED DESIGN

As the examples described above illustrate, SISARL devices differ vastly in purpose and complexity. Nevertheless, they share sufficient similarities to be built for the most part from configurable components based on a common component model.

## 3.1 Resource Components

Figure 2 shows a block diagram of typical automatic or semiautomatic devices. We use it as an architectural reference for exploitation of similarities among SISARL devices, as well as identification and definitions of components [11].

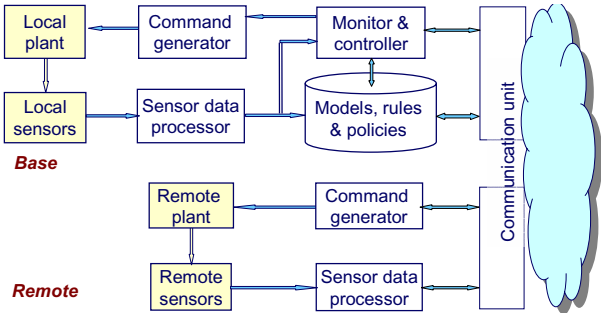


**Figure 2. Reference architecture**

Every device contains a communication unit. It connects parts within the device and provides the device with access to the world outside. The bandwidth and latency demands of typical SISARL devices are modest and can be met by existing networks and standards. Some devices have only a small non-volatile memory while others may have a sizable database. They are for storage of data, code, rules and policies, plant models and user preferences. An audio interface is used in most SISARL devices. It records voice segments of the user and plays back user voice interleaved with pre-recorded device voice. The interface makes user-device interaction friendlier. For example, it enables a smart pantry to confirm with the user verbally the identities of supplies when they are moved in and out of the pantry. Thus it helps to make the pantry more tolerant to misuse. There are acceptable off-the-shelf choices for these components; we do not build them from scratch.

Every SISARL device has a base (unit). The base of a device typically contains sensors and sensor data processor(s) of some kind(s). These parts may be all some devices have.

The base of a device may also have a monitor and controller and one or more command generators responsible for regulating the operations of the local plant. The controller typically has multiple levels. Take smart medication dispenser for example. The local plant consists of mechanism(s) for releasing medications from their containers and depositing them in the dispensing drawer. The mechanisms have a two-level controller: The lower-level controller is open loop. The scheduler computes when and how the mechanism(s) work based on the constraints defined by MSS. The higher-level loop is closed. It monitors the timing of user response and re-computes the medication schedule and thus changes the operation of the release mechanism(s) accordingly.

Some devices have no remote unit, while others have one or more. For example, the personal dispenser described earlier has no remote unit. A dispenser for professionals [15] can be set up to

monitor each patient's reaction to medications and adjust his/her directions according to a general MSS. In that case, the patient is the remote plant and the vital sign monitor(s) and associated signal processor(s) are parts of a remote unit.

## 3.2 Operational View

Looking at only hardware and software components in the base unit of a BAC pantry, one would conclude that it does not have the loop structure in Figure 2. However, when we take into account of human (i.e., user and supplier) activities, it in fact contains a feedback control loop. The loop operates as specified by the workflow graph in Figure 3.

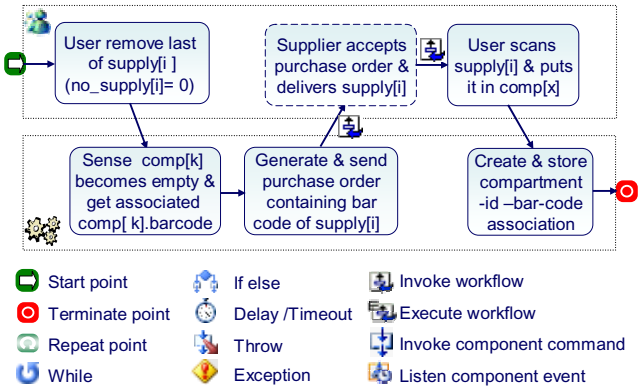


**Figure 3. Workflow to replenish a supply**

We adopt mostly the terms and notations used in Windows Workflow Foundation [18]. The nodes in workflow graphs are depicted by rectangular boxes. Text in a box describes the activity the node represents. Each directed edge represents a *transition*, i.e., a change of control flow between activities. The other notations are defined at the bottom of the figure. Specification of resources that implement the activities or are used by activities, as well as rules and policies governing resource allocation, is also a part of the workflow definition. Space limitation prevents us from including them here. An activity may be *composite*, i.e., a workflow of multiple dependent activities. Transitions to composite activities are labeled by the invoke-workflow symbol here to call attention to them. We usually separate human activities from device activities: Human activities are in the dotted rectangle labeled by a small figure on top (or to the left) of the dotted rectangle encircling device activities.

