Dynamic Adaptation through Event Reconfiguration

Le Truong Giang , Olivier Hermant
Matthieu Manceny and Renaud Pawlak

LISITE - ISEP, 28 rue Notre-Dame des Champs,
75006 Paris, France
{le-truong.giang,olivier.hermant,
matthieu.manceny,renaud.pawlak}@isep.fr

Abstract. *We introduce a new programming language called INI, which eases the development of self-adaptive software. INI combines both rule-based and event-based programming paradigms, by allowing the definitions of rules that can be triggered by events. Besides reacting to existing events, INI programmers can also write their own support for new events in Java or in C. Since events in INI are asynchronous, the programming model is inherently multithreaded, which implies that INI provides a simple but convenient language-level support for synchronization. Besides synchronization support, the key point with INI events is that they come with a configuration level, which is set up through input parameters passed to the events when created. Additionally, events can be stopped, reconfigured, and restarted at runtime by the program itself. Ultimately, this re-configuration can be triggered by other events, thus allowing the program to adapt to new operational environments.*

Keywords: Adaptive systems, rule-based programming, event-based programming, reconfiguration

1 Introduction

The role of software in our society has become more and more important and to satisfy higher demands from customers, software systems must become more versatile, flexible, resilient, dependable, robust, energy-eficient, recoverable, customizable, configurable, and self-optimizing [1]. However, developing and maintaining software systems is hard and resource-consuming because they are operating in an environment that is not well defined or predictable. Consequently, software systems need a new and innovative approach for evolving and running. One of the most promising approaches to achieve such properties is to equip software systems with self-adaptation and self-management mechanisms [8].

In recent years, with the development of autonomic computing, software engineering researcher has been motivated to pay more attention to design and develop self-adaptive software systems. The most important difference between

traditional software and adaptive software is related to the assumption of dynamic environment [14]. Most self-adaptive software use rules explicitly or implicitly to decide how to react to monitored events happening during execution [13]. Therefore, one of the current research trends is to develop a programming language to easily express and capture those changes so that the systems can modify their behavior at runtime. In this paper, we introduce a new programming language called INI, which we created to assess rule-based and event-based oriented programming. In INI, we allow programmers to define events, which need to be captured and monitored. Events are usually executed asynchronously but can synchronize on other events with intuitive language constructs. Moreover, events can be reconfigured at runtime through a well-defined reconfiguration mechanism. As a result, when something happens during execution, INI program can capture and adapt to these changes automatically by reconfiguring its events.

The rest of this paper is organized as follows. In Section 2, we give an overview of related work. In Section 3, we discuss events in INI, including its syntax, built-in events and user-defined events. Events synchronization and reconfiguration are shown in Section 4. In this section, we also demonstrate the capabilities of INI in handling variabilities in the operational environment with a small example. Finally, some conclusions and future work are presented in Section 5.

2 Related work

A general overview of research and problems on self-adaptive software can be found in [7, 12]. The key aspect of self-adaptive software is that code behavior is evaluated or tested at runtime, which may lead to a run-time change in behavior [7]. As a result, the run-time code should contain the following mechanisms:

- A mechanism to detect changes during execution
- A mechanism to switch algorithms and operations due to these above changes

Laddaga [7] discussed two useful paradigms for utilizing self-adaptive software. One of these is the planning paradigm, and the other is the control paradigm. In [13], Wang proposed a Rule Model for self-adaptive software, which includes three key concepts (event, parameter, and rule). In Wang's paper, rules were used to adjust software's behavior at runtime. Psaier *et al.* [11] presented a middleware for programming and adapting complex service-oriented systems. Their approach was based on monitoring and real-time intervention to regulate interactions based on behavior policies.

Cheng *et al.* [2] introduced a new language of adaptation with the ultimate aim is to automate human tasks in system management to achieve high-level stakeholder objectives. They claimed that this language is expressive enough to represent human expertise and the flexibility and robustness to capture complex and potentially dynamic preferences.

In our work, we develop INI by combining both rule-based and event-based programming styles. INI takes inspirations from existing rule-based languages [3], and event-based programming as described above through event-triggered rules.

