Performance Evaluation of Embedded ECA Rule Engines: A Case Study

Pablo E. Guerrero*, Kai Sachs, Stephan Butterweck, and Alejandro Buchmann

Dept. of Computer Science, Technische Universität Darmstadt
D-64283 Darmstadt, Germany
{guerrero, sachs, butterweck, buchmann}@dvs.tu-darmstadt.de

Abstract. Embedded systems operating on high data workloads are becoming pervasive. ECA rule engines provide a flexible environment to support the management, reconfiguration and execution of business rules. However, modeling the performance of a rule engine is challenging because of its reactive nature. In this work we present the performance analysis of an ECA rule engine in the context of a supply chain scenario. We compare the performance predictions against the measured results obtained from our performance tool set, and show that despite its simplicity the performance prediction model is reasonably accurate.

Keywords: Performance Evaluation and Prediction, Embedded Systems, ECA Rule Engines, Active Functionality Systems.

1 Introduction and Motivation

As software and hardware are becoming more complex, system engineers look more at architectures that help them cope with the speed at which the *business logic* changes. In architectures centered on *rule engines* [1], developers describe the business logic in terms of *rules* composed by *events*, *conditions* and *actions* (hereafter called ECA rules). These ECA rules are precise statements that describe, constrain and control the structure, operations and strategy of a business.

Business logic executes on multiple platforms with different capabilities ranging from clusters, through workstations all the way down to small embedded devices. To relieve developers from knowing in advance on which environment rules will execute, it is convenient to offer a uniform ECA abstraction for all of them. We have developed a complete ECA rule engine middleware [2] which supports a uniform rule definition language across platforms. Our implemented ECA rule engine offers a high level programming abstraction and thus achieves a fast change of re-utilizable business logic.

This flexibility comes at a performance cost. Therefore, the study of this tradeoff is crucial before migrating to an ECA middleware architecture. To avoid overload and unexpected errors, it is important to know the processing limits

* Supported by the DFG Graduiertenkolleg 492, *Enabling Technologies for Electronic Commerce*.

N. Thomas and C. Juiz (Eds.): EPEW 2008, LNCS 5261, pp. 48–63, 2008.

and possible bottlenecks of a rule engine. However, evaluating reactive behavior is not trivial, especially for embedded devices where resources such as processing power, memory and bandwidth are scarce.

The main contribution of this work is an analytical performance model for ECA rule engines. We discuss the difficulties in building a performance model of an ECA rule engine and present one that, despite its simplicity, accurately predicts overall system utilization. The measurement and monitoring of the ECA rule engine and its individual services are supported by our performance evaluation tool set. The model is validated against a case study based on SAP's intention to move business processing towards the periphery [2].

2 Background

2.1 ECA Rule Engines

An ECA rule engine is a software system that executes Event-Condition-Action (ECA) rules [3]. ECA rules contain a) a description of the events on which they should be triggered; b) an optional condition, typically referring to external system aspects; and c) a list of actions to be executed in response. In general, the structure of an ECA rule is `ON <event> IF <condition> THEN <action>`. Events are relevant changes of state of the environment that are communicated to the ECA rule engine via messages, possibly originated at heterogeneous sources.

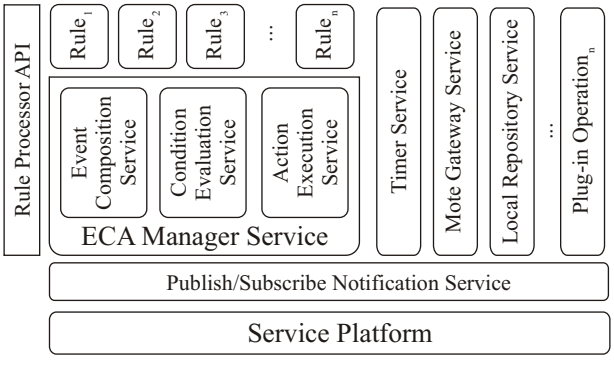


Fig. 1. ECA Rule Manager architecture

Our rule engine, depicted in Figure 1, was designed as a set of independent services, or *bundles*, managed by a service platform. For the embedded implementation we have chosen the Open Services Gateway initiative (OSGi) Service Platform, because of its minimalist approach to service life-cycle management and dependency checking. The services are decoupled from each other via a *Publish/Subscribe* notification service, for which we use a Rebeca [4] event broker wrapped as a service. The *ECA Manager* service exposes a *Rule Processor API* over which rules can be (un)registered and (de)activated. The rule execution is

split and delegated to elementary services, namely *Event Composition*, *Condition Evaluation* and *Action Execution*. Conditions and actions invoke high level functions exposed by other services such as the *Local Repository* service. This plug-in mechanism allows to dynamically extend the set of operations.

