An ECA-Based Coordination Framework for Ubiquitous Web Service Devices

Jae-Yoon Jung[a], Jonghun Park[b], Seung-Kyun Han[b], and Kangchan Lee[c]

[a]Eindhoven University of Technology, Netherlands

[b]Seoul National University, Korea

[c]Electronics & Telecommunication Research Institute, Korea

Abstract

Emerging ubiquitous computing network is expected to consist of a variety of heterogeneous and distributed devices. While web services technology is increasingly being considered as a promising solution to support the inter-operability between such heterogeneous devices via well-defined protocol, currently there is no effective framework reported in the literature that can address the problem of coordinating the web services-enabled devices. This paper considers a ubiquitous computing environment that is comprised of active, autonomous devices interacting with each other through web services, and presents an ECA (Event-Condition-Action)-based framework for effective coordination of those devices. Specifically, we first present an XML-based language for describing ECA rules that are embedded in web services-enabled devices. An ECA rule, when triggered by an internal or external event to the device, can result in the invocation of appropriate web service in the system. Subsequently, we consider the situation in which the rules are introduced and managed by multiple, independent users, and propose effective mechanisms that can detect and resolve potential inconsistencies among the rules. The presented ECA-based coordination approach is expected to facilitate seamless inter-operation among the web services-enabled devices in the emerging ubiquitous computing environments.

Keywords

ECA rules, ubiquitous web services, device coordination, rule conflicts, conflict detection and resolution

Published in Information and Software Technology, Vol. 49, No. 11, Nov 2007, pp. 1141-1161.

1. Introduction

Ubiquitous computing is emerging to innovate on the quality of human life and the behavior of business through the digital and technology convergence [44]. Today's ubiquitous environments are increasingly becoming heterogeneous and service rich domains, where a diversity of communication devices, such as laptop computers, portable digital assistances, cellular phones, digital home appliances, automotive telematics, and various sensors, are interconnected each other beyond different platforms and physical networks. In such environments, web services technology must be an effective means for achieving inter-operability among the communication devices just as it is becoming a de facto standard of integrating heterogeneous business applications [32, 42]. Indeed, several ongoing efforts, including UPnP 2.0 [41] for home network, and OWSER [36] for mobile services, and Microsoft's invisible computing platform [34] for tiny devices, are attempting to embed web services technology into various devices for the purpose of materializing pervasive home networks. In the meantime, recently proposed web service standards, such as WS-Eventing [3] and WS-Addressing [8], are accelerating the adaptability of web services technology into distributed service devices by supporting endpoint descriptions and communication mechanisms among them, respectively.

Ubiquitous web services have largely ranged over three fundamental issues just like the web services stack. The first issue is the service description and messaging protocols. The initial outstanding web services specifications, WSDL, SOAP, and UDDI, are still the base standards in ubiquitous areas even though REST and AJAX are new approaches to lightweight and asynchronous web services communications. The second is the service discovery and awareness. The researches on semantic web and ontology-based reasoning, as well as the quality of service (QoS) protocols, are actively progressing for the purpose of pervasive and autonomous computing [11, 21]. The last issue of ubiquitous web services is the service coordination technology, which we intend to deal with in this paper. The service coordination technology was considered a significant part to provide users with advanced services in ubiquitous computing environments [45, 31]. In the service-oriented architecture (SOA) of business area, business process standard languages, such as WS-BPEL [1] and WS-CDL [28], have enabled the orchestration and choreography of web services [38]. They facilitated to coordinate application services of internal systems and business partners according to their business logics or trading contracts. In the service-oriented architecture, the centralized process enactment engines play a role of interpreting the process descriptions, transiting the states of the business cases, and maintaining long-lived transactions.

However, such process enactment techniques have several difficulties in directly adopting them to the ubiquitous environments. Since many ubiquitous service devices assume mobility, they may frequently join or leave the virtual network. The previous service coordination techniques do not embrace the dynamic connectivity for mobile communication devices and the process descriptions use the service binding to other service providers. Moreover, the rather heavy enactment engine for the long-lived transaction cannot be embedded in ubiquitous devices that may not have so sufficient computing capacities as business applications.

As a result, the ubiquitous devices require more dynamic, lightweight and decentralized coordination mechanism in mobile communication network. Motivated by this, in this paper, a decentralized rule-based service coordination framework is proposed for web services-enabled devices in ubiquitous computing environments.

Our proposed framework of ubiquitous service coordination adopted an Event-Condition-Action (ECA) rule-based approach. The ECA rules have been originally presented in the database community in order to provide traditional database systems with the capability of event-driven, instantaneous response [4, 18]. After that, since the ECA rules offer the systems a means of the concise and distinct descriptions of reactive behaviors, ECA rule management has been adopted in a variety of application fields, and currently it is recognized as one of effective coordination techniques for carrying out distributed coordination [5, 7, 15].

In this paper, we first present an XML language, named WS-ECA (Web Services-ECA), which describes ECA rules that define reactive behaviors of various web services-enabled devices, as well as service interactions among the devices. The ECA rules are embedded into the service devices and then triggered by internal events of the device or external events from others. The triggered rules, when their condition parts are satisfied, will be activated to execute their actions. The event parts of the proposed language adopt four types of primitive events (time, service, internal and external events) and four operations (disjunction, conjunction, sequence, and negation). Specially, service events are defined to catch the point of invoking specific services and finishing the services, and external events are exploited to response on event notifications sent from external devices.

The mechanism of WS-ECA rules support two ways to service interaction: one is service invocation, and the other is event notification. Service invocation is the general concept of the previous web services technique of using the request / response mechanism. On the other hand, event notification is a means of the event-driven service interaction technique, and in this paper we assume the publish / subscribe mechanism based on the WS-Eventing standard. Since event-driven interaction technique is considered more loosely-coupled and reactive than service invocation technique, event-driven architecture is emerging to compensate with service-oriented architecture in which the service activation is rather passive [13, 24, 33]. Furthermore, the publish / subscribe mechanism of WS-Eventing standards is effective in ubiquitous environments where an event of a device may affect many unknown devices. In our proposed mechanism, two approaches to service interaction are complementary so that service invocation can be used to request public services and event notification to inform other devices of the occurrences of the interesting events.

Subsequently, the paper looks into the situation that WS-ECA rules are dynamically added to and removed from the devices by multiple users. In that case, several rules which have been stored in different devices at different time may cause some inconsistency among them. To detect the inconsistency among multiple rules and resolve the potential conflicts, an effective mechanism is proposed in this paper. For this purpose, we adopt the notion of service constraints, which represent a set of actions of service invocations that are not allowed to be made simultaneously. For example, two services of turning on and off an audio device should not be performed at the same time. Such mutual exclusion requirements can be modeled as service constraints. That is, the service constraints are considered as the means of deciding the rule conflicts in the service environment which consists

of a variety of services.

Our approach to conflict detection and resolution in ubiquitous service environments is differentiated from epoch-based approaches in [10, 27]. The previous techniques to conflict resolution of ECA rules assumed the time granularity, called an epoch, and they are based on time period in which the triggering events in the set occur simultaneously. The approach is meaningful in the long-running environments, like policy description languages (PDL), however, ubiquitous environments require service devices to react more instantly than their assumptions. In our approach, the rules that are considered to check conflicts are ones that have possibility of being triggered simultaneously in the same situation (i.e. events and condition). To clear that, some properties of rule sets are defined in this paper. After that, using those properties, service conflicts are defined and adopted in the framework.

In the mechanism for ubiquitous service environments, service conflicts are categorized into static and dynamic ones depending on the time when the potential conflicts are checked and resolved. The static conflicts are identified when a new rule is added to the system, and a rule cannot be registered if it causes the conflict with any of the existing rules. On the other hand, the dynamic conflicts are detected and resolved during run-time by utilizing some prescribed rules that can settle the potential conflicts.

The proposed framework for coordinating ubiquitous service devices has been developed to address the characteristics of ubiquitous computing environment considered in the paper as follows:

- *Active service invocation*: The proposed framework not only supports the traditional passive web service invocation model, but also the active invocation model in which the web services are triggered upon the occurrences of internal and external events.
- *Lightweight implementation*: Contrary to the modeling languages such as WS-BPEL [1] and WS-CDL [28] that focus on the stateful, long-lived business processes, the proposed framework attempts to support stateless interactions to provide the capability of spontaneous reaction of service devices.
- *Decentralized coordination*: The rules governing the behavior of devices are decentralized across the network and executed independently by individual devices. In addition, coordination is carried out by means of publishing and subscribing web service event messages.

The paper is organized as follows. We first discuss related work on event-driven rule management in Section 2. The proposed framework for ECA rule-based management of web service devices is presented in Section 3, and then the structure and elements of the WS-ECA language are presented in Section 4. Subsequently, the conflict detection and resolution mechanisms for decentralized ECA rule processing are described in detail in Section 5. Finally, Section 6 concludes the paper.

2. Related Work

2.1 Ubiquitous Web Services

The advent of ubiquitous technologies is urging to innovatively upgrade the quality of human life. To realize the ideal, a lot of ubiquitous computing projects, such as Oxygen [35], Smart Dust [26], and Smart-Its [23], have shown the broad potentials of ubiquitous computing. For instance, the Oxygen project aimed at pervasive, human-centered computing, which implies that in the future the ubiquitous technology will get along with humans just like oxygen any time and any place. The project is composed of five base technologies - device, network, software, perceptual, and user technologies. Specially, user technologies includes automation and collaboration technologies to facilitate device controls in user-centered networks and spontaneous communication between device to device, device to human, and human to human. The collaboration technologies are considered as a trial of ubiquitous service coordination by converging a variety of communication devices. This is also the main purpose of our framework proposed in this paper. More specifically, the aim of our research is the realization of ubiquitous service coordination by adopting active rules that can interact among distributed service devices and activate each other via standardized eventing messages.

Web services technology is the most effective means for achieving inter-operability among heterogeneous systems, furthermore, it is also considered as a promising communication protocol in ubiquitous computing networks [11, 39]. The ongoing work, such as Microsoft's invisible computing project [34], UPnP 2.0 device architecture [41], and OMA Web Services Enabler Release (OWSER) [35], shows substantial approaches to ubiquitous device communication based on web services technology. They assume the web services-enabled devices that can run across a diversity of mobile and wireless platforms. Microsoft's invisible computing platform attempts to realize seamless computing world for small devices by implementing web services on a chip of customer smart devices. UPnP (Univeral Plug and Play) is a distributed, open network architecture where devices are connected directly each other at home, office, and public spaces, and the UPnP version 2.0 was released by adopting a broad web services technology. The mobile web services working group of OMA (Open Mobile Alliance) has published the OWSER specification to define necessary infrastructure for offering web services in wireless network and device environment. Based on HTTP protocol and web technologies, they are anticipated to employ various web services technologies including service discovery, QoS policy, authentication, as well as service description and invocation. Those issues focus on the development of service description and communication protocols by which communication devices can interact in wireless and mobile network, while our framework concentrates on the effective coordination in such a wireless and mobile device environments.

The previous projects have provisioned the potential and usability of web services technology in ubiquitous environments. In this research, we assume that web services-enabled devices will increasingly spread out like those ubiquitous web services projects. On the basis of the web services technology, this research attempts to present the decentralized rule-based service coordination framework and the service conflict detection and resolution mechanism for ubiquitous web services-enabled devices.

2.2 Distributed Event-Driven Rules

Event-driven rules were introduced to provide rather passive traditional database systems with active functionality, for example, integrity enforcement and view materialization [12, 37]. Many research groups proposed active database systems, including SAMOS [22], Sentinel [12], DEVICE [4], EXACT [19], and HiPAC [18]. In such systems, triggering events are usually selected among data manipulating operations and transaction commands [37]. After that, event-driven rule management has been adopted for a variety of rule-based applications, like expert systems [4], workflow systems [9], collaborating agents [6, 27], and middleware [15].

Although event-driven rule management was mainly considered in centralized systems, recently it started to be applied to distributed and parallel database systems owing to the advantages of distinct and comprehensible rule descriptions [15, 43]. Vlahavas and Bassiliades summarized the important issues of parallel rule processing including condition matching and rule execution in expert systems and knoweldge base systems. The active rule management are recently adopted in broad distributed system area. Cilia and Buchmann [15] adopted active rules to describe business rules in heterogeneous e-business environments. The ECA rule can be a good alternative for rapid chaining business rules. Facca *et al.* [20] took an ECA based approach to develop adaptive web applications, and Kantere and Tsois [27] proposed an ECA rule management method for the data management in P2P network. Policy-based management is also an area where ECA rules, called the policy description languages (PDL), were adopted for distributed system coordination [10, 30]. For the policy-based framework, Shankar *et al.* [40] suggested ECA-P with post-conditions, which is the state of a system after processing the rule. They addressed weak points of previous ECA rules and also proposed the mechanism of static and dynamic conflict detection and resoultion for ubiquitous computing environments. However, their centralized approach to distributed application coordination has limitation in applying the mechanism to ubiquitous computing environments, mainly due to the properties of device capacities, communication overhead, adaptability to the mobility, and private resource management. Moreover, policy-based languages assuming the time granularity at run-time do not allow the ubiquitous devices to react on the instant events immediately. In this research, we have attempted decentralized, adaptive, and lightweight ECA rule processing mechanism for ubiquitous service devices by means of web services eventing technology.