Returning to the control loop in a BAC pantry, we note that the set of supplies in the pantry can be thought of as the local plant. The goal of the loop is to keep the pantry stocked with them (i.e., the state variable no_supply[i ] > 0 for every supply supply[i ] in it). The workflow starts when the user removes the last of a supply, say supply[i] in compartment comp[k]. The activity causes the pantry to detect that comp[k] has become empty. In response, it looks up the bar code comp[k].barcode of the supply from the compartment-id-bar-code association of the compartment. It then generates and sends to a default supplier a purchase order containing the bar code, the number of units ordered and the desired delivery date. The arrival of the purchase order causes the supplier to process the order, collect payment and then deliver the supply. The delivery in turn triggers the user to place the replenishment in some empty compartment comp[j] and the pantry to create and maintain a new compartment-id-bar-code association. Thus the control loop restores the supply to the desired no_supply[i] > 0 state.

We note that the workflow graph in Figure 3 consists of three workflows. The composite activity by the supplier, depicted by the dashed box in Figure 3, is not a part of the pantry. We will return in Section 5 to elaborate on the third workflow, consisting of the last two activities. The first workflow starts by the supply removal event ends on the device after third activity completes. The first two activities in this workflow are done by the user and embedded, event-driven components of the pantry. The activity that generates and sends a purchase order is carried out in the background by an off-the-shelf e-business component. It and other components like them are the reason we chose to make restrictions of nesC concurrency model optional for SISARL.

Figure 4 shows three ways the operational view is used. The example above illustrates how operational view complements resource view to complete the device specification. By writing operational specification of a new device in terms of workflows, we can simulate, emulate and experiment with its operations and its interaction with the user throughout the design, development and quality assurance process, as depicted by Figure 4(a).

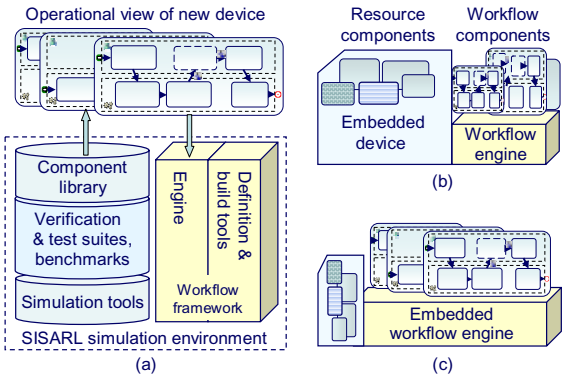


**Figure 4. Uses of operational view**

Figure 4(b) shows a workflow engine used as a middleware for integration of embedded device(s) into a larger system. An example is a workflow-based integration framework [19] that links medication dispensers with order entry systems and prescription authoring tools. In addition to resource components that implement the devices, the system also contains workflow components that execute on an engine external to the devices.

Figure 4(c) illustrates an architecture designed to make devices easily configurable. Only a small part of a device based on this architecture is built with resource components with hardwired control flow. Most of device is composed of workflow components that execute on a small embedded engine. We can change the functionality and operations of the device by simply changing the graphs of workflows that do the work.

# 4. ELEMENTS OF RESOURCE VIEW

We chose to make SISARL resource component definition language a variant of nesC [1, 2] because of strengths of nesC that are important for embedded devices and systems: Hardware/software transparency will ease our future efforts in migrating towards SoC (System-on-Chip) implementations. We also want some embedded components to be statically linked, as

nesC components do. This section first summarizes nesC features that are adopted for SISARL and illustrates how we use them, and then describes SISARL features that differ from nesC.

## 4.1 Use of nesC Features

We use the terms interface, component, module, and configuration as they are defined in nesC. Briefly, an *interface type* specifies a set of named functions. An *interface* is an instance of an interface type. A *component* is a hardware, firmware or software building block that is encapsulated by one or more interfaces. We call it a *resource component* when we want to distinguish it from an activity and highlight the fact it may be used to implement an activity.

A distinguishing feature of nesC adopted by SISARL is that each interface function is specified either as a command or an event. A component implements all *command functions*, (or *commands* for short) in its interfaces and provides them to other components (i.e., its users) as requests for its services. Commands may be split-phase operations. A split-phase command is asynchronous, i.e., it returns immediately and posts an event upon its completion. The callback function for handling the completion event is specified as *event* in the interface together with the command. A user component of the interface is expected to implement the event function. The specification of each component lists the interfaces it *provides* and interfaces it *uses* and thus specifies how the component can be statically linked with other components to compose a larger component. A *module* is a component that does not contain other components as parts. A *configuration* is a component that is composed of modules and smaller configurations and hides the details about them.