2.2 Performance Analysis

Performance has been studied in the context of Active Database Management Systems (aDBMS) [5,6,7,8]. These are complete "passive" DBMS extended with the possibility to specify reactive behavior beyond that of triggers. ECA rule engines, which stem from aDBMS, are more flexible in that they can interact with any arbitrary system, but do not necessarily provide full database functionality.

The work in [9] identified requirements for aDBMS benchmarks. Important aspects were the *response times* of event detection and rule firing, as well as the *memory management* of semi-composed events. The BEAST micro-benchmark [10,11] reported on the performance of various aDBMS available at the time, such as ACOOD, Ode, REACH and SAMOS. BEAST does not propose a typical application to test its performance. Konana et al. [12], in contrast, focus on a specific E-Broker application which allowed the evaluation of latency with respect to event arrival rate, among others. The experiments in [13] evaluate the effects of execution semantics (e.g., immediate vs. deferred execution) by means of simulation.

Previous work has focused on evaluating the performance of systems or prototypes, rather than predicting it. This is due to the complex nature of ECA rule engines and rule interactions.

3 Analytical Performance Model

In order to understand how a rule engine performs, we begin by defining relevant performance metrics. These metrics, mostly stemming from [14], aim at a statistical analysis of the rule engine and are not specific to ours, thus they can be used for an objective comparison. In this paper we employ *service time* R, *throughput* μ, *CPU utilization* U and *queue length* to model the system performance.

Building a performance model for an ECA rule engine requires an understanding of its rules and their relationships. An initial, straightforward attempt to come up with an analytic performance model is to consider the entire *rule engine* as a single black box. However, this approach is not convenient for our purposes because it leads to inaccurate results.

At a finer granularity, the model could be unbundled into the *event broker* and its individual *rules* as black boxes, each with a separate queue as depicted in Figure 2. For the time being, we consider that rules are independent of each other, i.e. each event can cause the triggering of only one rule. We are aware that this condition is not the general case, and will be later relaxed. The first step consists in obtaining the service time R_{broker}, associated with the event broker, and R_i (with $i = 1, ..., n$), associated with the n deployed rules. From these service times, the average throughput μ can be obtained as follows:

$$\mu_{broker} = 1/R_{broker};\ \mu_i = 1/R_i \text{ with } i = 1, ..., n$$

To calculate the CPU utilization U, the workload λ as well as the probability for a certain event to occur p_i (and thus the probability for a certain rule to be executed) have to be specified:

$$U = U_{broker} + \sum_{i=1}^{n} U_i = \lambda/\mu_{broker} + \sum_{i=1}^{n} p_i * \frac{\lambda}{\mu_i} \text{ (where } \sum_{i=1}^{n} p_i = 1)$$

In addition to the assumption of rule independence, this model presents the problem that it does not allow loops, i.e., cases in which a condition or action statement feeds an event to the broker and thus a service is visited twice (cf. the lower dotted line in Figure 2 for rule R_n). In the worst case, under a constant load, the number of events that *revisit* these services would grow infinitely. Next, we present our approach to solve the preceding issues.

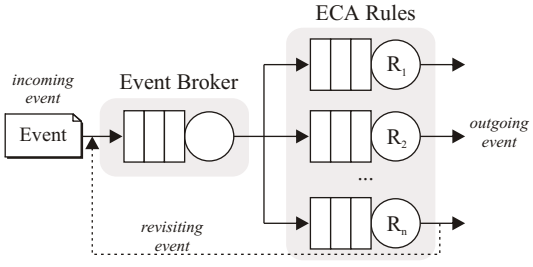


Fig. 2. Performance model with rules as black-boxes

3.1 A Simplified Model: Event Paths

The idea of the model is to consider all the possible *paths* that events may cause and assign a queue to each (see Figure 3.a). A *path* is defined as the sequence of ECA services an event goes through, possibly of different rules. The simplest paths to be identified are those initiated by events (whether they are simple or composite) that directly trigger a single rule and then exit the system. These paths must be then distinguished if, depending on the event values, the execution may conclude at the Condition Evaluation service or it may proceed until the Action Execution service. Moreover, the path must be split if it involves condition or action statements that differ in their service time under certain situations (first-time invocations, warm-up, caching, etc.). Finally, if a statement may generate another event which in turn triggers another rule, an extra path is included with the additional services. This avoids having loops between queues in the model (i.e., services are never visited twice).