There are several recent researches on event-based service computing using web services technology. SCXML (State Chart XML) specification [2] provides a means by which a control mechanism can be described by use of distributed finite state machines. Yet, it focuses on the behavioral description of an individual device rather than the mechanism consisting of multiple devices. In addition, recently proposed specifications, such as WS-Eventing and WS-Addressing are also accelerating the implementation of service-oriented ubiquitous computing. WS-Addressing [8] offers a promising way of the endpoint descriptions of service partners for synchronous and asynchronous communications. On the basis of the specification, WS-Eventing [3] provides the messaging protocols for the publish / subscribe mechanism among web service applications. Event issues are

subscribed by the protocols, and the event notifications are delivered on SOAP messaging. The content of the notifications can be described without restrictions for a specific application. The proposal can be used to develop an extended protocol to support event-driven communication mechanism via web services technology. In our research, the WS-Eventing mechanism was adopted as the means of event notification to external services. When a new ECA rule is registered to a device, if the rule contains new external event in the event part (or any new event notification in the action part), the device will subscribe the event to the external event source (or create the new event publication). The WS-Eventing standards provide the message set for the publish / subscribe mechanism on the basis of web services technology, which includes renewing, state checking, and faults messages, as well as subscription and unsubscription ones.

Meanwhile, multiple rule processing may cause unexpected results especially in distributed parallel rule systems. Several authors have reported significant results on managing conflicts among the ECA rules. Chomicki *et al.* [14] and Shankar *et al.* [40] presented logic-based approaches to handling rule conflicts. However, since they assume the time granularity at run-time, their approaches to rule conflict detection and resolution are not so effective in ubiquitous environments as in policy-based management. They do not deal with quite instant response on the triggering events, and furthermore they may result in some distortion in case that two significant events that occurred with the small time gap are decided to cause a service conflict. On the other hand, in our proposed mechanism, only the rules that may be triggered by the same event are checked at design-time. Besides, the potential rules that may cause any conflict are marked at the design-time, and they are then checked again at run-time. The mechanism is effective in that it can reduce the conflict checking time and network overhead in run-time, and also acceptable to ubiquitous scenarios. In other words, we defined the properties of ECA rules sets and the categorization of rule conflicts to address the characteristics of ubiquitous service devices, and we presented the mechanism that can effectively handle the conflicts.

3. WS-ECA Rules Management for Ubiquitous Service Devices

In the proposed framework, service devices are surrounded by a lot of event sources and service providers, which provide event notifications and public web services, respectively, on the basis of web services technology.The service devices are assumed to interact each other through the events generated via publish / subscribe mechanism and to invoke web services via request / response mechanism.

Following the structure of the traditional ECA rules, the proposed language for rule descriptions, named WS-ECA, consists of events, conditions, and actions [25]. The event is a notification message that can be one of four primitive event types, namely (i) *internal event* if it is generated by a device itself, (ii) *external event* if it is delivered from other devices, (iii) *time event* that occurs after certain time has elapsed, and (iv) *service event* that is generated at the point of invoking a service of a local device. The condition is a boolean expression that must be satisfied for some action of a device to occur. It is defined by use of event variables contained in an event

message or device variables maintained by a device. Finally, the action represents an instruction carried out by a device, which includes primitive actions such as web service invocation and event generation.

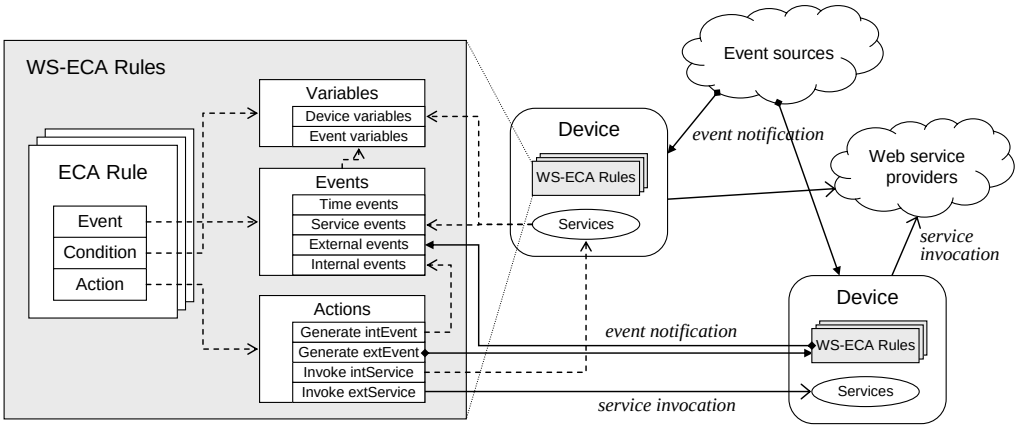


Figure 1. The structure of WS-ECA rules

Figure 1 shows the proposed structure of WS-ECA rules. There are two types of interactions performed by a device: one is the service invocation which is carried out by sending a SOAP message to an external web service provider, and the other is event notification which is transferred via WS-Eventing protocol. WS-Eventing specification defines a publish / subscribe messaging protocol for delivering subscription, notification, and fault messages to implement an event-driven interactions based on web services. Service invocation is used to request a well-defined service to the static service provider, while event notification is done to publish an event to multiple subscribers, which may have already subscribed or be going to subscribe the interested event in the future. The latter is more loosely-coupled and dynamic so that it is proper to mobile devices and it is also efficient to send an occurrence to multiple devices.

Furthermore, our framework introduces a global rule manager (GRM) for the purpose of robust and effective decentralized rule processing. GRM complements the shortcoming of decentralized service coordination by checking potential conflicts among the multiple rules at design-time and offering resolution mechanism at run-time. The decentralized devices maintain a set of WS-ECA rules defined by users, and activate the rules in response to the triggering events at run-time. On the other hand, the GRM is responsible for analyzing the consistency of new rules at design-time, and then for resolving -rule conflicts at run-time. For instance, a personal computer can play a role of the GRM in the home network while a variety of home appliances are service devices that interact each other.

The proposed architecture for processing WS-ECA rules between the service devices and the GRM is shown in Figure 2. In the figure, the procedures, (1) to (5), are performed at design-time in order to verify WS-ECA rules and register them to devices. When a user requests the local rule manager of a service device to register a new rule, the rule verifier of GRM consults two rule conflict detectors, namely the static rule conflict detector

and the dynamic rule conflict detector, to check the rule (indicated as (3) and (4) in Figure 2). While the static rule conflict detector is responsible for checking inconsistencies of the new rule with the existing rules, the dynamic rule conflict detector examines any potential conflict at run-time and requires the user to resolve the conflict if it turns out that a conflict may occur during run-time. Only the rules that complete the verification are allowed to be registered to the device, and the rules that can lead to a conflict at run-time are marked by the dynamic rule conflict detector. Detailed algorithms used by these detectors will be presented in Section 5.

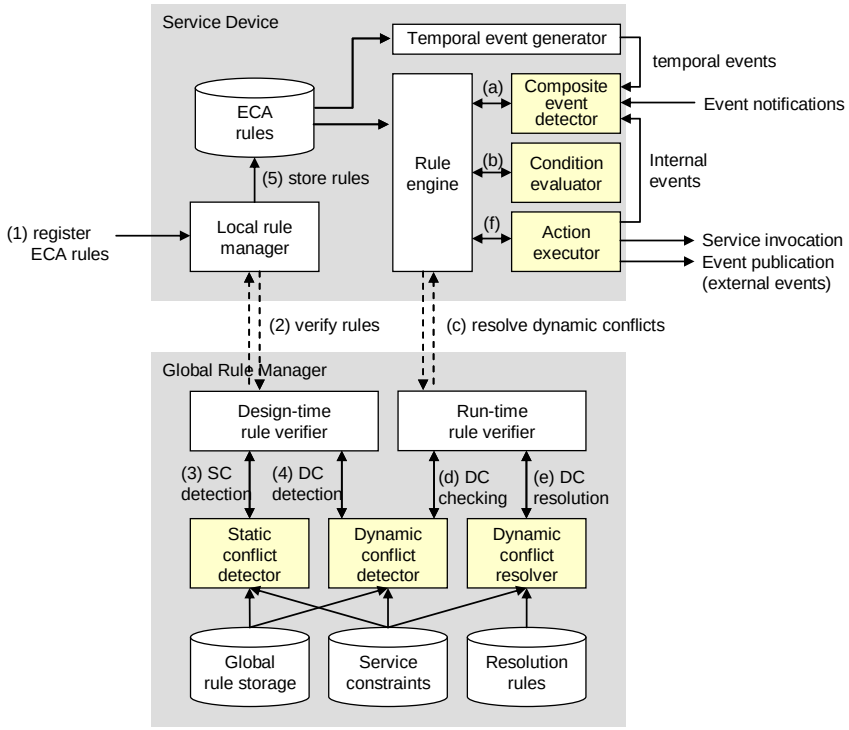


Figure 2. Architecture for WS-ECA rule processing

On the other hand, (a) to (f) in Figure 2 represent the run-time procedures carried out for processing the registered rules and resolving dynamic rule conflicts. The composite event detector embedded in a device waits for an event that can be either internal or external to the device, and then checks if the condition of the triggered rule is satisfied. When the condition is satisfied and the rule has no mark for potential dynamic conflicts, the corresponding action is executed immediately. Otherwise, the rule with a mark is checked by the dynamic conflict detector to see whether or not the conflict is imminent for a given state. If it is, the dynamic conflict resolver coordinates the devices with the conflicting rules by use of the predefined resolution rules. The resulting actions will be free from run-time conflicts.

4. WS-ECA: An ECA Rule Description Language for Web Service Devices

This section describes the XML schema of the proposed WS-ECA in detail. WS-ECA rules allow the web services-enabled devices to interact each other via event-based interactions. In particular, WS-ECA has been designed so that it can support (i) event passing by which certain events to be forwarded or broadcast to target devices, (ii) temporal reaction that allows different actions to be performed according to the occurrence times of the same event, and (iii) rule chaining where complex rules can be decomposed into several simpler rules.

```
<ECARule name="xs:NCName" targetNampespace="xs:anyURI"
     xmlns="http://di.snu.ac.kr/2005/eca/"
     xmlns:xs="http://www.w3.org/2001/XMLSchema" >
     <variables>? <variable ... />+ </variables>
     <events> event+ </events>
     <actions> action+ </actions>
     <rules>
          <rule name="xs:NCName">+
              <event name="xs:QName"/>
              <condition expression="XPath Expression"/>
              <action name="xs:QName"/>
          </rule>
     </rules>
<ECARule>
```

Figure 3. The overall WS-ECA schema

Figure 3 shows the overall XML schema of WS-ECA. WS-ECA consists of a series of definitions for variables, events, and actions from which rules are constructed after specifying conditions. Furthermore, it supports the primitive events and actions as well as the composite ones for distributed coordination of ubiquitous service devices. In what follows, we describe each element in detail.

4.1 Event

An event is an incident that triggers a rule. It is categorized into four primitive events, namely internal event, external event, time event, and service event. Internal event is generated by the internal system components of a device, and it is defined to recognize the state change of a device or to trigger other rules. External event is generated from a remote publishing device and transmitted to a subscriber device via WS-Eventing protocol. Time event occurs when the timer of a device reaches some specific point in time. It is further classified into three types: *absolute*, *periodic*, and *relative*. The time event of absolute type occurs once whereas the event of periodic type occurs periodically. The time event of relative type is defined in relation with some other event by use of 'before' and 'after' operators. Finally, service event is specified in reference to a specific service invocation action defined for a device. It can be one of two types: before and after. The service event of before (after, respectively) type is generated before (after, respectively) the specified service of a device starts (finishes,

respectively).