As an example, we describe now the MedicationSchedulerC configuration that implements the medication scheduler in medication dispensers. The scheduler gets the information it needs to compute medication schedules from two modules, MedScheduleSpecM and UserPreferenceM. Figure 5 lists their specifications. (To save space in figures, we indicate data types and parameter lists as "…" at places in where specifics about them are either provided in text or are unimportant for our discussion here.) The former encapsulates the MSS, which specifies hard and firm constraints to be satisfied by all medication schedules. UserPreferenceM contains data on user preferred times and frequencies, etc. the dispenser collected from the user. The scheduler treats them as soft constraints. We note that each of these modules implements a command with which user components can request the information. The commands are non-blocking. This is particularly important for the get MSS request. The specification is written in XML. The module must process the specification to extract medication directions from it and put the directions in a standard structure ready for use by the scheduler. In some dispensers, the module processes the XML file locally; in other dispensers, the module in turn requests a remote service to do the work. In any case, the work takes time. The requester is notified of the completion and proceeds to execute the event function specified in the interface.

MedicationSchedulerC configuration is composed of three modules: They are DosageSelectorM, ScheduleGeneratorM and ScheduleEnhancerM. Figure 6 shows their composition. In the diagram, interfaces are represented as smaller boxes outside of the boxes representing components. Filled arrows between interfaces represent commands; unfilled arrows represent events. The modules in Figure 5 and their interfaces should appear at the right of the diagram if there were space for them. The interface ReadMedicationSchedule is not connected because it is not used.

```
interface ReadMedScheduleSpec {
    command error_t Get (user_id_t current_user);
    event void GetDone (error_t err, mss_struct_t mss);
}
interface WriteMedScheduleSpec {
    command error_t Set (stream* mss);
    event void SetDone ( );
}
interface ReadUserPreference {
    command error_t Get (user_id_t current_user);
    event void GetDone (error_t err, … user_preference);
}
interface WriteUserPreference {
     command error_t Set (stream* user_preference);
     event void SetDone( );
}
module MedScheduleSpecM {
     provides interface ReadMedScheduleSpec;
     provides interface WriteMedScheduleSpec;
}
module UserPreferenceM {
     provides interface ReadUserPreference;
     provides interface WriteUserPreference;
}
```

**Figure 5. Modules used by MedicationSchedulerC**

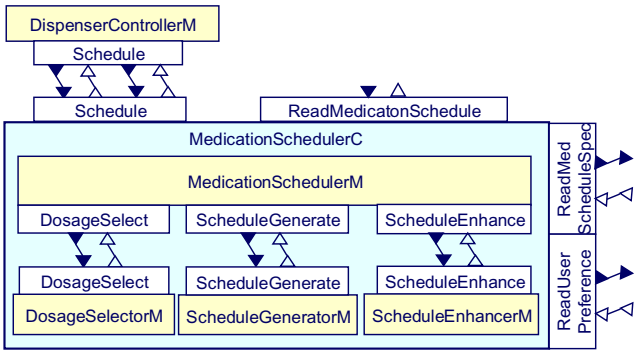


**Figure 6. Composition of MedicationSchedulerC**

Figure 7 lists the specification of MedicationSchedulerC together with interfaces and modules in it. DosageSelectorM and ScheduleGeneratorM take as input the medication schedule specification mss produced by module MedScheduleSpecM. DosageSelectorM selects sizes and nominal separations of individual doses based on the direction of each medication. ScheduleGeneratorM generates a medication schedule, starting from dosage selections produced by DosageSelectorM and factoring into account additional constraints due to drug interactions. ScheduleEnhancerM adjusts the medication schedule, taking into account of user preference, to make the schedule friendlier to the dispenser user. In lines specifying the implementation of component, the interface to the left of each assignment symbol "=" is the interface to the right of symbol. The symbol "->" links the provider and user of the interface.

```
interface Schedule {
    command error_t Schedule (user_id_t current_user);
    event void ScheduleDone (error_t err);
    command error_t GetNextDose (user_id_t current_user);
    event void GetNextDoseDone ( … );
}
interface ReadMedicationSchedule {
     command error_t Get (user_id current_user);
     event void GetDone (… err, … med_schedule);
}
interface DosageSelect {
    command error_t SelectDosage (mss_struct_t mss);
    event void SelectDosageDone ( … );
}
interface ScheduleGenerate (
    command error_t GenerateSchedule ( … );
    event void GenerateScheduleDone ( … );
}
interface ScheduleEnhance (
     command error_t EnhanceSchedule ( … );
     event void EnhanceScheduleDone ( … );
}
module DispenserControllerM {
     uses interface Schedule;
     … ;
}
module DosageSelectorM {
      provides interface DosageSelect;
}
module ScheduleGeneratorM {
      provides interface ScheduleGenerate;
}
module ScheduleEnhancerM {
      provides interface ScheduleEnhance;
}
module MedicationSchedulerM {
      provides interface Schedule;
      provides interface ReadMedicationSchedule;
      uses interface ReadMedScheduleSpec;
      uses interface ReadUserPreference;
      uses interface DosageSelect;
      uses interface ScheduleGenerate;
      uses interface ScheduleEnhance;
}
configuration MedicationSchedulerC {
      provides interface Schedule;
      provides interface ReadMedicationSchedule;
      uses interface ReadMedScheduleSpec;
      uses interface ReadUserPreference;
}
implementation {
      components MedicationSchedulerM, DosageSelectorM,
                 ScheduleGeneratorM, SchedulerEnhancerM;
      Schedule = MedicationSchedulerM.Schedule;
      ReadMedicationSchedule =
            MedicationSchedulerM.ReadMedicationSchedule;
      MedicationSchedulerM.DosageSelect ->
            DosageSelectorM.DosageSelect;
      MedicationSchedulerM.ScheduleGenerate ->
            ScheduleGeneratorM.ScheduleGenerate;
      MedicationSchedulerM.ScheduleEnhance ->
            ScheduleEnhancerM.ScheduleEnhance;
}
```