In this approach, the service time R_i of each event path i starts at the time of the entrance of the event at the rule engine, during its stay at all the involved services, and stops at its final departure (this is exemplified in the interaction diagram of Figure 3.b). Measuring each event path's service time might require

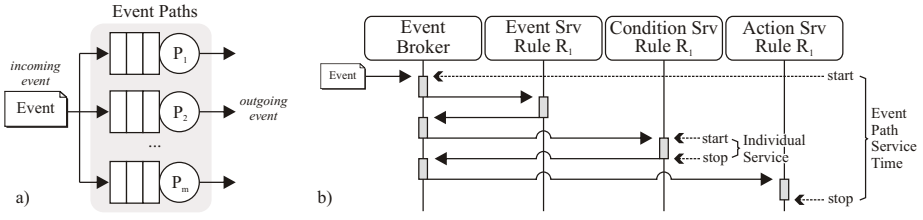


Fig. 3. a) event paths model, b) an actual event path for rule R_1

a major effort. However, and as it is shown in Section 4.2, it is acceptable to use the sum of the service times of the particular services an event path involves.

From the service time of each event path, their average throughput μ_i can be obtained, as before, from $\mu_i = 1/R_i$. To calculate the CPU utilization U for an event path model, the service times of the event paths need to be measured. The workload λ also must be defined, i.e., the probability p_i for each path to occur is needed. The CPU utilization for a model with m event paths can be calculated as follows:

$$U = \sum_{i=1}^{m} U_i = \lambda \times \sum_{i=1}^{m} \frac{p_i}{\mu_i}, \text{ with } \sum_{i=1}^{m} p_i = 1 \tag{1}$$

The simplicity of this model is counterbalanced by the fact that the more rules exist, the more complex the determination of all event paths gets. Note that this is not an operational problem but needed for the proper performance analysis. When the system must manage 100's or 1000's of rules, the number of paths can grow large, thus the usefulness of the model can be restrictive. This is not a major threat in embedded systems, since given their limited resources, the rule sets are small. Lastly, the model assumes an understanding of the rules and their relations.

3.2 Queueing Behavior

To calculate the length of a queue over time, both its service time R_i and the workload at time t, λ_t, are needed. The queue size behavior is quite intuitive: a queue grows (or shrinks) at a rate equal to the difference between the incoming events per time unit (λ_t) and the processed events per time unit (μ). Mathematically, the queue length Q is recursively constructed with the formula:

$$Q(t) = \begin{cases} 0 & \text{if } t = 0; \\ \max\{0; Q(t-1) + \lambda_t - \mu\} & \text{otherwise.} \end{cases} \tag{2}$$

3.3 Performance Evaluation Tool Set

The goal of the tool set is to support the testing phase of the software development process by helping measure and monitor different performance metrics of a

rule engine. The system supports the creation of test data, generation of events, and different time measurements. Its architecture, depicted in Figure 4 on the greyed area on the top left, is divided in two parts: one running on a server side, and the other running on the embedded, target system, as OSGi bundles.

The *Data Generator* runs on the server side. Its function is to generate the necessary test data for the performance evaluation. This includes the generation of *domain-specific data*, as well as *event properties*, i.e., metadata describing events. The user needs to describe each of the event types and provide the probability of their occurence.

