UNIFY EVENT-BASED AND RULE-BASED STYLES FOR DEVELOPING CONCURRENT AND CONTEXT-AWARE REACTIVE APPLICATIONS

Toward a convenient support for concurrent and reactive programming

Truong-Giang Le[1], Olivier Hermant[1], Matthieu Manceny[1], Renaud Pawlak[2], and Renaud Rioboo[3]

[1]*LISITE - ISEP, 28 rue Notre-Dame des Champs, 75006 Paris, France*

[2]*IDCapture, 2 rue Duphot, 75001 Paris, France*

[3]*ENSIIE, 1 square de la Résistance, F-91025 Évry CEDEX, France*

{*le-truong.giang, olivier.hermant, matthieu.manceny*}*@isep.fr, renaud.pawlak@gmail.com, renaud.rioboo@ensiie.fr*

Keywords: Event-based Programming, Rule-based Programming, Concurrent Applications, Context-awareness, Reactive Applications.

Abstract: We propose a new programming language called INI, which combines both event-based and rule-based styles and is suitable for building concurrent and context-aware reactive applications. In our language, both events and rules can be defined intuitively and explicitly, in a stand-alone way or in combination. Events in INI can run in parallel in order to handle multiple tasks concurrently and may trigger actions defined in related rules. Besides, events can interact with the execution environment to adjust their behaviors if necessary and response to unpredicted changes. This makes INI a convenient language to write many kinds of programs which need to take advantages of concurrency and context-awareness, such as embedded software, interactive applications, sensors applications, robotic systems, etc.

1 INTRODUCTION

Context-aware reactive applications are intelligent applications that can monitor the running context by registering event handlers. In case of changes in this context, they may adapt their behavior if needed and react accordingly in order to satisfy the user's current needs or anticipate the user's intentions (Daniele et al., 2009). Such aware systems have become one of the most exciting concepts in ubiquitous computing, with a wide range of application areas, e.g. monitoring and controlling systems. Ideally, since multiprocessors are now very widespread, a context-aware reactive system should use multithreading so that it may handle multiple tasks in parallel (Sandén, 2011).

Context-aware computing was first discussed by Schilit *et al.* (Schilit and Theimer, 1994). Since then, there have been numerous attempts to support context-aware computing. However, writing context-aware reactive applications is still challenging. The main reason is that we need a well-defined mechanism to handle with widely varied sources of context information. In order to support programmers to write these kinds of applications more intuitively and straightforwardly, we develop a new language called INI. INI compounds both rule-based and event-based paradigms, which are appropriate styles to monitor and react to changes in the environment. Although several event-based and rule-based programming language have been proposed so far (Cohen and Kalleberg, 2008; Baillie et al., 2010; Mircea Marin, 2006; Giurca et al., 2009), they still do not provide a flexible and intuitive way to define events and rules as in INI. Another advantage of INI is that it supports multithreading to speed up the execution.

The rest of this paper is organized as follows. In Section 2, we introduce INI, including its syntax, informal semantics and type system. We discuss a case study of using INI to write concurrent context-aware reactive applications in Section 3. Section 4 concludes the paper.

2 PROGRAMMING WITH INI

INI combines both event-based and rule-based programming styles. With event-based programming, changes in the operational environment can be eas-

ily captured and handled during execution. With rule-based programming, a program may react straightforwardly to changes.

2.1 Functions and Rules in INI

Each INI program contains functions, which combine event expressions, logical expressions (used to specify the conditions to trigger) and the actions (lists of statements) bound to them. The scope of all variables is the whole function. A function in INI has the following syntax:

```
function <name>(<parameters>) {
  <logical_expression> { <statements> }
  | <event_expression> { <statements> }
  | <event_expression>  <logical_expression>
   { <statements> }
}
```

A rule in INI consists of a logical expression and a corresponding action. When the logical expression part of a rule is evaluated to *true*, the action is invoked. INI also allows the use of logical expressions along with events, and in this case they play the same prerequisite role as in rules.

2.2 Events in INI

(Mühl et al., 2006) defines events as any happening of interest that can be observed by a system. An event instance in INI starts with @ and takes input and output parameters, both are optional. The scope of output parameters is the whole function, like other variables. Moreover, an event can also be bound to an id and synchronized on other events through *$(id1,...,idN)* construct (will be explained later). The syntax of event instances is shown below.

```
$(id1,id2,...,idN) id0:@eventKind
[inputParam1=value1, inputParam2=value2, ...]
(outputParam1, outputParam2, ...)
{ <statements> }
```

Programmer may use built-in events (listed in Table 1), or write user-defined events (in Java or in C/C++), and then integrate them to their INI programs (Le, 2012).