With rule-based programming, an action is executed only if its guard is evaluated to true in the context of the function. With event-based programming, we allow programmers to write their own events to capture and monitor changes happening during execution. Moreover, we fully support events synchronization and reconfiguration. Although several event-based programming languages have been proposed so far [4, 6], their support for writing self-adaptive software are limited. To our knowledge, no languages support event reconfiguration at runtime. The key feature of INI compared to all the languages we have studied is that events are dynamic reconfigurable. This feature allows INI programs to automatically reconfigure themselves to adapt to changes in the environment.

3 Programming with INI

The current INI implementation is built and runs on Java but its syntax and semantics are not Java's. All functions are written with rules in combination with event expressions. Each rule is defined explicitly with a guard and an action. The action will be executed only if the guard is evaluated to true in the context of the function. In other words, the guard triggers the evaluation of the action. Along with rules, programmers can use events in INI to capture changes when their programs are running. In this paper, we focus on event-triggered rules since we use them as the main mechanism for self-adaptation.

3.1 Events in INI

Built-in event kind	**Meaning**
`@init()`	The `@init` event is invoked when the function starts evaluating.
`@end()`	On the contrary to the `@init` event, the `@end` event is triggered when no more event can be applied and when the function is about to return.
`@every[time:Integer]()`	The `@every` event occurs periodically as specified by its input parameter (in milliseconds).
`@update[variable:T] (oldValue:T,newValue:T)`	The `@update` event is invoked when a specific variable is changed during execution (e.g., by one of the evaluated events).

Table 1. Some built-in events in INI

An event is defined as something happening during the execution of the program, which should be monitored and handled. In INI, events start with `@`, have a kind, and can take parameters. Optionally, an event can also be bound to an id, so that other parts of the program can refer to it. There are two kinds of parameters that can be used with events. The first one is input parameters, which can be understood as configuration parameters to tune the event execution. The second one is output parameters, which are some data or results we may need to monitor. These parameters may be optional. Syntactically, each event-driven rule is written as:

```
id:@eventKind[inputParam1=value1, inputParam2=value2, ...]
              (outputParam1, outputParam2, ...) {
     <statements>
}
```

To support programmers to write code more easily and conveniently, INI comes with some common built-in event kinds such as `@init`, `@end`, `@every`, `@update`. The meanings of these events are given in Table 1. For example, the following code creates an `@every` event-driven rule called `e`, which increments `v` every second. The `@update` event-driven rule `u` triggers the action (event handler) that prints out the `v` variable value when it changes, i.e. every second.

```
e: @every[time=1000]() { v = v + 1 }
u: @update[variable=v](oldv, newv) { println("v=" + newv) }
```

Additionally to built-in events, we also allow programmers to write their own event kinds in Java or in C and integrate them into an INI program as explained in the next section.

3.2 Example

In this section, we discuss a sample INI program, written in an event-driven style. In our example, we use a video camera to detect the movement of a ball, get its positions in space periodically, and save them into an CSV file.

To do so, we first need to define a new event kind to detect the ball and send its position to the program when detected. Our event has one input parameter called `frequency`. This parameter is applied to set how long the event should sleep between two image detections (time unit is in milliseconds). Besides, we have three output parameters `(r,x,y)`, which are the current radius and coordinates of the detected ball in the captured image. In Java, we need to subclass the `ini.event.Event` class to define the behavior of our new event as shown in the code below.

```
1  public class BallDetection extends ini.event.Event {
2    Thread ballDetectionThread;
3    @Override public void eval(final IniEval eval) {
4      (ballDetectionThread = new Thread() {
5        @Override public void run() {
6          ...
7          do {
8            try {
9              // Sleep as long as the configuration indicates
10             sleep(getInContext().get("frequency")
11               .getNumber().longValue());
12             // Use OpenCV to detect the ball
13               ...
14             // Write data to output parameters
15             variables.put(outParameters.get(0), r);
16             variables.put(outParameters.get(1), x);
```

```
17              variables.put(outParameters.get(2), y);
18              // Execute the event action
19              execute(eval,variables);
20            } catch (Exception e) {...}
21          } while (!checkTerminated());
22        }
23      }).start();
24    }
25    @Override public void terminate() {...}
26  }
```