**Figure 7. Components and interfaces of scheduler**

## 4.2 SISARL Variations and Extensions

SISARL resource component description language has nesC as a foundation, but SISARL model deviates from nesC model in many important aspects, including concurrency model, scope of static linkage and non-functional elements.

### *4.1.1 Concurrency Model and Static Binding Scope*

Tasks in nesC resemble tasklets in Linux and deferred procedure calls (DPC) in Windows; they do not preempt events and do not preempt each other. In contrast, SISARL execution model is traditional by default for reasons stated in Section 1. SISARL tasks are executed in context of threads. They are preemptively scheduled according to their priorities. Event functions handle interrupts and completion operations. They run ahead of tasks and may preempt each other. On Linux (or Windows), an event may be a split-phase operation: The (first-phase) handler may post a tasklet to complete the work. Tasklets may be preempted by event handlers but never by each other and never by tasks.

Another important difference is in the scope of static linkage: An application written in nesC has only one *top-level configuration* (i.e., a configuration that is not a part of a larger configuration). All modules used to implement an application are statically linked and compiled into a single image. This is not so by default for SISARL. A SISARL component may be statically linked within an upper-level configuration as in nesC. In this case, the component name is listed in the implementation specification of the upper-level configuration, as illustrated by Figure 7. By default, a component not thus named in the specification of any upper-level configuration is to be compiled by itself into an executable to provide at runtime a service or services via command(s) it provides. A component may be used in both ways. The developer makes this choice via instructions to SISARL preprocessor, telling the preprocessor to include the component in the list of separately compiled codes regardless whether the component is also a part of a larger configuration.

To illustrate the need for these choices, we return to Figure 6: The medication scheduler configuration is shown linked to DispenserControllerM. Indeed, a personal medication dispenser is implemented by one top-level configuration that connects all components used to implement the dispenser.

We also use MedicationSchedulerC as a component of a server that provides scheduling service to dispensers used within a hospital [15]. Such a dispenser helps a care provider manage medications of all patients under his/her care. Upon request, the scheduler server merges the medication schedules of all patients served by the dispenser into an overall administration schedule for the provider. A scheduler server can be implemented using a configuration composed of MedicationSchedulerC configuration and DispatchSchedulerM module. The latter first uses the Schedule and ReadMedicationSchedule interfaces provided by MedicationSchedulerC to generate and retrieve medication schedules of the patients. It then takes into account of patient conditions and locations while merging the schedules to optimize some objective functions (e.g., total travel time and average or minimum slack time). This implementation is straightforward but does not provide the configurability required for diverse dispensers used by a large institution.

Rather than hardwiring the components of the scheduler server together, an alternative implementation is to compile the modules specified in Figure 7 and DispatchSchedulerM separately into resource components for activities in a workflow that executes on a workflow engine. Then the process of computing administration schedules can be easily configured to suit different institutions and different departments in an institution by modifying the

workflow graph definition of the process. Our component model and tools support this choice as well.

### *4.1.2 Non-Functional Aspects*

SISARL component specifications use attributes to provide information about components. There are three major types: preprocessor and run-time directives, constraints and restrictions, and quality of service (QoS) capabilities and requirements. Possible targets of attributes include interface types, interfaces, modules and configurations. SISARL component language preprocessor provides basic built-in attributes and supports custom attributes. Following the syntax of C# [20], we enclose each attribute in a square-bracket pair:

[target: attribute type (attribute values)]

It gives the type or name of the target to which the attribute applies and the type and value of the attribute. We place an attribute applied to an interface type, a module or a configuration immediately before the specification of the target. An attribute applied to an interface of a component bears the name of the interface. It is placed immediately after where the interface is listed in the specification of the component. An attribute can be applied to an individual function in an interface type, and it is placed immediately after the definition of the function.

We use configuration BinarySensorArrayC in Figure 8 to illustrate some of the attributes. The component is used in both smart pantry and medication dispenser. It is composed of two modules. hwBinarySensorArrayM is a hardware component. It may be an array of mechanical switches each of which can change state in a fraction of a second or electronic sensors that can change state in a few or tens of milliseconds. A binary stream, one bit per sensor, gives the values of the sensors. BinarySensorArrayM is a driver that responds whenever the state of any sensor has changed. The event function in its interface picks up the new value of each sensor that has changed state and inserts a work item containing the sensor id in one of two work queues according to the new state of the sensor.

