1000-9825/2002/13(08)1402-10 ©2002 Journal of Software 软 件 学 报 Vol.13, No.8

Analysis of Composite Events in Adaptive Mobile Computing System by Unidirectional Queue Automata[*]

LI Guo-dong, ZHANG De-fu

(*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210093, *China*)
E-mail: dong@dislab.nju.edu.cn
http://www.nju.edu.cn

Received August 29, 2001; accepted December 18, 2001

Abstract: Mobile computing attracts more and more attention in recent years. Adaptive mobile computing systems need to process a broad range of composite events, which are combinations of primitive events such as logic event, time events, and temporal events, etc. This paper focuses on the design and implementation of composite events and composite actions in mobile computing systems, the sophisticated cases of event stream, time event and various operators are taken into consideration. Detection for composite events is supported efficiently by new kinds of extended automaton—Unidirectional Queue Automata and Bounded Unidirectional Queue Automata, which are defined and discussed in detail. The data structures and the mechanisms adopted by these new automata are described and how they can be efficiently applicable to adaptive mobile environments is illustrated.

Key words: mobile computing; adaptive; composite event; automata; unidirectional queue

Mobile devices and wireless networks are becoming more powerful and affordable, leading to the growing importance of mobile data access. However, factors such as weak network connectivity, energy constraints, and mobility itself raise new concerns regarding the security, reliability, and even correctness of a mobile computing system. Especially, mobile environments are inherently turbulent; the resources available to mobile clients change dramatically and unpredictably over time. In such an environment of changing resource constraints, a system with no in-built adaptiveness would make wrong decisions on resource allocation. Although some of these limitations are induced by current technological shortcomings, many of them are inherent to the nature of mobile computing, and can be expected to continue beyond future advances in technology.

Mobile computing applications must be made aware of mobility not only to better utilize constrained resources, but also to provide enhanced mobility related functionality. Unfortunately, the wide variety of environmental situations that mobile computing presents make it difficult to build an application that optimally handles all situations.

An efficient adaptation scheme is to let operating system and application cooperate with each other; the system is responsible for providing the mechanisms for adaptation, while applications are free to set adaptive policies. Environmental change can be modeled as asynchronous event, which is in charge of hiding and exposing mobility awareness. Thus basic application functionality in this architecture is cleanly decoupled from adaptiveness,

∗ Supported by the National High Technology Development 863 Program of China under Grant No.863-306-ZT06-04-4 (国家 863 高科技发展计划)

LI Guo-dong was born in 1975. He is an MS candidate at the Department of Computer Science and Technology, Nanjing University. His research interests are parallel and distributed computing. **ZHANG De-fu** was born in 1937. He is a professor and doctoral supervisor of Nanjing University. His current research areas include parallel processing and distributed systems.

allowing the application to evolve independently of any particular environmental situation. In this paper we introduce the concepts of composite events and application actions, which provide more flexibility for handling events and actions. We concentrate on using an extended automation - Unidirectional Queue Automata - to describe the behaviors of composite events and actions.

In this paper, Section 1 introduces background containing a brief description about adaptation. Section 2 addresses the concepts of events and composite events. Section 3 describes the unidirectional queue automata (UQA). Section 4 uses UQA to analyze composite events. In Section 5 related works and extensions are discussed. Finally, we give the conclusion.

1 Adaptation Mobile Computing System

1.1 Adaptation in mobile computing system

There are two key constraints on mobile devices in comparison to their fixed counterparts: resource-poor and less secure. The variations in demand for resources, together with the variation in the already scarce supply with those resources, lead to the requirement that mobile hosts adapt to these changes[1].

The general approach for application adaptiveness in response to the mobile environment ranges from attempting to minimize application awareness, to providing various API's for implementing adaptive behavior[1]. Some approaches utilize semantic knowledge of data in the application domain to predict access patterns and intelligently hoard data[2]. Unfortunately, each of these approaches either eliminates application environment awareness, or completely integrates the required adaptiveness into application functionality.

One of essential problems is about what party is responsible for making adaptation decisions: the operating system, the applications, or some combination of the two. There are three models of adaptation[1]: (1) Application-transparent adaptation, in which the system is wholly responsible for adapting to changes in the supply of and demand for resources. (2) Laissez-faire adaptation, in which applications are solely responsible for coping with the consequences of mobility. (3) Application-aware adaptation, in which the system is responsible for monitoring resource availability and individual application must be informed of and react to significant changes in the availability of resources. The third approach (application-aware adaptation) is preferable to its counterparts. Systems must support a diverse range of applications, with potentially different adaptive needs; they must also support concurrent, competing applications. These two requirements lead to providing adaptation as a collaborative approach between the system, which monitors and controls resources, and applications, which set adaptation policy. In the next we introduce our application-aware adaptation system model.