Not only the primitive events introduced above, WS-ECA supports specification of a composite event based on them by use of the following logical operators:

- *disjunction* ($e_1 \vee e_2 \vee \ldots \vee e_n$): The composite event of type "OR" has more than one sub-events, and it requires that at least one of the sub-events must occur during some specific time interval.
- *conjunction* ($e_1 \wedge e_2 \wedge \ldots \wedge e_n$): The composite event of type "AND" has more than one sub-events, and it requires that all of the sub-events must occur during some specific time interval.
- *serialization* ($e_1;e_2;\ldots;e_n$): The composite event of type "SEQ" has more than one sub-events, and it requires that all of the sub-events must occur sequentially during some specific time interval.
- *negation* ($\neg e$): The composite event of type "NOT" has only one sub-event, and it requires that the sub-event must not occur during some specific time interval. The event can be used only in another composite event of *conjunction* or *serialization*.

```
<events>
   <timeEvent type="once" name="xs:NCName"> xs:dateTime </timeEvent>
   <timeEvent type="periodic" name="xs:NCName" unit="xs:duration">
       xs:dateTime </timeEvent>
   <timeEvent type="relative" name="xs:NCName" baseEvent="xs:NCName"
       interval="xs:duration"/>

   <intEvent name="xs:NCName"/>
   <extEvent name="xs:NCName" eventID="xs:anyURI"/>

   <svcEvent type="before" name="xs:NCName" service="xs:QName"/>
   <svcEvent type="after" name="xs:NCName" service="xs:QName"/>

   <compositeEvent type="OR" name="xs:NCName" TTL="xs:duration">
       event+ </compositeEvent>
   <compositeEvent type="AND" name="xs:NCName" TTL="xs:duration">
        event+ </compositeEvent>
   <compositeEvent type="SEQ" name="xs:NCName" TTL="xs:duration">
        event+ </compositeEvent>
   <compositeEvent type="NOT" name="xs:NCName" TTL="xs:duration">
       event+ </compositeEvent>
</events>
```

Figure 4. Event schema of WS-ECA.

Figure 4 shows the event schema defined in WS-ECA for specifying the primitive events as well as composite events. The time interval necessary for a composite event is denoted as TTL. We remark that event composition can be done recursively to represent complex event structures.

4.2 Condition

The condition of WS-ECA rules is a boolean statement that must be satisfied in order to activate the rule. It is described in terms of an XPath expression [16], and the expression in a condition may refer to values from the event definition and use the variables defined in a WS-ECA document.

```
<variables>?
  <variable name="xs:NCName" deviceVar="xs:QName"?
     eventVar="eca:getVariable(event QName, path PathExpr)"? />+
</variables>
```

Figure 5. Variable schema of WS-ECA.

Table 1. Extension functions to XPath's built-in functions

Functions	return type	return value
`eca:getVariable(event QName, path PathExpr)`	`xs:any`	Specific value from an event variable
`eca:getDateTime(event QName)`	`xs:dateTime`	Date and time information

The syntax for the variables is presented in Figure 5. Variables may refer to specific elements of an event defined in WS-ECA rules (called *event variables*) or they may be used to represent a state of a device (called *device variables*). They can be also used to express the conditions or to assign necessary input data for actions such as service invocations and event generation. We define two extension functions to assign the value to a variable as shown in Table 1. The first function extracts a specific value from an event variable, and the second returns the date and time information.

4.3 Action

The action part of the WS-ECA contains the instruction that is executed when a triggered rule is activated. The role of action parts include processing the user's ultimate service by service invocation, triggering another rules by internal event creation, and interacting with other devices by event publication. A primitive action can be one of three types: (i) *invokeService (service)* that invokes an internal or external service, (ii) *createExtEvent (event)* that generates an external event and publishes it to subscribed devices, and (iii) *createIntEvent (event)* that generates an internal event and triggers other rules in the device. An *invokeService* action supports the request / reply mechanism while a *createExtEvent* action does the publish / subscribe mechanism in the device communications. On the other hand, a *createIntEvent* action is utilzed in chaining several rules to describe more complicated service logics.

Contrary to the event part, the action part supports only one kind of composite action. The conjunctive

action is defined by using more than one primitive action. A *conjunctiive action* ($a_1 \wedge a_2 \wedge \ldots \wedge a_n$) requires all sub-actions to be executed. Figure 6 shows the proposed schema for the action element. The conjunctive expression of the action part was introduced in order to reduce the labor to describe duplicatively several rules with the same event and condition parts. On the other hand, disjunctive expression of multiple primitive actions may cause some ambiguity because of the priorities among the sub-actions and sequence of their executions. However, the conditional disjunction of several actions can be express in separated rules that have the different condition parts. Furthermore, we do not consider sequential composite actions in WS-ECA schema since they can be defined by chaining a series of WS-ECA rules.

```
<actions>
    <invoke name="xs:NCName" service="xs:QName"> xs:any </invoke>
    <createIntEvent name="xs:NCName" intEvent="xs:NCName"> xs:any </createIntEvent>
    <createExtEvent name="xs:NCName" extEvent="xs:anyURI"> xs:any </createExtEvent>
    <compositeAction name="xs:NCName" operator="AND"> action+ </compositeAction>
</actions>
```

Figure 6. Action schema of WS-ECA.

4.4 Example

As a motivating example, we consider the following scenario, "morning cook service", illustrated in Figure 7: A user set the get-up time to 7:00 AM on the alarm clock before sleeping. In the next morning, the clock informs the rice cooker of '20 minutes before get-up'. The cooker starts to cook, and if rice is not enough, it alerts to the user at his/her get-up time. When the cooking is completed, the cooker informs the coffee maker, and the coffee maker will start to prepare a morning coffee after 10 minutes.

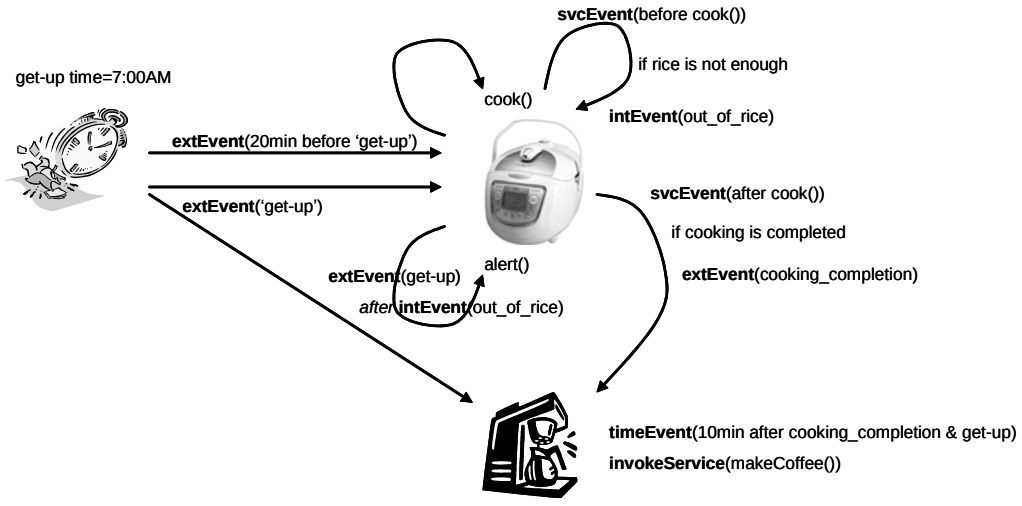


Figure 7. Morning cook service example

The rules described above can be presented as shown in Figure 8. For the alarm-clock, two time events are periodically generated, and the rules are provided to check if it is not a holiday. Each rule, when the condition is satisfied, will execute an action that publishes an external event to the subscribing devices, namely the rice-cooker and the coffee-maker. As for the rice-cooker, the first rule is to invoke the cook service when it receives the alarm of '20min before get-up'. The next two rules of the rice-cooker, one for generating an event before the cook service and the other for generating an event after the cook service, may respectively generate an internal event 'out_of_rice' and external event 'cooking_completion', depending on the condition. The last rule of the rice-cooker, which is triggered by a composite event enabled when the external event 'alarm' and the internal event 'out_of_rice' are received sequentially within an hour, will execute an alert service if the condition is satisfied. Finally, the rule defined for the coffee-maker will require the makeCoffee service to be started 10 minutes after it receives a composite event that is enabled when two external events 'cooking_completion' and 'alarm" are received in order within 30 minutes.

Alarm-Clock
on **timeEvent**(20min before getting up) ***if*** it is not holiday ***do*** **createExtEvent**(alarm(contents='20min before get-up'))
on **timeEvent**(at 7:00AM every day) ***if*** it is not holiday ***do*** **createExtEvent**(alarm(contents='get-up'))
Rice-Cooker
on **extEvent**(alarm) ***if*** alarm.contents='20min before get-up' ***do*** **invokeService**(cook())
on **svcEvent**(before cook()) ***if*** rice is not enough ***do*** **createIntEvent**(out_of_rice)
on **svcEvent**(after cook()) ***if*** cooking is succeeded. ***do*** **createExtEvent**(cooking_completion)
on **compositeEvent**(**intEvent**(out_of_rice) after **extEvent**(alarm) within 1hr) ***if*** alarm.contents='get-up' ***do*** **invokeService**(alert("out of rice"))
Coffee-Maker
on **timeEvent**(10min after **compositeEvent**(**extEvent**(cooking-completion) and **extEvent**(alarm) within 30min)) ***if*** alarm.contents='get-up' ***do*** **invokeService**(makeCoffee())

Figure 8. ECA rule example of morning cooking service

Having introduced the motivating example, we proceed to specify the rules in terms of the proposed WS-ECA language. First, the events for the alarm clock are defined by using the time events of periodic and relative types as follows. Note that the periodic time event 'get-up-time' occurs every morning, while the relative time event 'before-get-up-time' occurs in reference to 'get-up-time'. To describe the temporal situation, the `unit` and `interval` attributes of the `timeEvent` elements are expressed in `duration` of XPath specification [16]. That is, `unit="P1D"` of the periodic time event means that it occurs 'every day', and `interval="-PT20M"` of the relative time event means that it occurs '20 minutes before' the base event.

```
<events>
   <timeEvent type="periodic" name="get-up-time" unit="P1D"> 7:00 AM </timeEvent>
   <timeEvent type="relative" name="before-get-up-time"
   baseEvent="get-up-time" interval="-PT20M"/>
</events>
```

Next, the events for the rice cooker show the usage of other event types. The before and after service events defined for the cooking service are introduced, and the internal and external services are declared so that they can be used in the following composite event, 'get-up-after-out-of-rice' which will be triggered when the external event 'alarm' is followed by the internal event 'cooking' within 1 hour (i.e. TTL="PT1H").

```
<events>
   <svcEvent type="before" name="before-cooking" service="rc:cook"/>
   <svcEvent type="after" name="after-cooking" service="rc:cook"/>
   <intEvent name="cooking"/>
   <extEvent name="alarm" eventID="..."/>
   <compositeEvent type="SEQ" name="get-up-after-out-of-rice" TTL="PT1H">
       <event name="cooking"/><event name="alarm"/>
   </compositeEvent>
</events>
```

The actions defined below for the rice cooker contain three types of primitive actions, including *createIntEvent*, *createExtEvent*, and *invokeService*. The 'start-cooking' and 'detect-out-of-rice' actions respectively generate two internal events, 'cooking' and 'out-of-rice', which will be used to chain with other rules in the WS-ECA specification. The 'complete-cooking' action generates an external event with a URI that will be published to all subscribed external devices. Finally, the 'invoke-cooking' and 'alert-out-of-rice' actions will respectively invoke internal and external services based on the WSDL document with namespace 'rc:'. A complete specification of WS-ECA rules for the example presented in this section is provided in Appendix.

```
<actions>
    <createIntEvent name="start-cooking" intEvent="cooking"/>
    <createIntEvent name="detect-out-of-rice" intEvent="out-of-rice"/>
    <createExtEvent name="complete-cooking"
    extEvent="http://di.snu.ac.kr/event/rice-cooker/cooking-completion"/>
```

```
    <invoke name="invoke-cooking" service="rc:cook"/>
    <invoke name="alert-out-of-rice" service="rc:alert">
        <rc:contents>out of rice</rc:contents>
    </invoke>
</actions>
```

5. Conflict Detection and Resolution of ECA Rules

We assume that WS-ECA rules can be defined by multiple users and registered to distributed devices. As a result, some rules may have discrepancies with each other and they may cause conflicts when triggered and executed in a distributed manner at run-time. In this section, we present a framework for conflict detection and resolution of WS-ECA rules implemented in distributed devices.

We first introduce the notion of a *service constraint*, which represents a set of service invocation actions that are not allowed to occur at the same time. For example, an event that leads to both turning on the radio and turning it off is not permitted, and turning on a heater while an air conditioner is working may not be desired by a user. An event is not allowed if it can result in the execution of actions that belong to a service constraint, and we say that a set of WS-ECA rules has a *conflict* if some of the rules triggered by an event can lead to violation of the service constraints.

In addition, we categorize the WS-ECA rule conflicts into two types according to the time when they are resolved. The first type is a *static conflict* that is examined at design-time to determine if there is a set of rules that violate service constraints. Once such rules are found, they cannot be registered together to the systems and they must be modified to avoid the service conflicts. The second is a *dynamic conflict* which is judged at run-time to see if there is a set of rules enabled coincidentally that may lead to the conflicts against the service constraints although no logical contradiction was identified at design-time. In case that some rules with a dynamic conflict are intended to register, Global Rule Manager (GRM) announces when the conflict will occur and what service constraints they will violate. The rules with a dynamic conflict will be allowed to be registered only after supplementing manually the corresponding resolution rules that can settle the dynamic conflicts at run-time. The resolution rules , which are also defined in WS-ECA rules, are triggered when any dynamic rule occurs at run-time. The description of the resolution rules will be introduced in detail in Section 5.3.