SchedulingAttribute has an extensive set of values and some of them are platform dependent. Built-in values include base-priority levels. These values enable us to prioritize individual interface functions and components respective to each other. The configuration in Figure 8 uses the attribute only once: Its value requests that the event handler be executed without preemption. A fair criticism of this use of the attribute that it is not necessary since disable interrupt and preemption can be requested within the function itself. Indeed, that would be equivalent to what we have here: The target is within an interface type. So, the directive is applied to all instances of the interface. Alternatively, we can disable preemption or interrupt selectively for some instances but not others, by applying the attribute to the instances where the interface is used or provided. In addition to this flexibility, using attributes to provide scheduling directives make them more visible than burying them in code.

Some attributes constrain component usage. Metadata provided by these attributes enable the preprocessor to check components for compatibility automatically. Examples are Versions and Platform. The former provides information on compatible versions. The latter restricts the use of a component to specified platforms. A platform attribute applied to a configuration is applied to all components of the configuration. In our example, the driver and hardware modules must be selected from those for versions of Linux and ARM included in the platform specification given as value of the Platform attribute applied to the configuration.

```
interface hwBinarySensorArray {
    command int GetNumberSensors ( );
    command stream GetSensorValues ( );
    event void SensorValueChanged ( );
}
interface BinarySensorArray {
    event void InsertWork (stream sensor_ids,
                stream sensor_values, queue_t queue_name);
    [InsertWork: SchedulingAttribute (PreemptionDisabled)]
}
interface StdControl {
    command result_t init ( );
    command result_t stop( );
}

[module: QoSAttribute (ResponseTime <= 500)]
module hwBinarySensorArrayM {
    provides interface hwBinarySensorArray;
}
module BinarySensorArrayM {
    provides interface StdControl;
    provides  interface BinarySensorArray;
    uses interface hwBinarySensorArray;
    [hwBinarySensorArray: QoSAttribute (ResponseTime
                                        <= 750)]
}
[configuration: PlatformAttribute (Linux_Arm_04/07/00014)]
configuration BinarySensorArrayC {
    provides interface StdControl;
    provides interface BinarySensorArray;
}
implementation {
    components  hwBinarySensorArrayM,
                BinarySensorArrayM;
    StdControl = BinarySensorArrayM.StdControl;
    BinarySensorArray =
            BinarySensorArrayM.BinarySensorArray;
    BinarySensorArrayM.hwBinarySensorArray ->
            hwBinarySensorArrayM.hwBinarySensorArray;
}
```

**Figure 8. An example illustrating use of attributes**

Figure 8 shows two places where QoSAttribute is applied. The target of the first one is a component. Thus applied, the attribute declares a quality guaranteed by the component. The target of the second one is an interface function used by a component: It declares a quality requirement. In our example, the first one indicates that the sensor array hardware can report a state change in no more than 500 milliseconds. Since the required response time declared by the attribute applied to interface is an upper threshold of 750 milliseconds, the hardware module can be used. Similarly, values of other built-in QoS parameters such as SamplingRate, QuantizationSNR, and so on allow us to check components not only for functional compatibility but also for performance compatibility. QoSAttribute can also be used to provide directives in tradeoffs between conflicting performance metrics. A tradeoff policy specified by an attribute is fixed at build and configuration times. This offers us an alternative to specifying QoS tradeoffs by input parameters of commands.

## 5. OPERATIONAL SPECIFICATION

Specifications of resource components of a device and their interconnections offer us a static view of the device. The operational view tells us how the device works.

## 5.1 More on Workflows

Again, we capture the operational view of a device by a set of workflows. Each workflow defines a process that can execute concurrently with other processes if scheduling and resource allocation rules and policies allow it to do so. As stated earlier, workflows are represented by workflow graphs, and workflow graphs resemble task graphs except that transitions (e.g., flow path control and synchronization) between activities in workflows are carried by a runtime engine. This middleware component executes all the tests, branches, joins, etc. based on results produced by activities.

Definitions of activities and workflow graphs, together with resource components used by activities and rules and policies governing allocation of resources to workflows, specify device operations and device-user interactions. We define workflows in XPDL 2.0. XPDL (XML Process Definition Language) is the WFMC (Workflow Management Coalition) [18] standard language for interchange of process models between tools. XPDL treats human interactions as an integral part of process definition. This is of particular important for us. Version 2.0 incorporates event and message passing mechanisms with graphical elements and meta-models. Workflows in XPDL can execute directly on engines offered by some vendors. Most vendors provide tools to translate XPDL definitions into BPEL (Business Process Execution Language) for execution on their respective engines.