1.2 Adaptation in mobile computing system

Although current operating systems are capable of recognizing and adapting to a dynamic resource environment, the abstractions for informing an application of the induced changes are inadequate for mobile computing. Furthermore, many conditions in a mobile computing environment can only be determined in user-level daemons that form part of the mobile computing run-time[3]. In order to address these concerns, we propose a new approach to make an application aware of environmental changes. The architecture is based on an asynchronous event delivery mechanism over which typed events can be delivered to mobile computing applications. Event types can be arbitrarily extended, and the architecture itself can be configured and extended according to the needs of a particular system. Moreover, an application can choose to handle events at the level of abstraction that it deems appropriate. However, details about the system are neglected here because our paper focuses on the issue of composite events and actions.

2 Composite Events and Actions

Many applications require access to two or more event sources, an application may receive several events from the same user within a time period, thus a generic set of composite event services are needed.

2.1 Basic event operators

We now describe basic composite event operators, if a is an event, t_a denotes the time at which event a occurred.

Follow-by: It expresses temporal relations and is the most fundamental and important operator. To $eg1=a()\text{->} b()$, $eg1$ will be accepted if event a and event b occur where $t_b>t_a$. Other classes of event are not considered and play no part in the evaluation.

Disjunction: It can alternatively be termed exclusive or and is denoted by a |. To $eg2=a()|b()$,$eg2$ will be accepted if event a or event b occurs. The result of an accepting evaluation will be the event occurring first.

Conjunction: It can also be termed "and" and is denoted by a^. To $eg3 = a() \wedge b()$, $eg3$ will be accepted when both events a and b have occurred, a may occur before b or vice versa. Conjunction can alternatively be expressed using disjunction and follow-by: $eg3 =a()\wedge b()= (a()\text{->}b())|(b()\text{->}a())$.

Without : It is denoted by -. To $eg4=a()\text{->}b()–c()$, $eg4$ will be accepted if a is seen followed-by b with no intervening of c, that is when $t_a<t_b<t_c$. If event c is seen between the occurrence of a and b the evaluation is said to reject.

Closure: It is denoted with *. For example, $eg5 = a^*$ accepts whenever event a occurs at least once.

The operator precedence of operators are: *, | and ^, -, ->, highest precedence shown first and equal precedence are evaluated from left to right. Parentheses can be used to overcome operator precedence.

Each event has a set of attributes. A system should include facilities to enable an application to gain access to these attributes. The solution is to allow the use of variables with the expressions. The use of variables offers a useful way of correlating between different events, reducing the number of expressions that are triggered. For example, $Eg5=a(X)\text{->}b(Y) \mid c(Y)– d(X)$ means a followed-by b or c and without an intervening d.

All the event classes used in the expression take variables as parameters. At the beginning of evaluation all variables are uninstantiated. When event a occurs, the variable value X is instantiated with the corresponding attribute. This attribute value is used in the monitoring of event d. Concurrently event classes b and c are monitored.

2.2 Event type and event stream

Event type is used to differentiate these events from other events from different sources.

Definition 1. All events coming from the same source belong to the same event type. Events with same type identify themselves with variables taken as parameters.

Events $a(1)$, $a(2)$, $a(3)$ belong to event type a, for brief purpose, we denote them as $a1$, $a2$, $a3$. Consider the following event stream, where each succeeding event occurs at a later time than its written predecessor.

$$a1, a2, b1, a3, b2, a4, b3$$

Two a events, $a1$, $a2$, are seen followed-by a b event. In the most general case, every possible event combination is created, so ($a1$,$b1$), ($a2$,$b1$), ($a1$,$b2$), ($a2$,$b2$), ($a3$,$b2$), ($a1$,$b3$), ($a2$,$b3$), ($a3$,$b3$), ($a4$,$b3$) are produced. The result is a large number of accepting evaluations, to reduce the number, the recent event context can overwrite an event with a less recent one of the same class, in which the example produces ($a2$,$b1$), ($a3$,$b2$), ($a4$,$b3$). The operator ->^1 use this kind of scheme to reduce the total number of evaluations. On the other side, if the recent event context cannot overwrite less recent event, in which the example produces ($a1$,$b1$), ($a3$,$b2$), ($a4$,$b3$), new follow-by operator ->^2 is used to refer to this case. However, to add flexibility, two other follow-by operators are introduced, denoted by $->^3$ and $->^4$, which allows multiple instances of the same expression to be evaluated concurrently.

Consider the two new introduced operators $->^3$ and $->^4$. $a()->^3 b()$ results in ($a1$,$b1$), ($a2$,$b1$), ($a3$,$b2$), ($a4$,$b3$); $a()->^4 b()$ results in ($a1$,$b1$), ($a1$,$b2$), ($a1$,$b3$), ($a2$,$b1$), ($a2$,$b2$), ($a2$,$b3$), ($a3$,$b2$), ($a3$,$b3$), ($a3$,$b4$), ($a4$,$b3$).