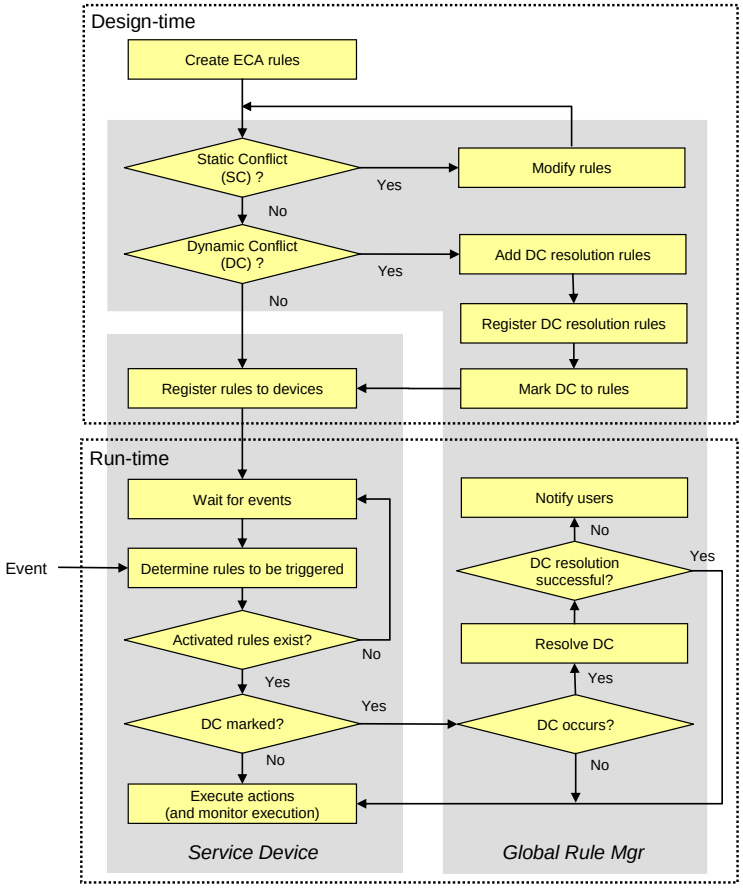


Figure 9. Framework for ECA rule conflict management

Figure 9 illustrates the overall framework for conflict detection and resolution in consideration for two types of WS-ECA rule conflicts. The proposed framework is composed of two phases of conflict management, namely the design-time conflict detection and run-time conflict detection and resolution. At the design-time phase, the rules created by a user are forwarded to GRM, which then checks if each rule causes any static conflict by comparing it with those registered already to devices. In case that any static conflict is detected, the user will be notified of the detailed conflict and requested to modify the rule to remove the conflict.

When no static conflict is identified, GRM subsequently checks if the rule can cause a dynamic conflict at run-time. If any potential dynamic conflict is detected, the user is notified of the service conflict (i.e., the service actions involved and the corresponding service constraints) and the conflict details (i.e., a set of events and conditions). In this case, the user must supply appropriate dynamic conflict resolution rules that can arrange the conflict situation at run-time. Only after registering these resolution rules to GRM, the new rule can be registered to a service device with a *mark* which indicates the rule may cause a dynamic conflict at run-time. The other rules involved in the dynamic conflict also need to be marked.

At run-time, the rule processor embedded in a service device waits for a triggering event. When a rule is triggered, it is checked whether or not the rule is marked with a dynamic conflict. If no marking is associated

with the rule, its actions will be executed immediately. Otherwise, the device needs to consult with GRM to see if a dynamic conflict is imminent. In case that no dynamic conflict is identified, the actions defined for the rule are executed. Otherwise, GRM sends the appropriate action instructions to the devices involved in the conflict through looking up the dynamic conflict resolution rules supplied at design-time.

5.1 Basic Concepts

In this section, we present formal semantics and properties of the proposed WS-ECA rules and provide necessary definitions on which the algorithms for conflict detection and resolution presented in Section 5.2 and Section 5.2 are based. First, we formally define the WS-ECA *normalized rule set*.

Definition 1 (Normalized rule set). *A normalized rule set, $\mathcal{RS}^{ws}$, is a finite set of WS-ECA rules, $R = (E^R, C^R, A^R)$, where E^R, C^R, and A^R respectively represent the sets of events, conditions, and actions defined as follows.*

1. $E^R = \{e \mid e = e_1;...;e_n$, where $e_i = e^{time}, e^{int}, e^{ext}$, or $e^{svc}, i = 1, \ldots, n\}$

Event set E^R consists of the disjunction of one or more serializations of events, where the serialization is expressed as $e_1;...;e_n$. An event e_i is one of the following types: time event e^{time}, internal event e^{int}, external event e^{ext}, and service event e^{svc}.

2. $C^R = \{c \mid c = c_1 \wedge \ldots \wedge c_m$, and $c_j = r_1 \ominus r_2, j = 1, \ldots, m\}$

Condition set C^R consists of the disjunction of zero or more conjunctions of condition predicates. Each conjunction is expressed as $c_1 \wedge \ldots \wedge c_m$, the condition predicate c_j is expressed as $r_1 \ominus r_2$, where $\ominus$ is an operator from the set $\{=, \neq, <, \leq, >, \geq\}$, and r_i, $i = 1, 2$, is a constant, an event variable, or a device variable.

3. $A^R = \{a \mid a = a^{svc}, a^{int}$, or $a^{ext}\}$

Action set A^R consists of the conjunction of one or more primitive actions, which can be invokeService $a^{svc}(svc)$, createIntEvent $a^{int}(e^{int})$, or createExtEvent $a^{ext}(e^{ext})$.

$\mathcal{RS}^{ws}$ represents the set of all WS-ECA rules defined for all devices in the system in a normalized form. The condition set bases the formalism on the work presented by Chomicki *et al.* [14] and extends it to allow both the conjunctions and the disjunctions of condition predicates in order to conform to WS-ECA language. The action set expresses the conjunction of the primitive actions which are required for execution when the rule is activated. Considering that WS-ECA supports conjunction for modeling a composite action, the action parts written in WS-ECA can be equivalently translated into the subset of A^R. As for the event set of $\mathcal{RS}^{ws}$, the following Lemma establishes that any composite event defined in WS-ECA by use of four logical operators (disjunction, conjunction, serialization, and negation presented in Section 4) can be transformed to a subset of E^R. Therefore,

any WS-ECA rule can be equivalently described in terms of the normalized rule set.

Lemma 1. *Any valid composite event defined in terms of WS-ECA can be equivalently translated to the subset of event set E^R of normalized rule set $\mathcal{RS}^{ws}$.*

Proof. A composite event is defined by use of four operators, disjunction $\vee$, conjunction $\wedge$, serialization ; , and negation $\neg$, and it satisfies the following properties:

(i) $e_1;(e_2;e_3) = (e_1;e_2);e_3$

(ii) $e_1;(e_2 \wedge e_3) = (e_1;e_2) \wedge (e_1;e_3)$

(iii) $e_1;(e_2 \vee e_3) = (e_1;e_2) \vee (e_1;e_3)$

(iv) $(e_1 \vee e_2) \wedge e_3 = (e_1 \wedge e_3) \vee (e_1 \wedge e_3)$

(v) $e_1 \wedge e_2 = (e_1 ; e_2) \vee (e_2 ; e_1)$

From the properties (i), (ii), and (iii), any valid composite event expression in WS-ECA can be transformed into the equivalent conjunctions and disjunctions of serializations. Subsequently, conjunctions can be re-written as disjunctions and serializations by using the properties (iv) and (v), resulting in the disjunction of serializations as defined in E^R.

Next, we present a series of definitions for the purpose of development of the rule conflict detection algorithms in the next section.

Definition 2 (Event dominance). *Let e^i and e^j be two serializations of events. e^i is said to be dominant over e^j (represented as $e^i \rightarrow^E e^j$) if the occurrence of e^i guarantees the occurrence e^j. That is, $e^i \rightarrow^E e^j$, if $e^i = e^i_1;...;e^i_n$, and $e^j = e^i_a;...; e^i_s;...; e^i_t;...;e^i_m$ $(1 \leq a < s < t < m \leq n)$.*

Definition 3 (Concurrability). *Let $\mathcal{RS}$ be a normalized rule set with two or more rules, and e^{final} be a primitive event. $\mathcal{RS}$ is said to be concurrable for a final event e^{final} if all rules in $\mathcal{RS}$ are triggered at the same time when e^{final} occurs. That is, $\mathcal{RS}$ is concurrable for e^{final} if $\forall R_i \in \mathcal{RS}$, $\exists e^i (= e^i_1;...;e^i_n) \in E^{Ri}$, such that $e^i_n = e^{final}$.*

Definition 4 (Co-triggerability). *Let $\mathcal{RS}$ be a normalized rule set with two or more rules, and e^{trig} be a primitive or composite event. $\mathcal{RS}$ is said to be co-triggerable for the triggering event e^{trig} if $\mathcal{RS}$ is concurrable for the final event of e^{trig} and e^{trig} triggers all the rules in $\mathcal{RS}$. That is, $\mathcal{RS}$ is co-triggerable for e^{trig}, where $e^{trig} (= e^{trig}_1;...;e^{trig}_n) \in E^{Rt}$, $R_t \in \mathcal{RS}$ if $\forall R_i \in \mathcal{RS}$ $(i \neq t)$, $\exists e^i (= e^i_1;...;e^i_m) \in E^{Ri}$, such that $e^i_m = e^{trig}_n$ and*

$e^{trig} \rightarrow^{E} e^{i}$.

Event dominance is defined to analyze the relationship among serializations of events. For example, for two serializations of events $e^a = e_1;e_2;e_3;e_4;e_5$ and $e^b = e_2;e_4$, the occurrence of e^a can guarantee that of e^b, therefore e^a is a dominant event over e^b. Furthermore, from the definition of event dominance, we can infer the event equivalence between two events that if $e^i = e^j$, $e^i \rightarrow^{E} e^j$ and $e^j \rightarrow^{E} e^i$. That is, two equivalent events are dominant over each other. Event dominance is used to identify the rules that can be triggered together by an event. When a normalized rule set is concurrable, all rules in the set have a serialization event with the same final event e^{final}. If e^{final} occurs after all the preceding events have occurred, all rules in the set will be triggered simultaneously. Therefore, it follows that if some rules in a concurrable rule set have service actions violating any service constraints, it may cause rule conflicts. On the other hand, if a normalized rule set is co-triggerable, all rules in the set have a serialization event that has the same final primitive event and also has the same dominant event e^{trig}, which is also an event of a rule in the set.

Note that, the rules in a co-triggerable rule set that have actions against service constraints necessarily lead to conflicts when the event e^{trig} happens and the required conditions are satisfied at run-time, and therefore they should not be allowed to be registered at design-time. However, the rules in a concurrable rule set that have actions against service constraints do not always result in a conflict when the event e^{final} occurs at run-time, since the preceding events of some rules may not have occurred. Hence, these rules should be checked at run-time to see if they actually lead to a conflict.

The rules triggered at the same time may have different conditions that need to be met, making some rules with potential conflicts actually result in conflicts or not depending on the condition. The next two definitions further characterize the potential conflicts in view of the conditions defined for a normalized rule set.

Definition 5 (Compatibility). *Let $\mathcal{RS}$ be a normalized rule set with two or more rules. $\mathcal{RS}$ is called compatible if the conditions in $\mathcal{RS}$ can be satisfied simultaneously. That is, $\mathcal{RS}$ is compatible if $c^1 \wedge \ldots \wedge c^m \neq false$, where $c^i \in C^{Ri}$, $R_i \in \mathcal{RS}$ $(1 \leq i \leq m$ and $C^{Ri} \neq \phi)$. Otherwise, $\mathcal{RS}$ is called incompatible.*

Definition 6 (Co-satisfiability). *Let $\mathcal{RS}$ be a normalized rule set with two or more rules, and c^{sat} be a condition of a rule in $\mathcal{RS}$. $\mathcal{RS}$ is called co-satisfiable for c^{sat} if c^{sat} can satisfy all rules in $\mathcal{RS}$. That is, $\mathcal{RS}$ is co-satisfiable for c^{sat}, where c^{sat} $(= c^{sat}_1 \wedge \ldots \wedge c^{sat}_m) \in C^{Rs}$ and $R_s \in \mathcal{RS}$, if $\forall R_i \in \mathcal{RS}$ $(i \neq s$ and $C^{Ri} \neq \varphi)$, $\exists c^i \in C^{Ri}$ such that $c^{sat} = c^i \wedge c^{any}$, where c^{any} is either a true condition or conjunction of any $c^{sat}_k (1 \leq k \leq m)$.*

If a normalized rule set is compatible, it means that it is possible for the rules in the set to be satisfied simultaneously in some case, whereas if the rule set is incompatible, all the rules in the set cannot be satisfied

simultaneously in any case. On the other hand, if a normalized rule set is co-satisfiable, a rule in the set has the condition c^{sat} that can also satisfy all the other rules in the set which have the same condition as c^{sat} or less constrained conditions than c^{sat}, including a null condition (i.e., *true* condition). A co-satisfiable rule set is compatible since there always exists a valid condition $c^{sat} = c^1 \wedge ... \wedge c^m \neq false$, *where* $c^i \in C^{Ri}$, $R_i \in \mathcal{RS}$, *and* $1 \leq i \leq m$.