Workflows are divided into two types: A *sequential workflow* uses activities, conditions, rules, etc. provided by its definition and proceeds without additional intervention. In contrast, a *state machine* (or *event-driven*) *workflow* relies on external events. Once started, such a workflow often spends time in some of its states waiting for external event(s) to proceed. Most workflows in a typical SISARL device are of this type.

## 5.2 An Illustrative Example

For sake of concreteness, we now discuss the use of operational specification with the help of an illustrative example: putting away and removing supplies in a pantry. The states of compartments in it are monitored by the binary sensor array (BSA) in Figure 8, one bit per compartment.

Figures 9 and 10 show two of three state machine workflows. They specify normal, event-driven operations of a BAC pantry. Again, we encircle user activities in a dotted rectangle on the right to separate them from device activities. An activity in both dotted rectangles, such as the one in Figure 9, is a composite activity. It is done collaboratively by the user and device. Time flows from top to bottom. User activities without transition edge(s) between them occur at separate times; the one on the top occurs earlier. To save space in these figures, we omit all activities triggered by timeouts except one in each workflow. The if-else test for that one is not drawn either. Rather, we label the activity that executes only when timeout occurs by a pair of timeout and exception symbols. If executed, the activity thus labeled in each workflow terminates the workflow.

After initialization, the pantry (i.e., the pantry controller) waits for the user to put supplies in it. The user may do this by touching the Load_Pantry button. This event triggers the workflow in Figure 9. The pantry responds by turning on the bar code scanner. The user scans a unit of a supply and then puts the supply in an empty compartment. This action causes the state of the compartment to change to non-empty, the BSA to insert a work item containing the compartment id, and the pantry to store the bar code obtained from the user's scan for the compartment. The user continues to scan and put away supplies until all supplies are put away. When the pantry timeouts while waiting for more bar codes, it shuts off the scanner and goes back to wait, and the workflow terminates. – The collaborative activity in the middle of the process executes when the pantry sees a bar code it has never seen before. Making use of the audio interface component mentioned in Section 3, the pantry asks the user to record a voice description of the new supply. It uses the recorded voice description associated with each bar code later on for many purposes, such as confirming orders and reporting errors.

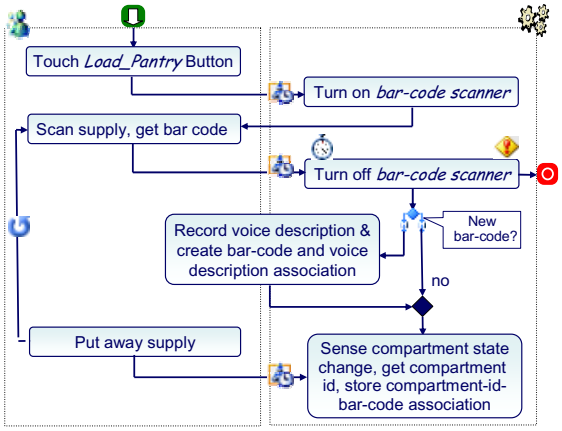


**Figure 9. load_pantry workflow**

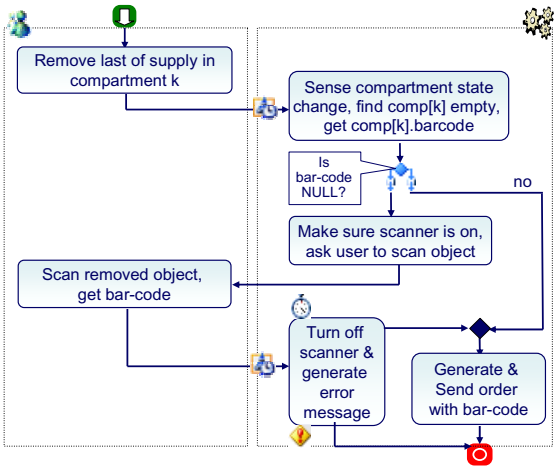


**Figure 10. remove_supply workflow**

The user may just put a supply in an empty compartment without first scanning its bar code. This action also triggers a put-away-supply workflow similar to the one shown in Figure 9. In that case, the pantry requests the user in voice to scan the bar code of the supply. If the user does as requested, the pantry again gets the desired compartment-id-bar-code association. The user may ignore the request, however, leaving the bar code associated with the compartment NULL.

The workflow in Figure 10 starts when the user removes the last unit of a supply, leaving a compartment empty. The pantry responds to a state change event and finds that compartment empty. If the associated bar code is NULL, the pantry requests the

user to scan the supply. If the user responds to the request, all is well: Armed with a bar code, the pantry can reorder the supply. If the user ignores the request, the pantry generates an error message to inform the user of its failure to reorder the just removed supply when timeout occurs and returns to wait again.

We note that each of the workflows by itself is simple, and simplicity is more typical than exceptional for most SISARL devices. It is often possible to manually verify the correctness of each workflow by itself. Furthermore, there are tools for transforming the XPDL or BPEL definition of a workflow into a Petri nets or state diagram so that some rigorous or formal verification method can be applied.