By default, except for the @*init* and @*end* events (see Table 1), all INI events are executed asynchronously. However, in some scenarios, a given event *id0* may want to synchronize on other events *id1,..., idN*. It means that the synchronizing event *id0* must wait for all running threads corresponding to the target events to be terminated before running. For instance, when *id0* affects the actions defined inside other events, we need to apply the synchronization mechanism.

Table 1: Some built-in events in INI.

Built-in event	Meaning
`@init()`	Used for initialization, when a function starts.
`@end()`	Invoked when no more rule or event in the function can run or when the function is about to return.
`@every [time:Integer]()`	Triggers an action, period is indicated by its input parameter (in milliseconds).
`@update[variable:T] (oldValue:T, newValue:T)`	Invoked when the value of an observed variable changes during execution.

Events in INI may also be reconfigured at runtime to adapt their behaviors to changes happening in the environment. Programmers can call the built-in function *reconfigure_event(eventId, [inputParam1 = value1, inputParam2 = value2,...])* in order to modify the the values of event's input parameters (see Section 3 for example). To understand more about the algorithm we use to implement events synchronization in INI, and also the event reconfiguration mechanism, please refer to our previous paper (Le et al., 2011).

2.3 Example

To illustrate rules and events in INI, let us consider an automatic lighting control system in the corridor (Figure 1). In our program, there is one user-defined event called @*motionDetection*, that is applied to detect movement. This event has one input parameter named *mode*, which is set to point out whether we apply a simple algorithm or an advanced one for detecting motion. Whenever a motion is detected, our program will turn on the lights if they were turned off before (lines 9-12). The event @*every* at lines 15-17 is applied to set how long does it pass without any motion. If there is no movement within fifteen minutes and the current lights state is on, a rule at lines 18-21 will be invoked to turn off the lights to save energy.

2.4 Type System in INI

2.4.1 Built-in Types

INI comes with 5 built-in types for numbers (*Double*, *Float*, *Long*, *Int*, and *Byte*), a *Char* type, and a *Boolean* type. Besides, INI provides built-in map types: $Map(K,V)$. Map types are polymorphic types with a key type K and a value type V. Lists are instances of maps whene $K = Int$ (in reality, it is more

```
function main() {
 @init() {
  lightOn = false
  timeWithNoMotion = 0
 }
 @motionDetection[mode = "simple"]() {
  timeWithNoMotion = 0
  case {
   !lightOn {
    //Turn on the lights
    lightOn = true
   }
  }
 }
 @every[time = 60000]() {
  timeWithNoMotion++
 }
 timeWithNoMotion > 15 && lightOn {
  //Turn off the lights
  lightOn = false
 }
}
```

Figure 1: An automatic lighting control system.

of an indexed set). A *String* type is a list of *Char*. Syntactically, lists can be noted with the * notation:

$$T* \widehat{=} Map(Int, T)$$

INI types are ordered with a subtyping relation $\succ$. By default, numerics are ordered so that it is not possible to assign more generic numbers to less generic numbers.

$$Double \succ Float \succ Long \succ Int \succ Byte$$

Other conversions must be done by using built-in functions (Le, 2012).

2.4.2 User-defined Types

To define a new type, the programmer uses the *type* keyword followed by a name starting by an uppercased letter. For example, we can use a *Person* type as:

```
type Person = [name:String, age:Int]
p = Person[name="Giang", age=28]
println("Info: " + p.name + " is " + p.age)
```

Field initialization is not mandatory and field access is done with a usual "dot". For instance, one can construct a person with undefined age or name. Moreover, types can be recursive. For example, one can use the *Person* type within the *Person* type definition itself. We can refine the above *Person* type as:

```
type Person = [name:String, age:Int, father:
 Person, mother:Person, children:Person*]
```

Note that INI also performs type-checking on object fields. In order to understand more about types in INI, please refer to INI Language Reference Documentation (Le, 2012).

2.4.3 Type Inference

INI provides *type inference*, so that the programmers may leave types implicit. For instance, the *i=0* statement will assign to *i* the *Int* type. If the programmer tries to set the type of *i* to any other type within the definition scope of *i*, (for instance with the *i=0.1* statement that assigns i with a *Float* type) the INI type checker will raise a type mismatch error.

Most types in INI are calculated with the type inference engine. The core of this type inference is based on a Herbrand unification algorithm, as depicted by Robinson in (Robinson, 1965). The typing algorithm is enhanced with polymorphic function support, abstract data types (or algebraic types) support, and with internal sub-typing for number types.