Example 1. Consider a normalized rule R_0 whose condition is an empty set, which means true condition. R_0 is compatible with any other rule since the true condition can be satisfied simultaneously with any other conditions. In addition, R_0 is also co-satisfiable with other rules because conditions of the other rules can satisfy the true condition of the rule R_0, (i.e., conditions of the other rules become c^{sat} in this case). As another example, consider three rules R_1, R_2, and R_3, such that $C^{R1} = \{c_1, c_2\}$, $C^{R2} = \{c_1 \wedge c_3\}$, and $C^{R3} = \{\neg c_3\}$. Out of all combinations of these, the only co-satisfiable rule set is $\{R_1, R_2\}$ for $c^{sat} = c_1 \wedge c_3$, and the compatible rule sets are $\{R_1, R_2\}$ and $\{R_1, R_3\}$, while the incompatible rule sets are $\{R_2, R_3\}$ and $\{R_1, R_2, R_3\}$ due to the fact that $(c_1 \wedge c_3) \wedge (\neg c_3)$ = false.

5.2 Rule Conflict Detection

Service constraints are defined as conjunctions of two or more service actions. The constraints are stored in GRM, and they are used as criteria for judging service conflicts. In the following two definitions, we formally define the service constraints and rule conflicts.

Definition 7 (Service constraints). *Service constraints* $\mathcal{AC}^{ws}$ *is a power set of the service action set such that* $AC = \{ a_1, ..., a_n \mid a_1 \wedge ... \wedge a_n$ *are not allowed to be activated simultaneously and* $a_i = a^{svc}$, $i = 1, ..., n\} \in \mathcal{AC}^{ws}$.

Definition 8 (Rule conflict). *Let* $\mathcal{RS}^{trig}$ *be a normalized rule set of which the rules are triggered on the occurrence of a primitive event. It is said that* $\mathcal{RS}^{trig}$ *has a rule conflict if some of their service actions are contained in one of the elements of* $\mathcal{AC}^{ws}$.

The rules can be triggered directly or indirectly. We say that a rule is directly triggered on the occurrence of a primitive event when its event specification has one or more event serializations whose final event is the primitive event. On the other hand, a rule is said to be indirectly triggered if it is triggered by the generated events from *createIntEvent* or *createExtEvent* on the occurrence of a primitive event. In what follows, we

elaborate more on the two types of rule conflicts and then present algorithms for detecting conflicts.

5.2.1 Static Conflict Detection

A normalized rule set is said to have a static conflict if there exists a situation in which triggering rules causes a service conflict. Depending on whether the rules with potential conflicts are triggered directly or indirectly by an event, the static conflict is further divided into two types, namely *absolute conflict* and *chained conflict*. First, the absolute conflict is defined as follows.

Definition 9 (Absolute conflict). *Let $\mathcal{RS}^{trig}$ be a co-triggerable and co-satisfiable set of normalized rules. It is said that $\mathcal{RS}^{trig}$ has an absolute conflict if the rules have service actions that violate a service constraint AC of $\mathcal{AC}^{WS}$. Formally, a co-triggerable and co-satisfiable set $\mathcal{RS}^{trig}$ has an absolute conflict if $\exists AC \in \mathcal{AC}^{WS}$ such that $AC \subset \{ a^{svc} \mid a^{svc} \in A^{Ri}, R^i \in \mathcal{RS}^{trig} \}$.*

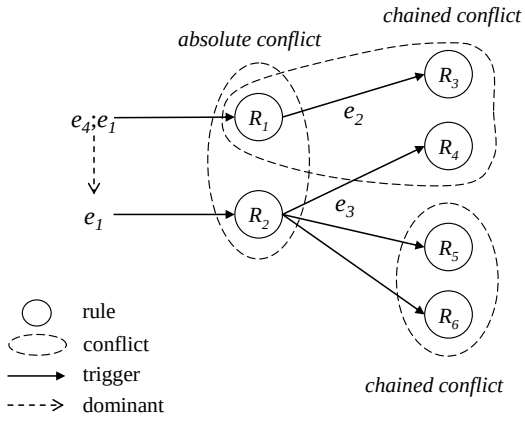
In the WS-ECA language, there are two primitive actions, *createExtEvent* and *createIntEvent*, that respectively generate an event to trigger other rules in an external device and internal device. The rules indirectly triggered by such an event generation action may also cause a service conflict with each other or with directly triggered rules. This type of conflict is called chained conflict which is formally defined in the following definition.

Definition 10 (Chained conflict). *Let $\mathcal{RS}^{trig'}$ be a co-triggerable and co-satisfiable set of normalized rules without absolute conflict, and $\mathcal{RS}^{chain}$ be a set of normalized rules that are co-satisfiable with $\mathcal{RS}^{trig}$ and can be triggered by an event generated from a rule of $\mathcal{RS}^{trig'}$ or from another rule of $\mathcal{RS}^{chain}$. It is said that $\mathcal{RS}^{trig'} \cup \mathcal{RS}^{chain}$ has a chained conflict if the rules in $\mathcal{RS}^{trig'} \cup \mathcal{RS}^{chain}$ have service actions violating a service constraint. Specifically, $\mathcal{RS}^{trig'} \cup \mathcal{RS}^{chain}$ has a chained conflict if $\exists AC \in \mathcal{AC}^{WS}$, such that $AC \subset \{ a^{svc} \mid a^{svc} \in A^{Ri}, R^i \in \mathcal{RS}^{trig'} \cup \mathcal{RS}^{chain} \}$.*

From the above definitions of absolute and chained conflicts, it follows that a normalized rule set with an absolute or chained conflict necessarily has a static conflict since the co-triggerability and co-satisfiability of the rule set implies the existence of the triggering event e^{trig} and the satisfiable condition c^{sat}.

Example 2. An example of static conflict is illustrated in the rule triggering graph of Figure 10, where all rules are co-triggerable for the event $e_4;e_1$, which is a dominant event over e_1. The table shows three parts of each rule. Note that the event set is disjunction of serialized events while the action set is conjunction of primitive actions. For example, rule R_2 can be triggered by any event of e_1 and e_5, and it requires to activate both actions $a^{svc}{}_1$

and $a^{int}(e_2)$. In the diagram, directly triggered rule set $\{R_1, R_2\}$, which is co-satisfiable for the condition $c_1 \wedge c_3$, has an absolute conflict, since they allow the actions against the service constraint AC_1. Furthermore, the directly or indirectly triggered rule set $\{R_1, R_3, R_4\}$, which is co-satisfiable for the condition $c_1 \wedge c_3$, makes a chained conflict against AC_2, while the indirectly triggered rule set $\{R_5, R_6\}$, which is co-satisfiable for the condition c_1, also results in a chained conflict against AC_3.



$\mathcal{RS}^{trig}$	E^R	C^R	A^R
R_1	$\{e_4;e_1\}$	$\{c_1 \wedge c_3\}$	$\{a^{svc}_1, a^{int}(e_2)\}$
R_2	$\{e_1, e_5\}$	$\{\}$	$\{a^{svc}_2, a^{int}(e_3)\}$
R_3	$\{e_2\}$	$\{c_3\}$	$\{a^{svc}_3\}$
R_4	$\{e_3\}$	$\{\}$	$\{a^{svc}_4\}$
R_5	$\{e_3, e_6\}$	$\{c_1\}$	$\{a^{svc}_5\}$
R_6	$\{e_3\}$	$\{\}$	$\{a^{svc}_6\}$

$\mathcal{AC}^{ws}$	action constraints
AC_1	$\{a^{svc}_1, a^{svc}_2\}$
AC_2	$\{a^{svc}_1, a^{svc}_3, a^{svc}_4\}$
AC_3	$\{a^{svc}_5, a^{svc}_6\}$

Figure 10. Absolute and chained conflict in a rule triggering graph

In the proposed framework, GRM detects static conflicts of rules by considering the other existing rules in all devices of the ubiquitous system when a new rule needs to be registered to a service device. Figure 11 shows the proposed algorithm for static conflict detection.

Input: A new rule R_{new}, a normalized rule set $\mathcal{RS}^{ws}$, a service constraints set $\mathcal{AC}^{ws}$, a conditional action set $\mathcal{cA}^{ws}$

Output: A static conflict set RC^{static}, where RC^{static}:={ (e, AC, RS^{static}) | triggering event e, service constraint AC, and conflicting rule set RS^{static}}

Algorithm *static_conflict_detection*(**in**(R_{new}, $\mathcal{RS}^{ws}$, $\mathcal{AC}^{ws}$), **out**(RC^{static}))

1: RC^{static}:={};
2: cA^{Rnew} := *makeCondActs*(R^{new}); // *make a conditional action set of the new rule*
3: **for** each $e \in E^{Rnew}$ **do**
4: *MDE(e)* := *findMDE(e)*; // *find the most dominant ones of the co-triggering events of e*
5: **if** $MDE(e)=\phi$ **then** *checkConflict(e, cA^Rnew)*; // *if there is no co-triggering event of e*
6: **else**
7: **for** each $e^{dom} \in MDE(e)$ **do**
8: $cA(e^{dom})=cA(e^{dom}) \cup cA^{Rnew}$;
9: *checkConflict*(e^{dom}, $cA(e^{dom})$);
10: **end for**
11: **end if**
12:**end for**
13:**if** $RC^{static}=\phi$ **then** *updateCondActs*(E^{Rnew});
14:**return** RC^{static};

Function *makeCondActs*(R^{new})

```
cA={};
for each a∈A^Rnew do
    if a=a^svc(svc) then cA = cA∪{(a, C^Rnew)};
    else if a=a^int(e^trig) || a=a^ext(e^trig) then cA = cA∪{(a^t, C^t ∧C^Rnew)| (a^t, C^t)∈cA(e^trig)};
end for
return cA;
```

```
Function checkConflict(e, cA)
for each AC∈𝒜𝒞^ws do
    if AC⊂{a | (a, C)∈cA} then
        cA^conflict:={(a, C)∈cA | a∈ AC};
        RS^static:={ R∈ℛ𝒮^ws | (a, C)∈cA^conflict and a∈A^R};
        if checkCoSatisfiability(cA^conflict)=true then RC^static:= RC^static ∪{(e, AC, RS^static)};
    end if
end for
return RC^static;
```

Figure 11. Algorithm *static_conflict_detection*

To start the algorithm *static_conflict_detection*, a new rule R_{new} is given with a normalized rules set $\mathcal{RS}^{ws}$, a service constraint set $\mathcal{AC}^{ws}$, and a conditional action set $\mathcal{cA}^{ws}$, which is updated for every rule insertion. The output of the algorithm is a static rule conflict set RC^{static}. First, the algorithm constructs the conditional action set *cA* from the new rule by function *makeCondActs*(R^{new}) (Line 2). The function identifies the service actions of both the new rule R_{new} and the rules that can be triggered by R_{new}, and then it returns a set of conditional actions (*a*, *C*), where *a* is a service action and *C* is the condition set that must be satisfied to execute the action *a*.

Subsequently, the algorithm starts to check for a static rule conflict for each serialization event $e \in E^{Rnew}$ (Lines 3-12). Static conflict can arise in two types of co-triggerable rule sets related to event *e*. The first is a set of rules that can be triggered by *e*, and the second is a set of the rules that can be triggered by e^{dom}, the dominant co-triggering events of *e*. In particular, in the second case, we have only to check the rules triggered by the most dominant co-triggering events. This is because the rule set of the most dominant co-triggerable events of *e* completely includes those of other co-triggerable events of *e*. It follows that if there is no static conflict for the most dominant co-triggering events, there is no static conflict for the other co-triggering events, as well as event *e*. In Line 4, function *findMDE(e)* finds the most dominant co-triggering events for *e* in previous event set. For instance, let $\{e_4, e_1;e_4, e_2;e_4, e_1;e_3;e_4, e_2;e_3;e_4\}$ be a co-triggering event set of e_4. The most dominant events of e_4 in this case are $e_1;e_3;e_4$ and $e_2;e_3;e_4$.