Verifying multiple concurrent workflows that content for resources is another matter, however, even for simple workflows. The problem is further complicated by the fact that the workflows are triggered by user actions and users are unpredictable. Take our example for instance. One or more users may remove supplies and empty some compartments while another user is putting away supplies. The workflows triggered by them content for BSA. Because the array is used by each workflow for a negligible amount of time, we can simply make its event handler non-preemptable, as shown in Figure 8. If some removed supplies need to be scanned, then load_pantry and remove_supply workflows also content for the bar code scanner and the audio interface. Disallowing preemption of the activities while they use these resources is not acceptable for both functional and performance reasons. In our current version, a workflow triggered by the removal of a supply without bar code has the highest priority. The pantry interrupts the load_pantry process and asks the bar code of the removed supply be scanned before the user taking it walks away, which may take only a second or two. When there are two users, this prioritization yields correct pantry operation (i.e., the pantry correctly associate the supplies with the bar codes acquired by the workflows). The simulation environment described below is motivated by our need to have clearer understanding of how the pantry behaves when an arbitrary number of concurrent workflows contents resources.

## 6. SIMULATION ENVIRONMENT

We are building incrementally the simulation environment illustrated by Figure 11. The environment contains a runtime engine together with a suite of tools for build, simulate, emulate and evaluate SISARL devices based on their operational specifications, resource component specifications and implementation. The current prototype is built on top of Windows Workflow Foundation (WF) [18] and .NET framework. Resource components can be implemented in C, C++, and C#. Activities in operational view can be written in XML and C#. Workflows can be defined graphically first, with code added as required.

Like similar tools, SISARL environment provides several extensible libraries: Resource component library is a repository of specifications, code and executables of building blocks in various stages of development and quality assurance. Activity and workflow libraries provide reusable definitions of operational view components and specifications. The repositories enable us to put together from design a new device model for experimentation and evaluation purposes easily. We want to be able to build parts of the model device from resource components as soon as the components are implemented and sufficiently tested, while using activities and workflows to simulate or emulate the parts that await implementation. Also, a variety of real and simulated resources and rules and policies governing their usages and allocations are available to support a wide range of experiments.

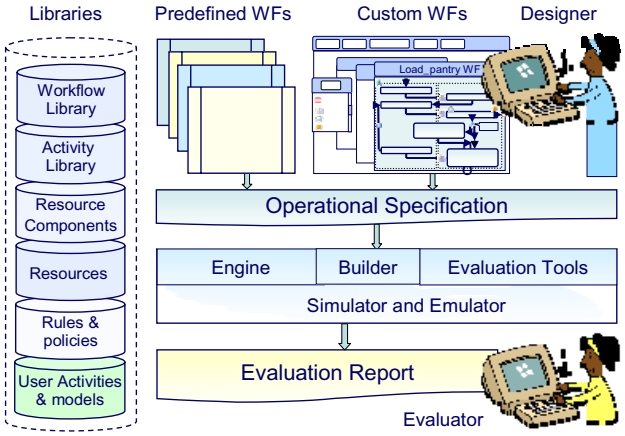


**Figure 11. Libraries and Tools**

The user activities and models are of particular importance. A user activity is used to simulate a work item or action performed by a human user. Figures 9 and 10 give a few examples. Examples medication dispensers include "plug a container in a socket", "respond to a reminder by pushing the Push-to-Dispense button", and "retrieve medication from drawer".

User activities are parts of state machine workflows. We use a pre-condition, one or more post-conditions and a set of delay paths to implement the user activities. The pre-condition of an activity in a workflow defines an initial state (or a set of states) of the workflow immediately before the execution of the activity. Each post-condition defines a final state immediately after the activity completes. By a delay path, we mean a delayed state transition from the initial state to a final state. The choice of the delay path (i.e., the final state) and the length of delay are selected from a set of probability distributions that models the user. For example, the activity "Scan supply, get bar code" activity in Figure 9 is modeled by

Pre-condition: controller_state == AWAITS_BAR_CODE
Post-conditions: (bar_code_register != 0 && Timeout == FALSE)
|| (bar_code_register == 0 && Timeout != FALSE)

The time taken by the user to scan the supply is a random variable. Different users are modeled by different probability density functions of this random variable. Given a user model, we can determine the probability of getting a bar code before timer expiration, and hence which of two final states the workflow reaches after the activity and how long the activity takes to reach the final state.

Compared with most existing user models (e.g., models of autopilot and medical devices users), ours is much simpler. The simple models suffice for our purposes. The amount of effort required to validate detailed user models can be enormous. The simplicity of our user models enables us to validate them incrementally. As more and more actual data on user behaviors become available, we will replace the hypothetical probability density functions used to model users with actual histograms.