3 Unidirectional Queued Automata

We are now ready to discuss the question of what kind of automaton is needed for accepting composite event languages. We get hints from the cases of Pushdown Automata and Context Free Languages[4] and begin by trying to construct an appropriate automaton for recognizing composite event languages. A new data structure "circle queue" is introduced as an auxiliary storage device.

3.1 Unidirectional circle queue

Unidirectional circle queue (Fig.1(a)) behaves in a first-in-first-out (FIFO) manner. Initially head pointer and tail pointer are pointing to the same slot in queue, which means that queue is empty. Before new element is inserted, tail pointer moves to next slot while head pointer keeps intact. After one element pointed by head pointer is deleted, head pointer move to next slot and point to next element to be deleted in the next time. The number of slots in a queue is called the size of the queue and can be denoted as, e.g., S_q.

Using two pointers can perform queue operations such as emptying the queue, being aware of whether the queue is empty or full, adding or deleting element and so on. We replace head pointer and tail pointer with special symbol $ and # respectively. Initially when the queue is empty, $ and # are adjacent (Fig.1(b)). Then symbol *a* and symbol *b* are sequentially inserted into the queue (Fig.1(c)). When new element is inserted, # is replaced by the new element and a new # is placed in next slot, for instance, a new symbol *c* is inserted into the tail of the queue (Fig.1(d)). When one element is to be deleted, the $ is deleted and then a new $ replace the element to be deleted, for example, symbol a at the head of the queue is deleted (Fig.1(e)).

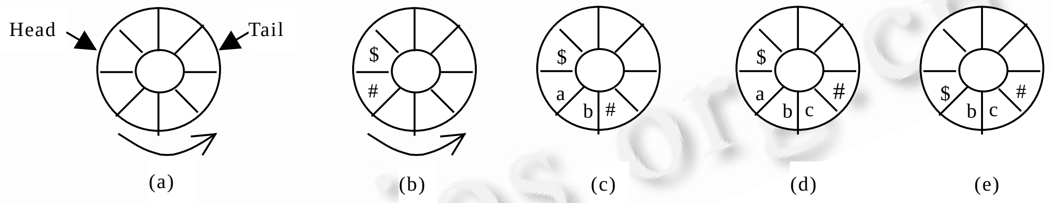

Fig.1 Circle queue

One interesting problem is how to know that the circle queue is empty or full. A simple answer resides on analyzing the relation of $ and # on the queue: when $ and # is adjacent, queue is empty or full, particularly, content "$#" means queue is empty and content "#$" means queue is full.

3.2 Unidirectional queued automata (UQA)

Queued Automata can only add symbol into the head of queue (via head pointer) and delete symbol from the tail of the queue (via tail pointer), adding in the head or deleting in the tail is prohibited. The size of queue can be limited or unlimited, as mentioned previously we use S_q to refer to the size of the queue, $S_q \in N$. Inspirited by the process of constructing pushdown automata[4], we now give formal definition of queued automata.

Definition 2. Unidirectional queued automata (UQA) is a seven tuple (Q, Σ, Γ, Δ, δ, q_0, F_a, F_r), where (Q, Σ, Γ, Δ, F_a and F_r are all finite set, and

1) Q is the states set.

2) Σ is the input alphabet.

3) Γ is the queue alphabet.

4) Δ is the output alphabet. Symbol_$\in\Delta$ is applicable to the case without output.

5) $\delta:Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon\times\Gamma_\varepsilon\to P(Q\times\Gamma_\varepsilon\times\Gamma_\varepsilon\times\Delta)$ is the transition function. Here $\Sigma_\varepsilon=\Sigma\cup\{\varepsilon\},\Gamma_\varepsilon=\Gamma\cup\{\varepsilon\}$, because current state, next input symbol as well as current queue symbols (including head symbol and tail symbol) determine the next action of the queue automata, permitting these two symbols to be $\in$ can enable the automata perform action without reading input symbol or accessing queue symbol. Transition function represents nondeterministics by returning a subset of $Q\times\Gamma_\varepsilon\times\Gamma_\varepsilon\times\Delta$, that is $P(Q\times\Gamma_\varepsilon\times\Gamma_\varepsilon\times\Delta)$.

6) $q_0\in Q$ is the initial state.

7) $F_a\subseteq Q$ is the set of accepting states.

8) $F_r\subseteq Q$ is the set of rejecting states. $F_a\cap F_r=\varnothing$.