In the `BallDetection` class, the method `eval` will be upcalled by the INI evaluator when the program uses our event. It creates a thread that sleeps accordingly to the event configuration (line 10), detects the ball using the OpenCV library (line 13) [5] and write the results in output parameters to be passed to the INI program (lines 15-17). The event-triggered action is executed at line 19 using the `execute` method provided by the API, which by default runs asynchronously. Finally, the method `terminate` is overridden to stop the event-driven rule: INI upcalls this method when the program exits or forces the rule to shutdown. To understand more about built-in events in INI and how to write user-defined events, interested readers may refer to INI Language Reference Documentation [10].

```
1 import "lib_io.ini"
2 @ballDetection[Integer](Float, Integer, Integer)
3   => "ini.ext.events.BallDetection"
4 function main() {
5   @init() { f = file("ballData.csv") }
6   // Use our event get notified for ball detection
7   @ballDetection[detection_frequency = 1](r,x,y){
8     case {
9       !file_exists(f) {
10        // Create a new CSV file to store data
11        create_file(f)
12        fprint(f, "Time,␣r,␣x,␣y␣\r\n")
13      }
14    }
15    fprint(f,""+time()+","+r +","+x+","+y+"\r\n")
16  }
17 }
```

Fig. 1. A sample INI program with event-driven style

In Figure 1, we then write the actual INI program, which binds our Java user-defined to the `@ballDetection` event kind (lines 2-3). In the `@init` event, we define a variable `f`, which indicates the CSV file we want to store data in after collecting the ball positions over time. In our program, the `@ballDetection` event is triggered periodically, i.e. each millisecond. If a ball is detected, we check

whether the CSV file exists or not (line 9). If the file does not exist, we create a new one and write the header. Finally, we write the data to the file (line 15), including the time when the ball was detected and its position.

4 Events synchronization and reconfiguration

The event reconfiguration mechanism is developed in INI so that we can modify the input parameters (configuration) of the event-triggered rules at runtime. For example, an event can be notified for a change in the environment and decide to reconfigure another event to adapt the program to the new environment. However, to achieve safe reconfiguration, synchronization among events is also essential. In this section, we first explain event synchronization, and then discuss event reconfiguration, which we illustrate using our ball-detection example.

4.1 Events synchronization

Except for the `@init` and `@end` events, most INI events are executed asynchronously by default. From an implementation perspective, it means that each event handler runs in its own thread, potentially simultaneously to other events. In many cases, a given event may want to synchronize to other events, for instance to ensure that a shared variable accessed both for reading and writing has a consistent state, or also to ensure that all event threads have terminated before reconfiguring the associated event-triggered rule. Thus, INI provides a specific language construct: the symbol `$` along with the list of identifiers of target events with which we want to synchronize. In INI, synchronizing with target events means that the synchronizing event must wait for all the target event threads to be terminated before running. For example, the following code ensures that the event `e0` synchronizes `N` target events.

```
$(e1,e2,...,eN) e0:@eventKind[...](...) { <statements> }
```

Note that one of the target events can also be synchronized with `e0`. Cross-synchronization of events means that their executions are mutually exclusive. We now precisely define the semantics of synchronization in INI.

To implement synchronization in INI, we use one lock and one *count* variable associated with each event. The lock is an instance of `java.util.concurrent.locks.ReentrantLock` and is used to avoid concurrent execution when required. We use the functions `lock` (blocking), `tryLock` (non-blocking and returns true of false depending on whether the locking was successful or not), and `unlock`. For more details, refer to the Java document of the methods of the same names in the `ReentrantLock` class [9]. The *count* variable holds the number of threads currently executing for the associated event. We use the following notation: for an event e_i, l_i is its associated lock and $count_i$ its associated thread counting variable.