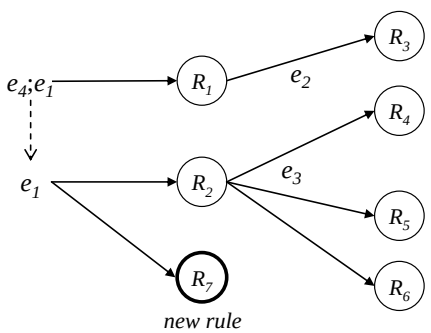
In case of no co-triggering event of the event *e*, we have only to check if the conditional action set of the new rule cA^{Rnew} has any service conflict in itself through the function *checkConflict*(*e*, *cA*) (Line 5). Otherwise, for all the most dominant co-triggering events $e^{dom} \in MDE(e)$, we take a union of $cA(e^{dom})$ and cA^{Rnew} and then check the service conflict (Lines 7-10).

Finally, if the new rule have no static conflict, the algorithm updates the conditional action sets for all e^{dom} of each event $e \in E^{Rnew}$ (Lines 13).

The algorithm uses function *checkConflict*(*e, cA*) in order to check static conflicts in conditional action sets. In this function, if the conditional action set includes any service constraint, the algorithm tests the co-satisfiablity of the rules with the conflicting actions by using function *checkCoSatisfiability*($cA^{conflict}$) which checks the co-satifiability of the conditional actions $cA^{conflict}$ based on the service constraint $AC \in \mathcal{AC}^{ws}$. The function first finds all combinations of the conditions that can result in an action in the service constraint *AC*, and then it checks co-satisfiability of all combinations of conditions. If there is a combination that turns out be co-satisfiable, it means that there exists a co-satisfying condition c^{sat}, where service actions in cA^{static} triggered by an event *e* will necessarily violate the service constraint *AC*.

The performance of the algorithm *static_conflict_detection* depends on the size of the normalized rule set and and the service constraints set. The complexity is $O(n_e n_a m)$, where n_e and n_a are the number of events and actions in the rule set, respectively, and m is the number of service constraints. This follows the observation that the total number of dominant events is lower than n_e (i.e., $|MDE(e)|<n_e$), and the traversals in the function *checkConflicts(e, cA)* take the time $O(n_a m)$ because the size of conditional action set is lower than n_a and the size of service constraints set is m (i.e. $|cA|<n_a$ and $|\mathcal{AC}^{ws}|=m$).

Example 3. We demonstrate the proposed algorithm through an example. Suppose that a new rule R_7 is inserted to Example 2 and $\mathcal{AC}^{ws}$ has only one constraint, as illustrated in Figure 12. The event specification E^{R7} has only one event of e_1, which has the co-triggering event $e_4;e_1$ (i.e. $E^{R7}=\{e_1\}$ and $MDE(e_1)=\{e_4;e_1\}$). Therefore it is not necessary to make a new conditional action set of e_1, since the set will be included in the set of $e_4;e_1$ (i.e. $cA(e_1) \subset cA(e_4;e_1)$). From Figure 10, we obtain $cA(e_4;e_1) = cA^{R1} \cup cA^{R2}$, where $cA^{R1} = \{(a^{svc}_1,\{c_1 \wedge c_3\}), (a^{svc}_3,\{c_1 \wedge c_3\})\}$ and $cA^{R2} = \{(a^{svc}_2,\{\}), (a^{svc}_4,\{\}), (a^{svc}_5,\{c_1\}), (a^{svc}_6,\{\})\}$. Next, we get $cA'(e_4;e_1) = \{(a^{svc}_1,\{c_1 \wedge c_3\}), (a^{svc}_2,\{\}), (a^{svc}_3,\{c_1 \wedge c_3\}), (a^{svc}_4,\{\}), (a^{svc}_5,\{c_1\}), (a^{svc}_6,\{\}), (a^{svc}_7,\{c_1 \wedge c_7\})\}$ since $cA'(e_4;e_1) = cA(e_4;e_1) \cup cA^{R7}$ and $cA^{R7} = \{(a^{svc}_7,\{c_1 \wedge c_7\})\}$. Finally, we obtain a static conflict set RC^{static} from the result of function *checkConflict*($cA'(e_4;e_1)$). The static conflict set is $RC^{static} = \{ (e_4;e_1, \{a^{svc}_5,a^{svc}_6,a^{svc}_7\}, \{R_5,R_6,R_7\}) \}$, which means that R_7 has only one static conflict in which the triggering event $e_4;e_1$ causes a service conflict against the constraint $\{a^{svc}_5, a^{svc}_6, a^{svc}_7\}$ in the rule set $\{R_5, R_6, R_7\}$.



$\mathcal{RS}^{trig}$	E^R	C^R	A^R
R_1	$\{e_4;e_1\}$	$\{c_1 \wedge c_3\}$	$\{a^{svc}_1, a^{int}(e_2)\}$
R_2	$\{e_1, e_5\}$	$\{\}$	$\{a^{svc}_2, a^{int}(e_3)\}$
R_3	$\{e_2\}$	$\{c_3\}$	$\{a^{svc}_3\}$
R_4	$\{e_3\}$	$\{\}$	$\{a^{svc}_4\}$
R_5	$\{e_3, e_6\}$	$\{c_1\}$	$\{a^{svc}_5\}$
R_6	$\{e_3, e_7\}$	$\{\}$	$\{a^{svc}_6\}$
R_7	$\{e_1\}$	$\{c_1 \wedge c_7\}$	$\{a^{svc}_7\}$

$\mathcal{AC}^{ws}$	*action constraints*
AC_4	$\{a^{svc}_5, a^{svc}_6, a^{svc}_7\}$

Figure 12. Example of static conflict detection

5.2.2 Dynamic Conflict Detection

Under the existence of composite events that are defined by using conjunctions and serializations, the exact event matching alone cannot guarantee the nonexistence of conflicts at run-time. Suppose that two rules respectively contain conjunctions of events, $e_1 \wedge e_2$ and $e_1 \wedge e_3$, and true conditions for the corresponding rules. Assuming that the resulting actions of two rules violate a service constraint, we cannot tell at design-time whether or not they will lead to a rule conflict because we do not know if two event e_2 and e_3 can occur simultaneously. The similar argument can be made for the case of conditions of the rules. We know that a co-satisfiable rule set has a situation in which all the rules can be satisfied. However, we do not know whether a compatible rule set may have such a situation or not at run-time. These kinds of conflicts are referred to as dynamic conflicts, which are dependent on the actual event occurrences and the evaluation of conditions at run-time. Dynamic conflict is formally defined as follows.

Definition 11 (Dynamic conflict). *Let $\mathcal{RS}^{trig}$ be a set of normalized rules. It is said that $\mathcal{RS}^{trig}$ has a dynamic conflict if it is a concurrable and compatible rule set and its rules have service actions against any service constraint in $\mathcal{AC}^{ws}$. Formally, $\mathcal{RS}^{trig}$ has a dynamic conflict if i) $\exists e^{final}$, such that $\forall R_i \in \mathcal{RS}^{trig}$, $\exists e^i (= e^i_1;...;e^i_n) \in E^{Ri}$, and $e^{final}=e^i_n$, ii) $\forall R_i \in \mathcal{RS}^{trig}$ $(1 \leq i \leq m)$, $\exists c^i \in C^{Ri}$, such that $c^1 \wedge ... \wedge c^m \neq false$, and finally iii) $\exists AC \in \mathcal{AC}^{ws}$, such that $AC \subset A^{RStrig}$, where $A^{RStrig} = \{ a^{svc} \mid a^{svc} \in A^{Ri}, R^i \in \mathcal{RS}^{trig}\}$*

The dynamic conflict means that it is possible for the rules in the set to result in some service conflict because i) the rules can be triggered at a time (*concurrable*), ii) be satisfied in a situation (*compatible*), and iii) cause a conflict against some service constraints. The concept of dynamic conflict has been proposed to improve the performance of rule conflict detection and resolution at run-time. If the checking of dynamic rule conflict is not preprocessed at design-time, it should be performed at run-time like in [14] and [40]. However, since it is quite time-consuming work to check if each rule causes those run-time conflicts, the strategy is not adequate to ubiquitous computing environments. Moreover, the concept of an epoch may result in some discrepancies in the environments where instant events and responses are required. In our mechanism, the potential dynamic conflicts are investigated at design-time, and only the marked rules, namely the rules with dynamic conflicts, are checked at run-time. The resolution mechanism of the dynamic conflict will be introduced in Section 5.3.

GRM identifies the possibility of dynamic conflicts induced by a new rule only after the rule is judged to have no static conflict. We propose a dynamic conflict detection algorithm in Figure 13. Indeed the algorithm is an extension from the static conflict detection algorithm.

Input: A new rule R_{new}, a normalized rule set $\mathcal{RS}^{ws}$, a service constraints set $\mathcal{AC}^{ws}$, a conditional action set with

preceding events $\mathcal{ECA}^{ws}$

Output: A dynamic conflict set RC^{dyn}, where RC^{dyn}:={ (e, AC, RS^{dyn}) | triggering event e, service constraint AC, and conflicting rule set RS^{dyn}}

Algorithm *dynamic_conflict_detection*(**in**(R_{new}, $\mathcal{RS}^{ws}$, $\mathcal{AC}^{ws}$), **out**(RC^{dyn}))

1: RC^{dyn}:={};
2: ecA^{Rnew} := *makeCondActs4DC*(R^{new}); // make a conditional action set of the new rule
3: **for** each $e \in E^{Rnew}$ **do**
4: e^{final} = *finalEvt*(e); e^{pre} =*precEvt*(e);
5: $ecA(e^{final}) := ecA(e^{final}) \cup ecA^{Rnew}$;
6: *checkConflict4DC*(e^{final}, $ecA(e^{final})$);
7: **end for**
8: **if** $RC^{dyn} = \phi$ **then** *updateCondActs4DC*(e^{final});
9: **else**
10: **for** $DC \in RC^{dyn}$ **do**
11: *createDCCR*(*DC*);
12: *markDC*(*RuleSet*(*DC*), *DC*);
13: **end for**
14: **end if**
15: **return** RC^{dyn};

Function *makeCondActs4DC*(R^{new})

ecA={};
for each $a \in A^{Rnew}$ **do**
if $a = a^{svc}(svc)$ **then** $ecA = ecA \cup \{(a, C^{Rnew}, e^{pre})\}$;
else if $a = a^{int}(e^{trig}) \,||\, a = a^{ext}(e^{trig})$ **then** $ecA = ecA \cup \{(a^t, C^t \wedge C^{Rnew}, E^t \wedge \{e^{pre}\} \mid (a^t, C^t, E^t) \in ecA(e^{trig})\}$;
end if
end for
return *cA;*

Function *checkConflict4DC*(e^{final}, ecA)

for each $AC \in \mathcal{AC}^{ws}$ **do**
if $AC \subset \{ a \mid (a, C, E) \in ecA\}$ **then**
$ecA^{conflict}$:={$(a, C, E) \in ecA \mid a \in AC$};
RS^{dyn}:={ $R \in \mathcal{RS}^{ws} \mid (a, C, E) \in ecA^{conflict}$ and $a \in A^R$};
if *checkCompatibility*($ecA^{conflict}$)=*true* **then** RC^{dyn}:=$RC^{dyn} \cup$ {(e, AC, RS^{dyn}, *PrecEvtSet*($ecA^{conflict}$), *CondSet*($ecA^{conflict}$))};
end if
end for
return RC^{dyn};

Figure 13. Algorithm *dynamic_conflict_detection*

The algorithm *dynamic_conflict_detection* starts with a new rule R_{new}, a normalized rules set $\mathcal{RS}^{ws}$, and a service constraint set $\mathcal{AC}^{ws}$, and a conditional action set $\mathcal{ECA}^{ws}$, which is also updated for every rule insertion, as an input. The output of the algorithm is dynamic rule conflict set RC^{dyn}. It has the same structure as in *static_conflict_detection*. However, since dynamic conflicts are dependent on the preceding events of final primitive events as well as the several conditions, the algorithm introduces $ecA(e^{final})$ which is the conditional

action set with preceding events for each final event e^{final}. Furthermore, $(a, C, E^{prec}) \in ecA(e^{final})$ denotes that, on the occurrence of e^{final}, service action a will be executed when the condition set C is satisfied and all the preceding actions in E^{prec} have already carried out.

Similar to the static conflict detection algorithm, the algorithm also checks dynamic rule conflict for each serialization event $e \in E^{Rnew}$ (Lines 3-8). Dynamic conflict can happen from the rules that have the events with the same final primitive event. Function *makeCondActs4DC*(R^{new}) of Line 2 is similar to function *makeCondActs*(R^{new}) in the algorithm *static_conflict_detection*. The only difference is that service actions in the set *ecA* accompany the preceding events set as well as their conditions set.