Transition function utilizes both head symbol marked by \$ and tail symbol marked by # to determine next action. For example, for $\delta(q_1,a,h,t)=(q_2,h,\text{a},x)$ where $q_1,q_2\in Q$, $a\in\Sigma_\varepsilon$ is the input symbol, $h,t\in\Gamma_\varepsilon$ is the head symbol and tail symbol respectively, $x\in\Delta$ is the output symbol. δ insert symbol a into the tail of queue, which changes its content from "\$$h\ldots t$#" to "\$$h\ldots ta$#", and UQA reach state q_2 as well as output symbol x. Another example $\delta(q_1,a,h,t)=(q_1,\varepsilon,t,\_)$ deletes symbol h from the head of queue and changes its content from "\$$h\ldots t$#" to "\$…t#" while stay in state q_1 and doesn't output any symbol (marked by output symbol "_"). For convenient purpose, transition function can be described by the change of queue's content, $\delta(q_1, a, h, t)=(q_2, h, a, x)$ can be described by $\delta(q_1, a,$ "\$$h\ldots t$#) = ($q_2$, "\$$h\ldots ta$#", x) and $\delta(q_1, a, h, t)=(q_1,\varepsilon, t,\_)$ by $\delta(q_1$, "\$$h\ldots t$#", t) = (q_1, "\$…t#", _).

The computation of a $UQA=(Q,\Sigma,\Gamma,\Delta,\delta,q_0,F_a,F_r)$ is as following. It accepts w if w can be written as $w=w_1w_2w_3\ldots w_n$ (here each $w_i\in\Sigma_\varepsilon$) and state queue $r_0, r_1, \ldots, r_n\in Q$ as well as string queue $s_0, s_1, \ldots, s_n\in\Gamma_\varepsilon^*$ satisfying the following three conditions exist. String $s_0, s_1, .., s_n$ is the queue of contents in the circle queue while computing the accepting branch.

1) $R_0=q_0$ and $s_0=\$\#$, it means that UQA begin with empty circle queue.

2) To $i=0,\ldots,n-1$, (r_{i+1},b), where $s_i=\$at\#$, $s_{i+1}=\$atb\#$ or $s_{i+1}=\$t\#$, $a,b\in\Gamma_\varepsilon$ and $t\in\Gamma^*$. It means that at each step M add new symbol or delete existing symbol according to current state, next input symbol and head symbol or tail symbol of the circle queue.

3) $r_n\in F_a$, it means that an accepting state emerges after the input is over.

Now we discuss some properties of UQA.

Theorem 1. For any regular language L, there is a UQA recognizing L

Proof. For any regular language L, there is a deterministic finite automaton (DFA) M recognizing it[4], construct a UQA M' which is just the same as M except having an empty queue. To a string $w\in L$, M' accepts w if and only if M accept w. It is easy to prove that M' recognizing L, we neglect the detail here.

Theorem 2. If L and L' are languages accepted by a UQA, then so is $L\cup L'$.

Proof. Without loss of generality, let M, M' are BQA which accept L and L' respectively, with Q, δ, q_0, F_a, F_r and Q', δ', q_0', F_a', F_r' the set of states, transition function, initial state, set of accepting states and rejecting states of M and M', respectively. We also assume that M and M' have no states in common, i.e. $Q\cap Q'=\varnothing$. Furthermore, we can assume that the input alphabets, output alphabets and queue alphabets of L and L' are the same, say A, B, C respectively. We define a new UQA M'' with state Q'', initial state Q_0'', set of accepting states F_a'', set of rejecting states F_r'', and transition function δ'' as follows:

$$Q''=Q\cup Q'\cup\{q_0''\}-\{q_0,q_0'\}$$

$$F_a''=\begin{cases} F_a\cup F_a'\cup\{q_0''\}-\{q_0,q_0'\} & \text{if } q_0\in F_a \text{ or } q_0\in F_a' \\ F_a\cup F_a' & \text{otherwise}\end{cases}$$

$F_r'' = F_r \cup F_r'$

The transition function of M'' is defined as follows for $s \in A$, h, $l \in C$:

$$\delta''(q, s, h, l) = \begin{cases} \{\delta(q, s, h, l)\} & \text{if } q \in Q - \{q_0\} \\ \{\delta'(q, s, h, l)\} & \text{if } q \in Q' - \{q_0'\} \end{cases}$$

$$\delta''(q_0, s, h, l) = \{\delta(q_0, s, h, l)\} \cup \{\delta'(q_0, s, h, l)\}.$$

Thus, since $Q \cap Q' = \varnothing$, once a first transition has been selected, the automaton M'' is locked into one of the two automata M and M'. Hence $L(M'')=L \cup L'$.

This theorem guarantees the validity of uniting several UQAs into one single UQA, and it illustrates the process of constructing UQA for disjunction operator (as section 4.1 indicates).

Definition 3. Language $A_{UQA}=\{\langle M,w\rangle \mid M$ is a UQA and w is a string such that M accepts $w\}$. Language A_{UQA} includes all *UQA*s and the strings they accept, $\langle M,w\rangle$ is an element of A_{UQA} means that *UQA M* accepts w. The problem of whether a given *UQA* accepts a given string turns to the problem of whether A_{UQA} is decidable.

Theorem 3. A_{UQA} is a Turing-decidable language.

Proof. The proof process is very simple. We only need to construct a Turing Machine TM which is capable of deciding A_{UQA}. Let TM = "to input $\langle B,w\rangle$ where B is a *UQA* and w is a string:

1) Simulate *UQA* on input string w.