Let $(e_1, e_2, ..., e_N)$ be the list of target events with which e_0 synchronizes. To execute the event e_0 in INI, we apply Algorithm 1, which also applies to any

Algorithm 1 Our algorithm to execute e_0 synchronized with $(e_1, e_2, ...e_n)$

```
1: repeat
2:    allLocked := true
3:    lock(l_0)
4:    for i := 1 to N step 1 do
5:       if ¬ tryLock(l_i) then
6:          allLocked := false
7:          for i := 1 to i − 1 step 1 do
8:             unlock(l_j)
9:          end for
10:         unlock(l_0)
11:         sleep(randomBetween(1, 5))
12:         break
13:      end if
14:   end for
15: until ¬allLocked
16: for i := 1 to N step 1 do
17:    wait-until count_i = 0
18: end for
19: count_0 := count_0 + 1
20: for i := 1 to N step 1 do
21:    unlock(l_i)
22: end for
23: unlock(l_0)
24: eval(e_0)
25: count_0 := count_0 − 1
```

event execution in the INI system. We can see that the execution of events in INI includes four steps. First, e_0 locks its own lock and tries to lock all target events (lines 1-15). When all events are locked, the event e_0 needs to wait until all other events are terminated (lines 16-18). The `wait-until` mechanism in our algorithm is currently implemented with Java monitors and thread notification. Next, the number of threads executing for event e_0 is incremented and locks for all events are released (lines 19-23). Finally, the event can be actually evaluated and when it is terminated, the number of running threads for it is decremented (lines 24-25).

4.2 Example using synchronization

Let us take again our example shown in Figure 1. Assuming that after we collect data about positions of a ball in space, we need to upload the CSV file to an FTP server. To do this, we create a new `@every` event as shown in Figure 2. Inside this event, first, we compress data and then upload the compressed file. Since we collect data periodically, we use a timestamp (line 4) to distinguish data at different time.

By default, the two events `@ballDetection` and `@every` will be executed asynchronously. However, it is essential that when we collect data (`@ballDetec-`

```
1  ...
2  // Compress and upload the collected data to an FTP server
3  @every[time = 5000](){
4    t = time()
5    zip(file_name(f), "" + t + "ballData.zip")
6    upload_ftp("ftp_address", "user_name", "password", "" + t
7      + "ballData.zip", "" + t + "uploadedBallData.zip")
8  }
```

Fig. 2. An event used to upload collected data

tion), the uploading process (`@every`) should not happen and vice versa. In other words, these two events need to be mutually exclusive. To ensure this, the programmer needs to modify the program at lines 7 of Figure 1 and 3 of Figure 2 by naming the two events and use their identifiers to cross-synchronize them, as it is shown in the following code:

```
$(e) b:@ballDetection[detection_frequency = 1](r,x,y) { ...
$(b) e:@every[time = 5000](){ ...
```

4.3 Event reconfiguration

In this section, we will demonstrate the capabilities of INI in handling changes happening in the environment though the event-reconfiguration mechanism. Event reconfiguration consists of changing the values of the event-triggered rule input parameters. Programmers can call the built-in function `reconfigure(eventId, [inputParam1=value1, inputParam2=value2,...])` to reconfigure their events. Moreover, we also allow programmers to stop and restart events with the built-in functions `stop_event(eventId)` and `restart_event(eventId)`. Typically, it is required to stop an event before reconfiguring it.

Let us now consider that our ball-detection example of Section 3 runs in an embedded environment where the power is supplied by a battery. One way to take into account this new constraint is to adapt the data-collection frequency to the power level. First, we add a new user-defined event kind called `@powerAlarm`, which notifies the program each time the power level passes a given threshold both ways, when charging or discharging. This event has one input parameter named `threshold`, and one output parameter named `currentLevel`, which tells us if the level is currently lower or greater than the threshold. When the program below is running, if it detects that the power-level is lower than 50% (configured line 1), it stops the event `b:@ballDetection` (line 4), then changes the value of its input parameter (i.e., `detection_frequency`) to 1000 (line 5), and finally restarts it (line 6). Conversely, if the power goes over the threshold the value for the parameter `detection_frequency` is set again to 1 (lines 8 - 10). The event `@powerAlarm` is synchronized with events `b:@ballDetection` and `e:@every` as specified by `$(b,e)` at line 1 in order to avoid unfinished detection or upload jobs to be stopped.

```
...
// Adapt the ball detection frequency at the 50% threshold
$(b,e) a:@alarmPower[threshold = 0.5](currentLevel) {
  case {
    currentLevel < threshold {
      stop_event(b)
      reconfigure(b, [detection_frequency = 1000])
      restart_event(b)
    } default {
      stop_event(b)
      reconfigure(b, [detection_frequency = 1])
      restart_event(b)
    }
  }
}
```