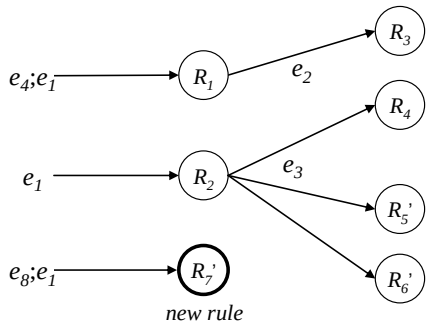
Next, we take a union of the conditional action set of e^{final}, $ecA(e^{final})$, and that of the rule, ecA^{Rnew}, and then check if the set has any service conflict through the function *checkConflict4DC*(e^{final}, *ecA*) (Lines 5-6). In case that the event e is the first event with the final event e^{final} in all rules, the algorithm makes a new conditional action set from ecA^{Rnew} and then check the service conflict.

Finally, if the new rule has no dynamic conflict, the algorithm updates the conditional action set of the final event, $ecA(e^{final})$ (Line 8). If there exists any dynamic conflict, the algorithm creates a dynamic conflict check rule (DCCR) for each dynamic conflict, and then marks the rules that are involved in the dynamic conflict (Lines 10-13). At run-time, DCCR is responsible for checking if the prescribed conditions are satisfied after the preceding events have occurred. If it turns out that the dynamic conflict is imminent, a corresponding resolution rule will be triggered.

The complexity of the algorithm *dynamic_conflict_detection* is also $O(n_e n_a m)$, where n_e, n_a, and m are the number of events, actions, and service constraints, respectively. In this case, the traversals in the function *checkConflicts4DC(e, ecA)* take the time $O(n_e n_a m)$ because the size of conditional action set *ecA* is lower than $n_e \cdot n_a$ and the size of service constraints set is m (i.e. $|ecA| < n_e n_a$ and $|\mathcal{AC}^{ws}| = m$).

Example 4. The proposed dynamic conflict detection algorithm is demonstrated through an example. Suppose that a new rule R_7' is introduced in the modified example of Figure 10 and $\mathcal{AC}^{ws}$ has only one constraint, as shown in Figure 14. Event specification $E^{R7'}$ has only one serialization event $e_8;e_1$, whose final primitive event e_1 also appears as a final event of other serializations $e_4;e_1$ and e_1 (i.e. $E^{R7'} = \{e_8;e_1\}$, and $e^{final} = e_1$). In this case, we use an existing conditional action set of e_1. We obtain $ecA(e_1) = ecA^{R1}(e_1) \cup ecA^{R2}(e_1)$, where $ecA^{R1}(e_1)=\{(a^{svc}{}_1,\{c_1 \wedge c_3\},\{e_4\}),\ (a^{svc}{}_3,\{c_1 \wedge c_3\},\{e_4\})\}$ and $ecA^{R2}(e_1)=\{(a^{svc}{}_2,\{\},\{\}),\ (a^{svc}{}_4,\{\},\{\}),\ (a^{svc}{}_5,\{c_1 \wedge c_5\},\{\}),\ (a^{svc}{}_6,\{c_6\},\{e_7\})\}$. And then we compute $ecA'(e_1) = \{(a^{svc}{}_1,\{c_1 \wedge c_3\},\{e_4\}),\ (a^{svc}{}_2,\{\},\{\}),\ (a^{svc}{}_3,\{c_1 \wedge c_3\},\{e_4\}),\ (a^{svc}{}_4,\{\},\{\}),\ (a^{svc}{}_5,\{c_1 \wedge c_5\},\{\}),\ (a^{svc}{}_6,\{c_6\},\{e_7\}),\ (a^{svc}{}_7,\{c_7\},\{e_8\})\}$ since $ecA(e_1) = ecA(e_1) \cup ecA^{R7}(e_1)$ and $ecA^{R7} = \{(a^{svc}{}_7,\{c_7\},\{e_8\})\}$. Finally, we get the dynamic conflict set RC^{dyn} from the result of function *checkConflict4DC*($ecA'(e_1)$): $RC^{dyn} = \{ (e_1, \{a^{svc}{}_5, a^{svc}{}_6, a^{svc}{}_7\}, \{R_{5'}, R_{6'}, R_{7'}\}, \{e_7 \wedge e_8\}, \{c_1 \wedge c_5 \wedge c_6 \wedge c_7\}) \}$, which means that R_7' has only one dynamic conflict. That is, if the preceding event $e_7 \wedge e_8$ has occurred before the final event e_1, and the condition $c_1 \wedge c_5 \wedge c_6 \wedge c_7$ is satisfied, the event e_1 will cause a

service conflict against the constraint $\{a^{svc}_5, a^{svc}_6, a^{svc}_7\}$ by the rule set $\{R_{5'}, R_{6'}, R_{7'}\}$.



$\mathcal{RS}^{trig}$	E^R	C^R	A^R
R_1	$\{e_4;e_1\}$	$\{c_1 \wedge c_3\}$	$\{a^{svc}_1, a^{int}(e_2)\}$
R_2	$\{e_1, e_5\}$	$\{\}$	$\{a^{svc}_2, a^{int}(e_3)\}$
R_3	$\{e_2\}$	$\{c_3\}$	$\{a^{svc}_3\}$
R_4	$\{e_3\}$	$\{\}$	$\{a^{svc}_4\}$
R_5'	$\{e_3, e_6\}$	$\{c_1 \wedge c_5\}$	$\{a^{svc}_5\}$
R_6'	$\{e_7;e_3\}$	$\{c_6\}$	$\{a^{svc}_6\}$
R_7'	$\{e_8;e_1\}$	$\{c_7\}$	$\{a^{svc}_7\}$

$\mathcal{AC}^{ws}$	*action constraints*
AC_4	$\{a^{svc}_5, a^{svc}_6, a^{svc}_7\}$

Figure 14. Example of dynamic conflict detection

5.3 Dynamic Rule Conflict Resolution

GRM identifies static conflicts and potential dynamic conflicts before new rules are registered in distributed devices, as illustrated in Figure 9. When a rule has only dynamic conflict without any static conflict, it can be registered to a device after adding dynamic conflict resolution rules that will resolve the dynamic conflicts at run-time. Such a rule is registered with a mark indicating the possibility of a dynamic conflict, and it is the responsibility of GRM to check if the dynamic conflict actually can happen before activating the service actions of the rule. The procedures defined for managing dynamic conflicts are illustrated in Figure 15.

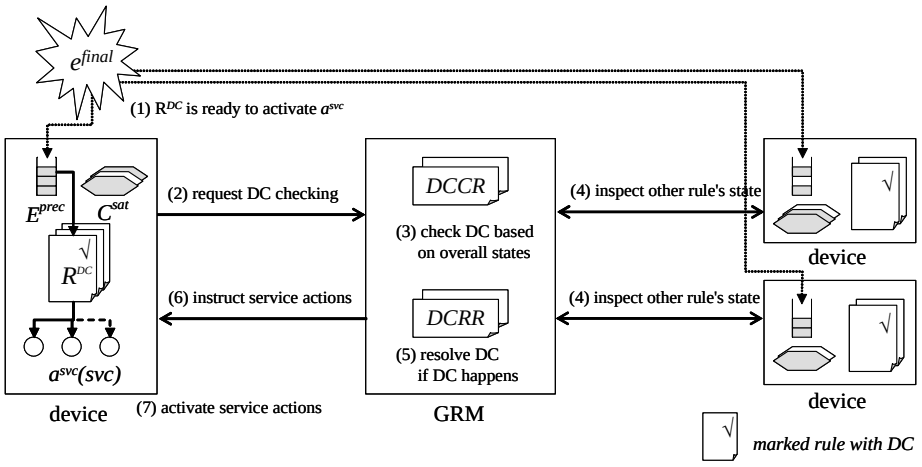


Figure 15. Procedure for dynamic conflict management

When a rule marked with a dynamic conflict (DC) is ready to activate its actions after its event specification and conditions are satisfied, the device requests DC checking to GRM as shown in Figure 15. For this purpose,

GRM inspects the status of the other devices that have registered the marked rules of the DC. Later, the other devices with the marked rules report their status (i.e., events history and conditions). GRM then determines the possibility of occurrence of the DC based on the overall status, and informs the devices of the result, which includes the instructions for service actions in case that the DC is going to happen. The devices finally activate their service actions according to the instruction.

The structures for the dynamic conflict check rule (DCCR) and the dynamic conflict resolution rule (DCRR) are shown in Figure 16. Note that both rules are also WS-ECA rules. The event specification of DCCR includes the event for checking DC (e^{DC}) and the preceding-event-set (E^{Prec}). The e^{DC} is an external event that will be sent from the device to GRM. When GRM receives e^{DC}, it inspects the preceding events in E^{Prec} from all devices that are associated with the marked rule.

DCCR
on compositeEvent(*final-event* (e^{final}) after *preceding events* (E^{prec}))
if *conflicting-conditions* (C^{DC})
do *dynamic-conflict-detection* (DC^{det})
DCRR
on *dynamic-conflict-detection* (DC^{det})
if *activation-conditions* (C^{rsv})
do *activation-actions* (A^{rsv})

Figure 16. Schema of DCCR and DCRR

Finally, Figure 17 shows examples of DCCR and DCRR for detecting and resolving a dynamic conflict of Example 4. The dynamic conflict of the example was RC^{dyn} = { (e_1, {$a^{svc}{}_5$, $a^{svc}{}_6$, $a^{svc}{}_7$}, {$R_{5'}$, $R_{6'}$, $R_{7'}$}, {$e_7 \wedge e_8$}, {$c_1 \wedge c_5 \wedge c_6 \wedge c_7$}) }. DCCR is generated automatically when the marked rule is registered to the device, while DCRR should be defined by an administrator. Once a dynamic conflict is detected, GRM will generate internal event RC^{dyn}. Two DCRRs presented in Figure 17 are handling the conflict based on the current temperature in this example.

$DCCR_1$
on **compositeEvent**(e_1 after $e_6 \wedge e_8$)
if $c_1 \wedge c_5 \wedge c_6 \wedge c_7$
do **createIntEvent**(DC_1)
$DCRR_1$

on **intEvent**(DC_1) ***if*** $c_6.temp \geq c_7.temp$ ***do*** **compositeAction**(**invokeService**(a^{svc}_5) and **invokeService**(a^{svc}_6))
$DCRR_2$
on **intEvent**(DC_1) ***if*** $c_6.temp < c_7.temp$ ***do*** **compositeAction**(**invokeService**(a^{svc}_5) and **invokeService**(a^{svc}_7))

Figure 17. DCCR and DCRR for the dynamic conflict of Example 4

6. Conclusions and Future Work

This paper presented an event-based rule description language, named WS-ECA, for effective coordination of web services-enabled devices in ubiquitous computing environment. WS-ECA enables users to describe required interactions among the service devices in a system where multiple devices exchange their events and interact with each other based on WS-Eventing and web service invocations. While existing web service based process execution languages such as WS-BPEL and WS-CDL are specifically proposed for supporting long-running, transactional business processes, the proposed WS-ECA attempts to support instantaneous, reactive actions of web services-enabled devices upon a WS-Eventing message through providing means for stateless, event-based interactions.

WS-ECA rules for individual devices may have discrepancies with each other and cause undesirable situations when they are executed concurrently, since they are created and processed independently. To address this problem, we proposed a framework for conflict detection and resolution of distributed WS-ECA rules. The conflicts of WS-ECA rules defined for distributed devices were categorized into static and dynamic conflicts depending on whether the conflict is resolved at design-time or run-time. When a rule is evaluated to contain logical contradiction with other rules at design-time, it is said to be in a static conflict, and it cannot be registered to the system and must be modified by a user.

On the other hand, if it is judged to have any potential dynamic conflict with others at run-time, additional rules that are responsible for handling the conflict by instructing some prescribed actions to the corresponding devices need to be supplemented in case that the conflict can actually happen at run-time. In this paper, we proposed necessary concepts and formal characterizations of WS-ECA rules, and presented algorithms for static conflict detection as well as dynamic conflict detection and resolution. The presented framework for event-driven coordination of distributed web service devices is expected to contribute to the efficient implementation of emerging ubiquitous service-based systems.

Future work includes access control mechanism and resource management for the purpose of effective rule management in ubiquitous service environments with multiple users. For instance, authority management with

ACL (Access Control Level) can be employed to resolve service conflicts in multi-user environments. Moreover, advanced service monitoring and administration are also required for robust rule management which includes the issues such as detection and resolution of deadlock and livelock arising in ubiquitous service network supporting active rule processing.

References

[1] T. Andrews *et al*., Business Process Execution Language for Web Services: Version 1.1, OASIS, 2003. http://www-128.ibm.com/developerworks/library/specification/ws-bpel/

[2] R.J. Auburn, J. Barnett, M. Bodell, and T.V. Raman, State Chart XML (SCXML): State Machine Notation for Control Abstraction 1.0, W3C Working Draft, 2005.