2) If simulation ends with accepting status, then TM accept; if simulation ends with rejecting status (at that time the queue is empty), or simulation ends with queue being not empty, TM reject.

This theorem guarantees that a UQA can always reject or accept a given input string, and that no cycle or deadlock will be possible when recognizing arbitrary input string.

4 Using BQA to analyze composite events/actions

4.1 Basic operators and time event

The operators follow-by, disjunction and closure is corresponding to the regular operators union, joint and closure in regular grammar respectively (Table 1).

Table 1 Corresponding relations between event operators and regular operators

Event operators	$a()$ -> $b()$	$a()$ | $b()$	$a()^*$
Regular operators	$(R_a \circ R_b)$	$(R_a \cup R_b)$	R_a^*

To conjunction operator ^, $a()$^$b()$=($a()$->$b()$)|($b()$->$a()$), operator ^ can be simulated by operator -> and |. Hence the language L produced by these four operators is regular language, according to Theorem 1, L can be recognized by a *UQA*.

Referring to the proof process of Theorem 2, it is easy to construct a concrete *UQA* and build diagram for disjunction operator |, so do operator -> and – (Fig.2).

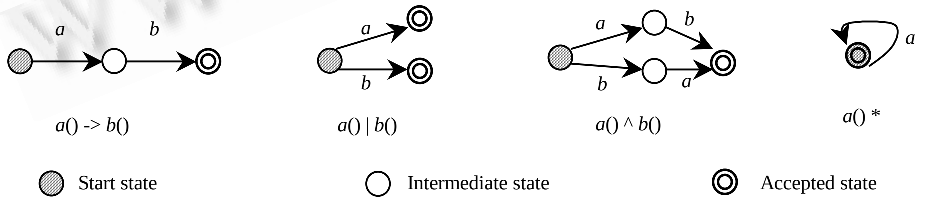

Fig.2 Constructing UQA for operators ->, |, ^ and *

To the without operation -, for instance, $a()$->$b()$–$c()$=$a()$->($b()$–$c()$), after event a occurs, event b's arrival without the interfering of event c will cause the equation to be accepted, or it will be rejected. Rejected states are useful in these cases to describe the complicate relation of there events. Similarly we can draw the state diagrams of $a()\text{->}b()\text{–}c()$ and $(a()\text{–}c()\text{->}b())$ (Fig.3).

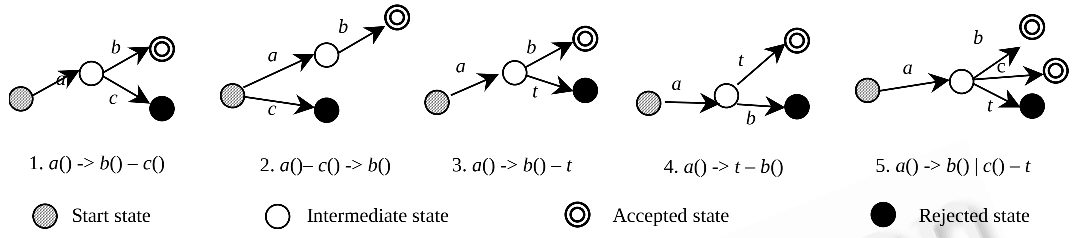

Fig.3 Constructing UQA for operator - and time event

Time is an important element in the management of events. The application can reject those component event results which do not satisfy timing requirements. It is convenient if time constraints can be expressed within a composite event expression.

The semantics of timers change can be in conjunction of the usual event in which timer generate time event *t* after a preset time interval to inform the application. After event *a* occurs at *ta*, to enforce event *b* must occur within time *t*, the timer expires at time a_t+t and generates the event *t*. If event t arrives earlier than event *b*, evaluation of the monitoring thread rejects, indicated by the reject state (Fig.3), otherwise the evaluation accepts. We use language notation $a()\text{->}b()\text{–}t$ to represent this.

To the case of *b* must occur within time *t*, if *b* occurs before the event *t* occurs, the monitoring thread evaluation rejects, the language notation is $a()\text{->}t\text{–}b()$. Now we consider more complicate case which event *b* or event c must occur within time *t*, composite expression of operator ->, | and - can be use to represent it (Fig.3).

4.2 Extended follow-by operators

In Section 2.2 we introduce four follow-by operators ->^1, ->^2, ->^3, ->^4, which are suitable to deal with event stream. Now we describe the process of constructing UQA for each of these operators. Because in event stream events have event type and identify themselves with variables taken as parameters, we need to modify the concept of symbol to contain parameters in symbols, particularly, symbol $b(x)$ refers to a type *b*'s event with parameter *x*, furthermore symbol $b()$ refers to *a* event belonging to type *b* but its parameter can be arbitrary, in other words $b()$ may be $b(x)$ or $b(y)$ and so on.