Finally in the following code, we show how to add an event-triggered rule that stops all other events when the threshold goes below 10%.

```
$(b,e) @alarmPower[threshold = 0.1](currentLevel) {
  case {
    currentLevel < threshold { stop_events(b, e, a) }
    default { restart_events(b, e, a) }
  }
}
```

5 Conclusion and future work

In this paper, we introduced a new language called INI, which can be used to write self-adaptive software through events synchronization and reconfiguration mechanisms. Programs written in INI may dynamically change their behavior or improve their operation depending on dynamic operational conditions of the environment. With the example that we used along the paper to demonstrate the capabilities of INI, we can see that using INI to write new event-driven rules does not require too much effort. The programmers can capture the changes during execution by writing their own user-defined events and then define suitable actions inside them. The changes necessary to take into account new constraints can thus be localized and encapsulated within some rules, which implies code maintainability and robustness with regard to changes.

For future work, we will improve INI so that it can handle changes in environment more efficiently. For example, as stated in [2], we need a mechanism to handle failure during adaptation execution. If a failure happens when our systems try to adapt to the environment, it is essential to recover from that failure. Besides, we will define formal semantics for validating and verifying useful properties of INI programs and prove the soundness of its type system. Moreover, we also have a plan to perform more case studies to evaluate the capabilities of INI.

References

1. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.): Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar], Lecture Notes in Computer Science, vol. 5525. Springer (2009)
2. Cheng, S.W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems. pp. 2–8. SEAMS '06, ACM, New York, NY, USA (2006)
3. Cirstea, H., Moreau, P.E., Reilles, A.: TomML: A Rule Language For Structured Data. In: International RuleML Symposium on Rule Interchange and Applications - RuleML 2009. pp. 262–271. Las Vegas, United States (2009)
4. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. pp. 134–143. PEPM '07, ACM, New York, NY, USA (2007)
5. Garage, W.: OpenCV Documentation (Jun 2011), `http://opencv.willowgarage.com/wiki/`
6. Kamina, T., Aotani, T., Masuhara, H.: Eventcj: a context-oriented programming language with declarative event-based context transition. In: Proceedings of the tenth international conference on Aspect-oriented software development. pp. 253–264. AOSD '11, ACM, New York, NY, USA (2011)
7. Laddaga, R.: Self adaptive software problems and projects. In: Proceedings of the Second International IEEE Workshop on Software Evolvability. pp. 3–10. IEEE Computer Society, Washington, DC, USA (2006)
8. Müller, H.A.: Towards self-adaptive software-intensive systems. In: Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops. pp. 7–8. IWPSE-Evol '09, ACM, New York, NY, USA (2009)
9. Oracle: Java SE 6 Documentation (Jun 2011), `http://download.oracle.com/javase/6/docs/`
10. Pawlak, R.: INI Online (Jun 2011), `http://perso.isep.fr/rpawlak/ini/`
11. Psaier, H., Skopik, F., Schall, D., Juszczyk, L., Treiber, M., Dustdar, S.: A programming model for self-adaptive open enterprise systems. In: Proceedings of the 5th International Workshop on Middleware for Service Oriented Computing. pp. 27–32. MW4SOC '10, ACM, New York, NY, USA (2010)
12. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4, 14:1–14:42 (2009)
13. Wang, Q.: Towards a rule model for self-adaptive software. SIGSOFT Softw. Eng. Notes 30, 8– (2005)
14. Yoo, C., Jung, W., Park, D., Lee, B., Kim, H., Wu, C.: An adaptive software framework based on service composition. In: Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications. pp. 476–484. SERA '07, IEEE Computer Society, Washington, DC, USA (2007)