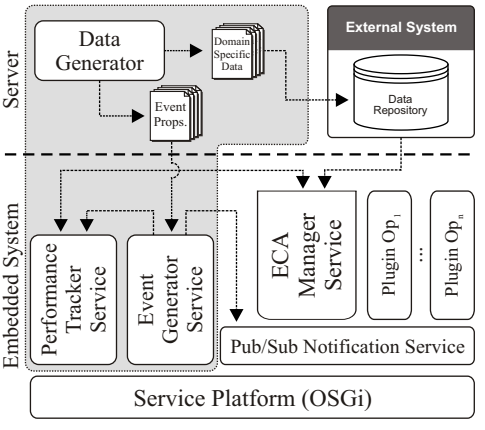


Fig. 4. Performance Evaluation Tool Set architecture

The *Event Generator* is concerned with generating and publishing at the correct points in time the necessary events the rule engine needs for a performance evaluation. The workload parameters such as which events to generate, at which time, with which frequency and the total run time, are obtained from the Data Generator. Event generation can take two forms (illustrated in Figure 5). In the *constant* mode, events are published following a constant rate ρ, during a specified interval width ϖ. The rate can be incrementally scaled by a Δ factor, which is typically used to increase the event workload and stress the system. In the *impulse* mode, events are published following a Gaussian function, which better suits real world situations of bursty event sequences. This mode must be parameterized with an interval width ϖ and a peak a. Finally, both modes can be configured to have a *gap* after each interval.

The *Performance Tracker* service is responsible for tracing service times and queue lengths. There are two mechanisms to measure service times. The first consists of adding code that starts and stops a timer in the particular component methods that are to be measured, and then send the time observations to the tracker. This mode is used when a measurement starts and stops in the same component. The second alternative consists in starting a timer, let the tracker initiate the process to be measured, and then stop it when the process

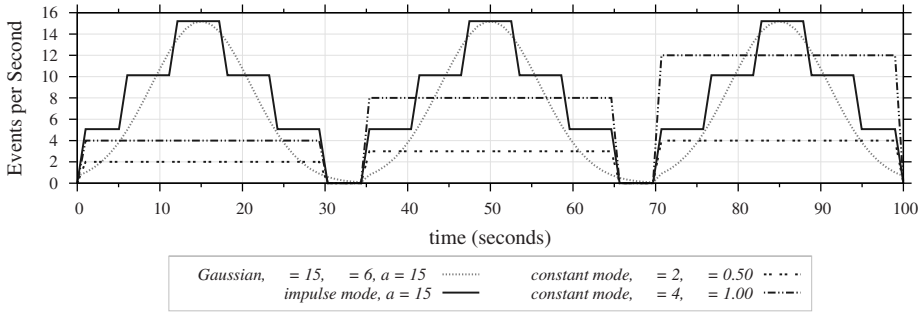


Fig. 5. Different event generation patterns with $\varpi = 30$ seconds and $gap = 5$ seconds

is finished. This mode avoids having to modify the system's source code, but includes method invocation times, context switches, etc., which don't necessarily relate to the service to be measured, and thus is used to measure service times of event paths. For both modes, the Performance Tracker attempts to measure the average service time over multiple runs, and not the time an individual event took. The time observations are stored in *buckets*. The granularity of the buckets determines the accuracy of the measurements, but also affects the memory requirement. With each new run, the buckets are flushed to disk for later analysis, which enables their reuse. The queue lengths can also be traced over time, normally in conjunction with the Event Generator in impulse mode. Since tracing queue length over time requires large amounts of memory, the traces are restarted with every new impulse.

Additionally, we have implemented a set of scripts which measure, monitor and collect other metrics, e.g. CPU, which are provided by the OS.

4 Case Study

The selected scenario is part of a supply chain environment, where a supplier ships goods stacked in pallets to a retail distribution center. These pallets have RFID tags and wireless sensors attached to them. The supplier's system sends an *Advance Shipping Notice* (ASN) to the retailer. An ASN can be seen as a document consisting of a list of *Electronic Product Codes* (EPCs) and optional constraints about the good's conditions, which imply the usage of sensors. On the other end of the supply chain, the retailer's enterprise application receives the ASN. Once the shipment arrives at the destination, a comparison between the delivered goods and the received ASN needs to be carried out.