## 7. SUMMARY AND FUTURE WORK

This paper describes a component model for building from reusable components affordable, configurable and dependable SISARL devices, i.e., consumer electronics and assistive devices for the elderly. SISARL component model offers two views: In resource view, resource components, interfaces and component connections are specified in a variant of nesC. The underlying SISARL execution model is traditional, without nesC restrictions on memory allocation and concurrency. In operational view, a device and its user(s) are defined by executable activities and workflows that specify their operations and interactions. By defining a device in terms of what it does, the operational specification of a device complements its resource view specification, which defines how it is made.

The simulation environment and integration framework we are building will enable us to experiment with and evaluate new device designs and prototypes as soon as they are specified and during their development. In particular, by executing the operational specification of a new device with user activities based on validated user models, the environment can help us to better assess the usability of the device.

We chose to build the current version of the environment on Windows Workflow Foundation because we want a prototype environment for experimentation in minimal time. In the meantime, we are building a lightweight WFMC workflow management system in C. Being lightweight and small, it will fit on many embedded devices. We want to use it for integration of activities and workflows in devices base on the architecture illustrated by Figure 4(c). We will migrate the simulation environment to this workflow management system so that we can release the environment under a open source software license.

## 8. ACKNOWLEDGMENTS

This work is supported by Taiwan Academia Sinica Thematic Project SISARL, Sensor Information Systems for Active Retirees and Assisted Living.

## 9. REFERENCES

[1] Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., and Culler, D., "The nesC language: a holistic approach to networked embedded systems," *Proceedings of PLDI,* June 2003.

[2] Levis, P., *TinyOS Programming*, http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf

[3] Zha, X. F. and Sriram, R. D., "Feature-based component model for design of embedded systems," *Proceedings of SPIE,* Volume 5605, 2004.

[4] Crnkovic, I., "Component-based approach to embedded systems," 9th International Workshop on Component-Oriented Programming, June 2004.

[5] Crnkovic, I., Axelsson, J., *et al*,. "COTS component-based embedded systems - A Dream or Reality?," 4th International Conference, ICCBSS 2005, February 2005.

[6] Oreback, A., "A component framework for autonomous mobile robots," Doctoral thesis, KTH, Numerical Analysis and Computer Science, Sweden, 2004.

[7] Wisspeintner, T., Nowak, W. and Bredenfeld, A., "VolksBot – A flexible component-based mobile robot system," RoboCup 2005.

[8] Makarenko, A., Brooks, A. and Kaupp, T., "Orca: components for robots," *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'06)*, 2006.

[9] DSPGuru, http://www.dspguru.com/sw/opendsp/index.htm , OpenDSP.

[10] Muskens, J., Chaudron,M. R. V. and Lukkien, J. J., "A component framework for consumer electronics middleware," in *Component-Based Software Development for Embedded Systems,* C. Arkinson, *et al*. ed, Springer, 2005.

[11] Liu, J. W. S., Wang, B. Y. *et al*., "Reference Architecture of Intelligent Appliances for the Elderly," *Proceedings of the 18th International Conference on System Engineering*, Las Vegas, August 2005.

[12] Hsu, C. F., Liao,Y. H., Hsiu, P. C., Lin,Y. S., Shih, C. S., Kuo, T. W. and Liu, J. W. S., "Smart pantries for homes," *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics,* October 2006.

[13] Hsu,Y., Chiang,C. E., Chien, Y. H., Tseng, H. W., Pang, A. C., Kuo T. W. and Chiang, K. H., "Walker's buddy: an ultrasonic dangerous terrain detection system," *Proceedings of IEEE International Conference on Systems, Man, and Cybernetics*, October 2006.

[14] Tsai, P. H., Yeh, H. C., Yu, C. Y., Hsiu, P. C., Shih, C. S. and Liu, J. W. S., "Compliance enforcement of temporal and dosage constraints," *The 27th IEEE Real-Time Systems Symposium*, December 2006.

[15] Liu, J. W. S., Shih, C. S., *et al*, "Point-of-care support for error-free medication process," to appear in *Proceedings of High-Confidence Medical Device Software and Systems,* June 2007.

[16] Chou, T. S. and Liu, J. W. S., "Design and implementation of RFID-based object locators," *Proceedings of IEEE International Conference on RFID Technology*, March 2007.

[17] Musuvathi, M., Park, D. Y. W., Chou, A., Engler, D. R., and Dill, D. L. "CMC: a pragmatic approach to model checking real code," *ACM SIGOPS Operating Systems Review,* 2002.

[18] WfMC: Workflow Management Coalition, http://www.wfmc.org/, and Windows Workflow Foundation.http://msdn2.microsoft.com/en-us/netframework/aa663328.aspx.

[19] Shih, C. S., *et al*. "A workflow-based integration framework for medication use," submitted to IEEE SMC, October 2007.

[20] Liberty, J., *Programming in C#*, Chapter 18, O'Reilly, 2001.

.