Let $t\subseteq\Gamma^*$ represents the event sequence in the circle queue, the transition functions corresponding to follow-by operators are as following:

✧ ->^1. $(q_2\in F_a)$

$\delta(q_1, a(x), \text{“\$\#”}) = (q_1, \text{“\$}a(x)\text{\#”}, \_)$ //insert new event $a(x)$ into the queue

$\delta(q_1, a(y), \text{“\$}a(x)\text{\#”}) = (q_1, \text{“\$}a(y)\text{\#”}, \_)$ //delete old event $a(x)$ and insert new event $a(y)$

$\delta(q_1, b(\,), \text{“\$}a(x)\text{\#”}) = (q_2, \text{“\$\#”}, x)$ //output the most recently event $a(x)$

✧ ->^2. $(q_2\in F_a)$

$\delta(q_1, a(x), \text{“\$\#”}) = (q_1, \text{“\$}a(x)\text{\#”}, \_)$ //insert new event $a(x)$ into the queue

$\delta(q_1, a(y), \text{“\$}a(x)\text{\#”}) = (q_1, \text{“\$}a(x)\text{\#”}, \_)$ //event $a(x)$ in queue is keeped intact

$\delta(q_1, b(\,), \text{“\$}a(x)\text{\#”}) = (q_2, \text{“\$\#”}, x)$ //output the least recently event $a(x)$

✧ ->^3. $(q_3\in F_a)$

$\delta(q_1, a(x), \text{“\$}t\text{\#”}) = (q_1, \text{“\$}ta(x)\text{\#”},)$ including $\delta(q_1, a(x), \text{“\$\#”}) = (q_1, \$a(x)\#, \_)$

//insert new event a(x) into the queue

$\delta(q_1, b(\,), \text{“\$}a(x)t\text{\#”}) = (q_2, \text{“\$}t\text{\#”}, x)$ //output the most recently event $a(x)$

$\delta(q_2, \varepsilon, \text{“\$}a(x)t\text{\#”}) = (q_2, \text{“\$}t\text{\#”}, x)$ //output less recently event $a(x)$

$\delta(q_2, \varepsilon,$ “\$#”$) = (q_3,$ “\$#”, _) //after output all events in queue, UQA stops

✧ $->^4$. $(q_3 \in F_a)$

$\delta(q_1, a(x),$ “\$$t$#”$) = (q_1,$ “\$$ta(x)$#”, _) including $\delta(q_1, a(x),$ “\$#”$) = (q_1,$ \$$a(x)$#, _)

//insert new event $a(x)$ into the queue

$\delta(q_1, b(\,),$ “\$$t$#”$) = (q_2,$ “\$%t#”, _) //add a new special symbol % as the delimiter

$\delta(q_2, \varepsilon,$ “\$$a(x)t$#”$) = (q_2,$ “\$$ta(x)$#”, $x)$

//output the most recently event $a(x)$, delete it from the head and insert it into the tail

$\delta(q_2, \varepsilon,$ “\$$t$%#”$) = (q_1,$ “\$$t$#”, _)

// delimiter % is in the tail, all events have been outputted but stay in the queue, avoid endless cycle

For the convenient purpose, these transition functions are described by the change of queue’s content, in fact they can be simulated by traditional format function. For example, $\delta(q_1, a(x),$ “\$#”$) = (q_1,$“\$$a(x)$#”, _) equals to $\delta(q_1,a(x),\varepsilon, \varepsilon) = (q_1, \varepsilon, a(x),$ _). Another example, $\delta(q_1, a(y),$ “\$$a(x)$#”$) = (q_1,$ “\$$a(y)$#”, _) can be simulated by sequence $\delta(q_1, a(y), \varepsilon, a(x)) = (q_2, a(x), a(y),$ _), $\delta(q_2, \varepsilon, a(x), a(y)) = (q_3, \varepsilon, a(y),$ _).

Example. With respect to event stream “$a1$, $a2$, $b1$, $a3$, $b2$” mentioned in section 3.2, $a() ->^3 b()$ results in ($a1$, $b1$), ($a2$, $b1$), ($a3$, $b2$), ($a4$, $b3$), the corresponding UQA=$(Q,\Sigma,\Gamma,\Delta,\delta,q_0,F_a,F_r)$ is $Q=\{q_0,q_1,q_2\}$, $\Sigma=\{a1, a2, a3, b1, b2\}$, $\Gamma=\{a1, a2, a3\}$, $\Delta=\{1, 2, 3, \_\}$, $F_a=\{q_2\}$, $F_r=\varnothing$, δ is transition function (Table 2).

Table 2 Transition function δ of a UQA

Input event		$a()$		$b()$		ε	
Event queue		\$$t$# (e.g. \$#)	\$$a(x)t$#	\$$t$#	\$$a(x)t$#	\$#	\$$a(x)t$#
States	q_0	$(q_1$,\$$a(x)t$#, _)					
	q_1				$(q_1$,\$$t$#$,x)$	$(q_2$,\$#, _)	$(q_1$,\$$t$#$,x)$
	q_2					$(q_0$,\$#, _)	

We show the process of defined UQA’s accepting event stream “$a1$, $a2$, $b1$, $a3$, $b2$” (Table 3).