The retailer's system is organized in 4 layers (columns in Figure 6). The inbound process is initiated at the destination when the retailer's enterprise application (1st column) receives an ASN {a}. The document is stored in an ASN Repository for later use {b}. When a truck arrives at the distribution center, a dock is assigned for unloading and the pallets are scanned while entering the warehouse. When an EPC is obtained from a scanned RFID tag on a pallet

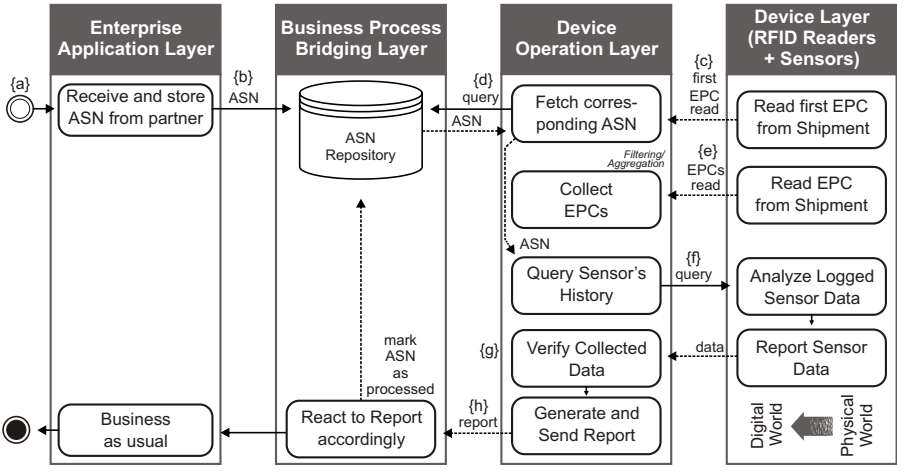


Fig. 6. Advance Shipping Notice inbound process

{c}, the corresponding ASN is fetched from the ASN Repository {d} while (in parallel) other tags are being aggregated {e}. Additionally, the Device Operation Layer can initiate sensor data validation of the attached sensor to check the logged transport conditions {f} (e.g., in case of perishable goods, temperature and humidity values are relevant, while for other goods shock, acceleration and pressure values are of interest). Based on the fetched ASN, the accuracy verification is carried out {g}. The result of the complete verification is sent to the Business Process Bridging Layer {h}, where further business logic is applied.

The rule engine implements the functionality of the Device Operation Layer, running on a Crossbow Stargate hardware platform. This is based on an Intel X-Scale processor and offers multiple network connectivity options. The system is bundled with Embedded Linux BSP, on top of which IBM's J9 Java VM for the ARM processor runs. The rule engine's services run on Oscar, an open source OSGi implementation. The sensor nodes used are Crossbow's Mica2s. We implemented a logging component in TinyOS that is able to be wirelessly queried, analyze its data and answer with the required information. For this prototype we also experimented with the Skyetek M1 Mini RFID reader attached to a Mica2's sensor board, which fed the rule engine with EPCs stored in ISO 15693 tags.

The business logic of the scenario was split into four ECA rules, which are summarized in Table 1. The rule *Incoming EPC* (R_1) listens for `EPC` events. Its condition part C_1 uses the Local Repository service to search for an ASN containing the received EPC. If there is no local cached copy, the Local Repository tries to fetch it from the remote ASN Repository. If no matching ASN is found, an `UnexpectedEPC` event is published and the action part of the rule is not executed. If a matching ASN does exist, the action part of the rule, A_1, is executed. First, the EPC is checked as 'seen'. Then, if the pallet carries a sensor node, it is queried for its

Table 1. ECA rules for the supply chain management scenario

Rule ID	Rule Name	Services	Reacts to
R_1	Incoming EPC	C_1, A_1	`EPC`
R_2	Incoming Sensor Data	E_2, A_2	`SensorData` | `MoteTimeOut`
R_3	End of Shipment	E_3, A_3	`DataCollectionReady` | `ASNTimeOut`
R_4	EPC Exception	A_4	`UnexpectedEPC`

collected sensor data. Finally, if all the expected data for the ASN has been collected, a `DataCollectionReady` event is published.

The rule *Incoming Sensor Data* (R_2) is triggered either when a wireless node sends sensor data (which occurs only when a sensor node is queried) or when a timer (which is associated with the sensor node query) times out. The action part A_2 registers this incoming sensor data in the ASN at the Local Repository. The rule *End of Shipment* (R_3) reports the results of the ASN comparison back to the Business Process Bridging Layer.

Finally, the rule *EPC Exception* (R_4) is triggered when an incoming EPC does not belong to any ASN. Its action A_4 consists in reporting the EPC back to the Business Process Bridging Layer, together with contextual information such as date, time, and dock where it was read. Note that rules R_1 and R_4 react to simple events, and thus don't require the Event Composition service, while in contrast, rules R_2 and R_3 react to composite events, in this case a disjunction with a timeout.