3 CASE STUDY

Figure 2 shows an intelligent virtual personal assistant written in INI, which can recognize voice commands from users and then do appropriate actions. One of the most interesting features of our program is that it can detect who is using it (based on face detection), in order to adjust the voice recognition process with regard to his or her maternal language, tones and accents. This data is previously collected and stored in a database during a speech training procedure. As a result, the accuracy of voice recognition will be improved.

There are two user-defined events in our program. The event *@faceRecognition* (identified by f) is applied to detect a human face. This event has one output parameter called *recognizedId*, which is used to indicate the corresponding id (if existing) of the user. If a new face is detected, we stop the event *@voiceRecognition* (identified by v), reconfigure it depending on whether the user has been recognized or not, and restart the event v in order to improve the performance of voice recognition. The event *@voiceRecognition* is used to recognize user's voice. It has one input parameter specifying the id of the user, which is applied in order to tune the recognition pattern. Moreover, there is one output parameter called *voiceCommandString*, which is the returned recognized spoken sentence. At line 28, we match the user's command against a regular expression to guess its meaning (by using the match operator $\sim$), and then do a suitable action. The

```
function main() {
 @init() {
  defaultId = 1
  currentId = 1
 }
 $(v) f:@faceRecognition(isKnown,
  recognizedId) {
  case {
   //A familiar person is detected,
   //then change current settings
   //to new settings
   isKnown && recognizedId != currentId {
    currentId = recognizedId
    stop_event(v)
    reconfigure_event(v,[userId=currentId])
    restart_event(v)
   }
   //A stranger is detected,
   //then use default settings
   !isKnown {
    ...
   }
  }
 }
 v:@voiceRecognition[userId = defaultId]
  (voiceCommandString) {
  case {
   voiceCommandString ~ regexp(...) {
    //Do action 1
   }
   ...
   default {
    //Do a default action
   }
  }
 }
}
```

Figure 2: An intelligent virtual personal assistant.

event f should be synchronized on the event v, which means that if there is a current running thread for v, f has to wait before it can be executed. The synchronization is necessary since we want to avoid unfinished voice recognition jobs to be stopped.

4 FUTURE WORK

For future work, we will develop a context recovery mechanism in INI so that when the reaction fails, our program can recover to the previous stable state. Additionally, we will define formal semantics for INI with thoughtful consideration of parallel and concurrent aspect. Operational semantics, denotational semantics or axiomatic semantics with some exetensions can be taken into account. Furthermore, we also have a plan to apply INI in more real applications in order to better evaluate its capabilities.

ACKNOWLEDGEMENTS

The work presented in this article is supported by the European Union. Europe is committed in Ile-de-France with the European Regional Development Fund.

REFERENCES

Baillie, J.-C., Demaille, A., Hocquet, Q., and Nottale, M. (2010). Events! (reactivity in Urbiscript). *CoRR*, abs/1010.5694.

Cohen, N. H. and Kalleberg, K. T. (2008). EventScript: an event-processing language based on regular expressions with actions. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, pages 111–120, New York, NY, USA. ACM.

Daniele, L. M., Silva, E., Ferreira, L., and van, M. S. (2009). A SOA-based platform-specific framework for context-aware mobile applications. In *Enterprise Interoperability*, volume 38 of *Lecture Notes in Business Information Processing*, pages 25–37, Berlin Heidelberg. Springer Verlag.

Giurca, A., Gasevic, D., and Taveter, K. (2009). *Handbook of Research on Emerging Rule-based Languages and Technologies: Open Solutions and Approaches*. IGI Publishing, Hershey, PA, USA.

Le, T.-G. (2012). Ini Online. https://sites.google.com/site/inilanguage .

Le, T.-G., Hermant, O., Manceny, M., and Pawlak, R. (2011). Dynamic adaptation through event reconfiguration. In Meersaman, R., Dillon, T., and Herrero, P., editors, *On the Move to meaningful Internet Systems: OTM 2011 Workshops*, volume 7046 of *Lecture Notes in Computer Science*. Springer.

Mircea Marin, T. K. (2006). Foundations of the rule-based system pLog. *Journal of Applied Non-Classical Logics*, 16:151–168.

Mühl, G., Fiege, L., and Pietzuch, P. (2006). *Distributed Event-Based Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Robinson, J. A. (1965). A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41.

Sandén, B. (2011). *Design of Multithreaded Software: The Entity-Life Modeling Approach*. John Wiley & Sons.

Schilit, B. and Theimer, M. (1994). Disseminating active map information to mobile hosts. *IEEE Network*, 8:22–32.