Table 3 Steps of a UQA recognizing a event stream

Input Events		$a1$	$a2$	$b1$	ε	ε	ε	$a3$	$b2$	ε
States	q_0	q_0	q_0	q_1	q_1	q_2	q_0	q_0	q_1	q_0
Queue	\$#	\$$a1$#	\$$a1a2$#	\$$a1$#	\$#	\$#	\$#	\$$a3$#	\$#	\$#
Output				2	1				3	

4.3 Event count and bounded UQA (BUQA)

Consider the following event stream, where after four events with type a occur before event $b1$ arrives, and then three a events occur before $b2$ arrives, in these cases we must take into consideration of the number of events. We further introduce a new follow-by operator $->^5$ which $a ->^5(n)\ b$ denotes that before event b must have n precedents of type a. Then the following event stream can be accepted by $a ->^5(4)$ b and a $->^5(3)\ b$.

$a1$, $a2$, $a3$, $a4$, $b1$, $a5$, $a6$, $a7$, $b2$,

We now introduce a extend UQA to recognize the expressions consisting this new operator $->^5(n)$.

Definition 4. Bounded Unidirectional Queue Automata (BUQA) with size n is a *UQA* $(Q,\Sigma,\Gamma,\Delta,\delta,q_0,F_a,F_r,S_q)$ where max $\{t\in\Gamma^*\}=S_q=n$.

Theorem 4. Let language $A_{UQA}=\{\langle M,w\rangle|\ M$ is a BUQA and w is a string such that M accepts $w\}$, A_{UQA} is a Turing-decidable language.

Proof. Using reduction on computing history of M’s configuration, this proven process is analogous to that about Lineal Bounded Automata[4], we neglect it here.

Because head symbol ‘\$’ and tail symbol ‘#’ occupy two slots of the circle queue, the size n of a BUQA equals the subtraction of the size of circle queue and two. We need to prove that a BUQA is capable of recognizing language containing operator $->^5(n)$.

Theorem 5. For a given n, an arbitrary symbol $x\in\Sigma$, there is a BUQA accepting language $L=\{x^n\Sigma^1\}$.

Proof. To prove this theorem we can construct a BUQA which accepts L, the essential point is to find the transition function which only accept a new symbol, for example, y, after receiving n identical symbols x. Let $Q=\{q_0,q_1,q_2\}$, $\Sigma=\{x,y\}$, $\Gamma=\{x\}$, $\Delta=\{x,\_\}$, $F_a=\{q_1\}$, $F_r=\{q_2\}$, $S_q=n+2$ and

$\delta(q_0, x, \text{“\$\#”}) = (q_0, \text{“\$}x\text{\#”}, \_)$ //insert symbol x into the empty queue

$\delta(q_0, x, \text{“\$}x^m\text{\#”}) = (q_0, \text{“\$}x^{m+1}\text{\#”}, \_)$ //$t= x^m$, insert symbol x into the queue

$\delta(q_0, x, \text{“\#\$”}) = (q_1, \text{“\#\$”}, \_)$ //‘#’ and ‘$’ are adjacent, queue become full

$\delta(q_0, y, \text{“\# }x^m\text{\$”}) = (q_2, \text{“\# }x^m\text{\$”}, \_)$ //when queue is not full and y arrives, reject

$\delta(q_0, y, \text{“\#\$”}) = (q_1, \text{“\#\$”}, x)$ //when queue is full and symbol y arrives, output x

$\delta(q_1, \varepsilon, \text{“\$}x^m\text{\#”}) = (q_1, \text{“\$}x^{m-1}\text{\#”}, x)$ //output x until the queue is empty

$\delta(q_1, \varepsilon, \text{“\$\#”}) = (q_0, \text{“\$\#”}, \_)$ //the number of x being outputted is $n=S_q$

BUQA add x into the queue continuously until queue’s content is “#$”, which means that the queue is full, then if the next symbol is y, BUQA reaches state q_1 and accepts the input string x^ny. If the next symbol is still x, BUQA reaches state q_2 and rejects the input string x^{n+1}. BUQA accepts if and only if exactly n event x arrives before y arrives, hence L can be recognized by BUQA.

From the proof process we get the law about using BUQA to recognize expressions consisting operator $->^5(n)$ such as $a()\ ->^5(n)b()$.

4.4 Composite actions

Composite actions consist of expressions containing serialize operator ‘;’, parallelize operator ‘||’ and inherit operator ‘&’, We implement composite action support by making action expression as the output of UQA for actions depend on the result of composite event evaluation. Suppose Δ' and δ are the original output alphabet and transition function respectively, the output alphabet and transition function is modified to $\Delta=\Delta'\cup\{;, \|, \&\}$ and $\delta{:}Q\times\Sigma_\varepsilon\times\Gamma_\varepsilon\times\Gamma_\varepsilon\rightarrow P(Q\times\Gamma_\varepsilon\times\Gamma_\varepsilon\times\Delta^*)$ respectively. Because the mechanism of composite action support is similar to that of composite event support, we neglect relevant details here.