4.1 Identification of Event Paths

In this section we analyze the ECA rules of the ASN scenario in order to identify the event paths. The first step in modeling the ASN scenario with event paths was to write down all sequences of ECA services that an event can take. Concerning the execution of the rule *Incoming EPC*, it was very important whether the corresponding ASN already exists in the Local Repository, or it had to be fetched from the ASN Repository, since this distinction affected the service time. For that reason, event paths containing C_1 were split into two paths. The event paths were:

1. **Event Path I**: $C_1 \rightarrow A_1$
   Triggered by an `EPC` event, with no associated sensor data, where the corresponding ASN document still has unchecked EPCs besides the one recently read.
   - **I.1**: The ASN document already existed in the Local Repository.
   - **I.2**: The ASN document had to be fetched from the ASN Repository.
2. **Event Path II**: $C_1 \rightarrow A_1 \rightarrow E_3 \rightarrow A_3$
   Triggered by an `EPC` event, with no associated sensor data, where the corresponding ASN document is now completed. A `DataCollectionReady` event follows, which reports the ASN comparison results to the Business Process Bridging Layer.
   - **II.1**: The ASN document already existed in the Local Repository.
   - **II.2**: The ASN document had to be fetched from the ASN Repository.

3. **Event Path III**: $C_1 \rightarrow A_4$
   This path starts with an `EPC` event for which no ASN is found, thus is chained with an `UnexpectedEPC` event which triggers the report to the server.
4. **Event Path IV**: $C_1 \rightarrow A_1 \rightarrow E_2 \rightarrow A_2$
   This path is similar to path I, except that the `EPC` has sensor data associated. The respective sensor node is queried, to which a `SensorData` event is answered. This data is finally registered in the ASN document.
   - **IV.1**: The ASN document already existed in the Local Repository.
   - **IV.2**: The ASN document had to be fetched from the ASN Repository.
5. **Event Path V**: $C_1 \rightarrow A_1 \rightarrow E_2 \rightarrow A_2 \rightarrow E_3 \rightarrow A_3$
   This event path is similar to event path IV, aside from the fact that the ASN does not have unchecked EPCs anymore, hence the ASN is reported back to the ASN Repository.
   - **V.1**: The ASN document already existed in the Local Repository.
   - **V.2**: The ASN document had to be fetched from the ASN Repository.

4.2 Measured Service Times

This subsection summarizes the results on service times. For these experiments, the Event Generator was used in *constant* mode, with $\rho = 20$ events / minute. This low frequency ensured that events were not influenced by each other. The experiments ran for 80 minutes, the first 30 being ignored as warm-up phase.

The service times of individual services and the event paths are summarized in Table 2 and 3, respectively. We compare both approaches to measure service times in Figure 7. The absolute (i.e., concrete) min., max. and avg. values for both the sum of individual service times and event paths is shown in 7(a). Figure 7(b) contrasts the relative service time difference between the event path average service time (middle greyed area, 100%) against the min., max. and avg. sum of individual service times. The average deviation for these event paths was 9.25%.

4.3 Queueing Behavior

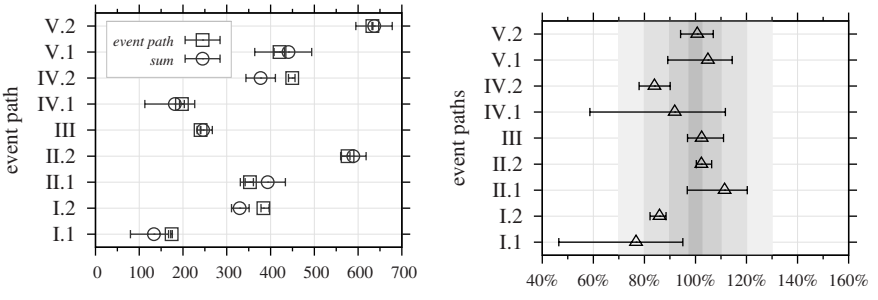
In this section we present the results about the queueing behavior. To study this, it is necessary to have event bursts such that queues form, followed by periods without events. For this purpose, the Event Generator was used in impulse mode, with intervals of $\varpi = 30$ seconds and a $gap = 4.5$ minutes (where no events were sent). During the peak, the Event Generator published events with a peak