[3] D. Bank *et al.*, Web Services Eventing, W3C Member Submission, 2006. http://www.w3.org/Submission/WS-Eventing/

[4] N. Bassiliades, and I. Vlahavas. DEVICE: Compiling production rules into event-driven rules using complex events. Information and Software Technology, 39(5): 331-342, 1997.

[5] J. Bemmel, P. Dockhorn, and I. Widya. Paradigm: event-driven computing. Lucent Technologies, white paper, 2004. https://doc.telin.nl/dscgi/ds.py/

[6] M. Berndtsson, S. Chakravarthy, and B. Lings. Extending database support for coordination among agents. Int'l Journal of Cooperative Information Systems, 6(3-4): 315-340, 1997.

[7] G. von Bulltzingsloewen, A. Koschel, P.C. Lockemann, and H.-D. Walter. ECA functionality in a distributed environment. In N.W. Paton (editor), Active Rules in Database Systems, Springer, 147-175, 1999.

[8] M. Gudgin, M. Hadley, T. Rogers. Web Services Addressing 1.0- Core. W3C Recomendation, 2006. http://www.w3.org/TR/ws-addr-core/

[9] C. Bussler, and S.Jablonski. Implementing agent coordination for workflow management systems using active database systems. In Proc. of the Int'l Workshop on Research Issues in Data Engineering (RIDE), pages 53-59, 1994.

[10] S. Calo, and M. Sloman, Policy-based management of networks and services. Journal of Network and Systems Management. 11(3): 249-252, 2003.

[11] A. Carter, and M. Vukovic. A framework for ubiquitous web service discovery. In Proc. of the 6th UbiComp, 2004. http://ubicomp.org/ubicomp2004/adjunct/posters/carter.pdf

[12] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: an object oriented DBMS with event-based rules. Information and Software Technology, 36(9): 555-568, 1994.

[13] K. M. Chandy. Event-driven applications: costs, benefits and design approaches, Gartner Application Integration and Web Services Summit 2006, 2006. http://www.infospheres.caltech.edu/papers/Gartner_20060620.pdf

[14] J. Chomicki, J. Lobo, and S. Naqvi. Conflict resolution using logic programming. IEEE Trans. Knowl. Data Eng., 15(1): 244-249, 2003.

[15] M. Cilia and A. Buchmann. An active functionality service for e-business applications. ACM SIGMOD Record, 31(1): 24-30, 2002.

[16] J. Clark and S. DeRose, XML Path Language (XPath) Version 1.0, W3C Recommendation, 1999, http://www.w3.org/TR/xpath

[17] S. Comai, and L. Tanca. Termination and confluence by rule prioritization. IEEE Trans. Knowl. Data Eng., 15(2): 257-270, 2003.

[18] U. Dayal, B. Blaustein, A.P. Buchmann, S. Chakravarthy, D. Goldhirsch, M. Hsu, R. Ladin, D. McCarthy, and A. Rosenthal. The HiPAC project: combining active databases and timing constraints. ACM SIGMOD Record, 17(1): 51–70, 1998.

[19] O. Diaz, and A. Jaime. EXACT: an extensible approach to active object-oriented databases. VLDB Journal, 6(4): 282-295, 1997.

[20] F.M. Facca, S. Ceri, J. Armani, and V. Demalde. Building reactive web applications. In Proc. of the 14th Int'l World Wide Web Conf. (WWW 2005), pages 1058-1059, 2005.

[21] A. Friday, N. Davies, N. Wallbank, E. Catterall, and S. Pink. Supporting service discovery, querying and interaction in ubiquitous computing environments. Wireless Networks 10(6): 631–641, 2004.

[22] S. Gatziu, K.R. Dittrich. SAMOS: an active object-oriented database system. IEEE Data Eng. Bull., 15(1-4): 23-26, 1992.

[23] H. Gellersen, G. Kortuem, A. Schmidt, and M. Beigl. Physical prototyping with Smart-Its. IEEE Pervasive Computing, 3(3): 74-82, 2004.

[24] J. Hanson. Event-driven services in SOA: Design an event-driven and service-oriented platform with Mule. JavaWorld. 2005 http://www.javaworld.com/javaworld/jw-01-2005/jw-0131-soa.html

[25] J.-Y. Jung, S.-K. Han, J. Park, and K. Lee. WS-ECA: an ECA rule description language for ubiquitous services computing. In Proc. of the Workshop on Empowering the Mobile Web (MobEA IV) in conjunction with WWW 2006, 2006. http://www.research.att.com/~rjana/MobEA-IV/PAPERS/MobEA_IV-Paper_10.pdf

[26] J.M. Kahn, R.H. Katz, and K.S.J. Pister. Mobile networking for Smart Dust. In Proc. of ACM/IEEE Int'l Conf. on Mobile Computing and Networking (MobiCom 99), pages 17-19, 1999.

[27] V. Kantere, and A. Tsois. Using ECA rules to implement mobile query agents for fast-evolving pure P2P networks. In Proc. of the 3rd Int'l Joint Conf. on Autonomous Agents and Multiagent Systems, pages 1510-1511, 2004.

[28] N. Kavantzas, et al., Web services choreography description language Version 1.0, W3C Candidate Recommendation, 2005. http://www.w3.org/TR/ws-cdl-10/

[29] K. Liu, L. Sun, A. Dix, and M. Narasipuram. Norm based agency for designing collaborative information systems. Information Systems Journal, 11(3): 229-247, 2001.

[30] J. Lobo, R. Bhatia, and S. Nagvi. A policy description language. In Proc. of National Conference on the American Association for Artificial Intelligence, pagers 291-298, 1999.

[31] T.S. Lopez, D. Kim, T. Park. A service framework for mobile ubiquitous sensor networks and RFID. In Proc. of the 1st Int'l Symp. on Wireless Pervasive Computing, 2006

[32] Z. Maamar. On coordinating personalized composite web services. Information and Software Technology, 48(7): 540-548, 2006.

[33] B. Michelson. Event-Driven Architecture Overview: Event-Driven SOA is Just Part of the EDA story. White paper, Patricia Seybold Group, 2006. http://www.psgroup.com/detail.aspx?ID=681

[34] Microsoft, The Microsoft invisible computing project web site. http://research.microsoft.com/invisible/

[35] MIT Project Oxygen web site. http://oxygen.lcs.mit.edu/

[36] OMA: OMA Web Services Enabler (OWSER): Overview. OMA-AD-OWSER Overview-V1 1-20060328-A (2006). http://www.openmobilealliance.org/release program/owser v1 1.html

[37] N.W. Paton, and O. Díaz. Active database systems. ACM Computing Surveys, 31(1): 63-103, 1999.

[38] C. Peltz. Web services orchestration and choreography. IEEE Computer 36(10): 46-52, 2003.

[39] A. Sashima, N. Izumi, and K. Kurumatani. Location-mediated coordination of web services in ubiquitous computing. In Proc. of IEEE Int'l Conf. Web Services (ICWS'04), pages 109-114, 2004.

[40] C.S. Shankar, A. Ranganathan, and R. Campbell. An ECA-P policy-based framework for managing ubiquitous computing environments. In Proc. of the 2nd Int'l Conf. on Mobile and Ubiq. Sys., pages 33-42, 2005.

[41] UPnP, The UPnP forum web site, http://www.upnp.org

[42] S. Vinoski. Integration with web services. IEEE internet computing, 7(6): 75-77, 2003.

[43] I. Vlahavas, and N. Bassiliades. Parallel, object-oriented, and active knowledge base systems. Advances in Database Systems, Kluwer Academic Publishers. 1998.

[44] M. Weiser. The computer for 21st century.Scientific American, 265: 94-104, 1991.

[45] Y. Yokohata, Y. Yamato, M. Takemoto, and H. Sunaga. Service Composition Architecture for Programmability and Flexibility in Ubiquitous Communication Networks. In Proc. of Int'l Symp. on Applications and the Internet Workshops (SAINTW'06), pages 142-145, 2005.

Appendix. WS-ECA rules for the morning cook service example

A. alarm-clock.xml

```
<ECARule name="alarm-clock-rules"
     targetNampespace="http://di.snu.ac.kr/alarm-clock/rules/"
     xmlns:al="http://di.snu.ac.kr/alarm-clock/type/"
     xmlns="http://di.snu.ac.kr/2005/eca/">
<events>
     <timeEvent type="periodic" name="get-up-time" unit="P1D">
         0000-00-00T07:00:00Z </timeEvent>
     <timeEvent type="relative" name="before-get-up-time"
         baseEvent="get-up-time" interval="-PT20M"/>
</events>
<actions>
     <createExtEvent name="pre-alarm"
         extEvent="http://di.snu.ac.kr/event/alarm-clock/alarm">
         <al:contents>'20min before get-up'</al:contents>
     </createExtEvent>
     <createExtEvent name="get-up-alarm"
         extEvent="http://di.snu.ac.kr/event/alarm-clock/alarm">
         <al:contents>'get-up'</al:contents>
     </createExtEvent>
</actions>
<rules>
     <rule name="alarm-before-get-up-rule">
         <event name="before-get-up-time"/>
         <condition expression="/alarm/today/@holiday='no'"/>
         <action name="pre-alarm"/>
     </rule>
     <rule name="alarm-on-get-up-rule">
         <event name="get-up-time"/>
         <condition expression="/alarm/today/@holiday='no'"/>
         <action name="get-up-alarm"/>
     </rule>
</rules>
</ECARule>
```

B. rice-cooker.xml

```
<ECARule name="rice-cooker-rules"
     targetNampespace="http://di.snu.ac.kr/rice-cooker/rules/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:rc="http://di.snu.ac.kr/rice-cooker/WSDL/"
     xmlns="http://di.snu.ac.kr/2005/eca/">
<variables>
     <variable name="hasEnoughRice" deviceVar="rc:hasEnoughRice"/>
</variables>
<events>
     <intEvent name="cooking"/>
     <extEvent name="alarm"
```

```
        eventID="http://di.snu.ac.kr/event/alarm-clock/alarm"/>
    <svcEvent type="before" name="before-cooking" service="rc:cook"/>
    <svcEvent type="after" name="after-cooking" service="rc:cook"/>
    <compositeEvent type="SEQ" name="alarm-after-out-of-rice" TTL="PT1H">
        <event name="cooking"/><event name="alarm"/>
    </compositeEvent>
</events>
<actions>
    <createIntEvent name="start-cooking" intEvent="cooking"/>
    <createIntEvent name="detect-out-of-rice" intEvent="out-of-rice"/>
    <createExtEvent   name="complete-cooking"   extEvent="http://di.snu.ac.kr/event/rice-
cooker/cooking-completion"/>
    <invoke name="invoke-cooking" service="rc:cook"/>
    <invoke name="alert-out-of-rice" service="rc:alert">
        <rc:contents>out of rice</rc:contents>
    </invoke>
</actions>
<rules>
    <rule name="cooking-rule">
        <event name="alarm"/>
        <condition     expression="(/alarm/contents='20min     before     get-up')     |
(/alarm/contents='30min before dinner')"/>
        <action name="start-cooking"/>
    </rule>
    <rule name="enough-rice-rule">
        <event name="cooking"/>
        <condition expression="hasEnoughRice=true"/>
        <action name="invoke-cooking"/>
    </rule>
    <rule name="not-enough-rice-rule">
        <event name="cooking"/>
        <condition expression="hasEnoughRice=false"/>
        <action name="detect-out-of-rice"/>
    </rule>
    <rule name="out-of-rice-alarm-rule">
        <event name="alarm-after-out-of-rice"/>
        <condition expression="/alarm/contents='get-up'"/>
        <action name="alert-out-of-rice"/>
    </rule>
    <rule name="cooking-completion-rule">
        <event name="after-cooking"/>
        <condition expression="true"/>
        <action name="cooking-completion"/>
    </rule>
</rules>
</ECARule>
```

C. coffee-maker.xml

```
<ECARule name="coffee-maker-rules"
    targetNampespace="http://di.snu.ac.kr/coffee-maker/rules/"
    xmlns:cm="http://di.snu.ac.kr/coffee-maker/WSDL/"
    xmlns="http://di.snu.ac.kr/2005/eca/">
<events>
    <extEvent name="cooking-completion"
        eventID="http://di.snu.ac.kr/event/rice-cooker/cooking-completion"/>
    <extEvent name="alarm"
        eventID="http://di.snu.ac.kr/event/alarm-clock/alarm"/>
    <compositeEvent type="AND" name="cooking-and-get-up" TTL="PT30M">
        <event name="cooking-completion"/><event name="alarm"/>
    </compositeEvent>
```

```
    <timeEvent type="relative" name="coffee-making-time"
        baseEvent="cooking-and-get-up" interval="PT10M"/>
</events>
<actions>
    <invoke name="make-coffee" service="cm:makeCoffee"/>
</actions>
<rules>
    <rule name="coffee-making-rule">
        <event name="coffee-making-time"/>
        <condition expression="/alarm/contents='get-up'"/>
        <action name="make-coffee"/>
    </rule>
</rules>
</ECARule>
```