5 Related Works and Extension

Many systems support asynchronous event delivery and management. CORBA Event Services[5] suggests the push and pull models for event delivery, where the delivery is driven by the event producer and the event consumer respectively. Odyssey[1] provides an API aimed at supporting alternate file access policies. In some systems[3], an environmental change is modeled as an asynchronous event, and operations on actions include trigger ‘()’, serialize ‘;’ , parallelize ‘||’, extend ‘$\oplus$’ and compose ‘$\circ$’, which is much similar to our composite action operators. However, they suffer from lack of expressive capacity and support of composite events, which are focuses of our paper.

CALAIS[6] allow applications to more easily handle event services by providing composite events management. CALAIS system is extended with a CORBA interface specified in IDL. The composite operations include follow-by, disjunction, without which is similar to our scheme. However, the data structures used and the mechanism for composite event detection is not described, the capacity of dealing with various kinds of composite events is much weaker than that of our scheme, for example, follow-by ->2, $->^5(n)$ is not supported in CALAIS’s composite event services.

At the aspects of other fields other than mobile computing systems such as describing composite event support in active database[7], events can be combined using logical operators such as AND, OR and NOT or other event constructs related to temporal events including HISTORY. The constructs Disjunction, Conjunction, Sequence, History, Negation, Closure is similar to our constructs |, ^, ->1, $->^5$, - and *. For detecting composite events, a step-by-step approach is adopted. The detector needs to know the ordered sequence of all component events that comprise the composite event. Extended Syntactic Trees is used in which the flow of tokens propagates through the trees. This approach is complex and time consuming compared with our automata scheme.

LOTOS (Language of Temporal Ordering Specification)[8,9] has two special operators – parallel composition and disruption. Basic LOTOS has operators including action-prefix ‘;’, choice ‘[]’, sequential composition ‘>>’, disabling ‘[>’, parallel composition ‘|[‘ ‘|]’, relabelling ‘[]’, etc. Time operators include timed action-prefix, timed interrupt, timeout and so on. Most LOTOS operators such as choice, sequential composition, disabling, parallel composition, timed interrupt and timeout can be simulated by our operators. To the event structures with conflict relation models for modeling conflict and a bundle relation for modeling causality[10], the operations for times event structures can be simulated by our operators too.

6 Conclusions

The development of an adaptive mobile computing system coincides with the rising need of event management, which can recognize certain event occurrences and act upon them automatically without user intervention. The ability to specify and monitor a rich set of composite events enhances the power of the adaptive mobile computing system to cater for a wider range of application domains. With the help of unidirectional queue automata, detection and for and handling of composite events and actions becomes easier.

References:

[1] Noble, B.D. Mobile Data Access [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1998.

[2] Watson, T. Application design for wireless computing. In: Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications. Santa Cruz, California, 1994.

[3] Welling, G.S. Designing Adaptive environment-aware application for mobile computing [Ph.D. Thesis]. Rutgers, NJ: The State University of New Jersey, 1999.

[4] Michael, S. Introduction to the Theory of Computation. New York: Thomson, 1997. 19~110.

[5] Object Management Group. Common object request broker architecture.2002.http://www.omg.org/.

[6] Nelson, G.J. Context-Aware and location systems [Ph.D. Thesis]. London: University of Cambridge, 1998.

[7] Tan, C.H., Goh, A. Composite event support in an active database. Computers & Industrial Engineering, 1999,37(4):731~744.

[8] Langerak, R. Bundle event structures: a none-interleaving semantics for LOTOS. In: Diaz, M., Groz, R., eds. Formal Description Techniques V. North-Holland, 1993. 331~346.

[9] Ed, B., Joost, P.K., *et al*. Partial order models for quantitative extensions of LOTOS. Computer Networks and ISDN Systems, 1998, 30:925~950.

[10] Joost, P.K., Christel B., Diego L. Metric semantics for true concurrent real time. Theoretical Computer Science, 2001,254:501~542

通过单向队列自动机对适应性移动计算系统中组合事件的分析

李国东, 张德富

(南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

摘要: 近年来,移动计算吸引着越来越多的注意力.适应性移动计算系统要求大范围的事件组合,这些事件包括逻辑事件、时间事件和暂时事件等.致力于组合事件和组合反应的设计和实现,考虑了事件流、时间事件和不同操作符等情况.组合事件的检测由一种新的扩展自动机——单向队列自动机来有效地支持,详细定义和讨论了单向队列自动机,并具体描述了使用它来检测组合事件的机制和所使用的数据结构.

关键词: 移动计算;适应性;组合时间;自动机;单向队列

中图法分类号: TP311 文献标识码: A