Table 2. Service times for the Event, Condition and Action services

Id	Rule	Note	Event	Condition	Action
R_1	Incoming EPC	fetch ASN	—	241.75ms	87.78ms
		do not fetch ASN	—	29.22ms	104.41ms
		no ASN available	—	192.05ms	—
R_2	Incoming Sensor Data		6.36ms	—	41.47ms
R_3	End of Shipment		6.50ms	—	253.15ms
R_4	Unknown EPC		—	—	53.83ms

Table 3. Service times for event paths

Event Path	Description	Service time
I	$C_1 \rightarrow A_1$	
I.1	do not fetch the ASN	174.15ms
I.2	fetch the ASN	383.59ms
II	$C_1 \rightarrow A_1 \rightarrow E_3 \rightarrow A_3$	
II.1	do not fetch the ASN	352.93ms
II.2	fetch the ASN	575.87ms
III	$C_1 \rightarrow A_4$	
	fetch the ASN	239.99ms
IV	$C_1 \rightarrow A_1 \rightarrow E_2 \rightarrow A_2$	
IV.1	do not fetch the ASN	197.50ms
IV.2	fetch the ASN	449.46ms
V	$C_1 \rightarrow A_1 \rightarrow E_2 \rightarrow A_2 \rightarrow E_3 \rightarrow A_3$	
V.1	do not fetch the ASN	420.61ms
V.2	fetch the ASN	632.64ms



(a) Absolute service times (ms) (b) Relative service time difference (%)

Fig. 7. Service time measurements: absolute values (a) and relative difference (b)

$a = 15$ events/s. Once this 5-minute process finished, it was repeated again. For the measurement of the queue length, the Performance Tracker service was used. Each time the Event Generator started sending events for the 30 seconds period, it also started the queue trace at the tracker, hence queue lengths of all observed queues were recorded at every second. The queues to be observed were selected by the Event Generator in the initialization phase.

Now we present the queueing behavior of the Condition Evaluation service for the rule R_1, i.e., C_1. For this purpose, a test was carried out where the ASN Repository stores 200 ASNs, each containing one EPC with no sensor data. The queue length predictions were based on the service times for C_1 from Table 2. The Condition Evaluation service does not have its own queue. Indeed, events were queued in the Event Broker service, waiting for being serviced by C_1. The Event Broker can be seen as the queue of the Condition Service because its service time is negligible. The Event Generator split the ϖ interval in five stages. The event arrival rate at the Broker varied at these stages according to the workload λ_t as defined in Equation 3.

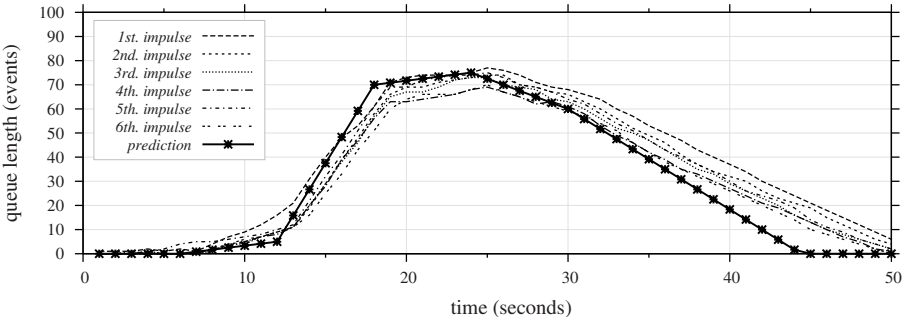


Fig. 8. Measured and predicted queue length for service C_1

$$\lambda_t = \begin{cases} 1\frac{2}{3} \text{ events/s if } 1 \leq t \leq 6 \text{ or } 25 \leq t \leq 30 \\ 5 \text{ events/s if } 7 \leq t \leq 12 \text{ or } 19 \leq t \leq 24 \\ 15 \text{ events/s if } 13 \leq t \leq 18 \end{cases} \quad (3)$$

From C_1's service time (where the ASN must be fetched), we obtained $\mu = 1/0.24175$ events/s $= 4.13$ events/s. By using Equation 2, the queue length can be calculated. For the measurement, the Event Generator ran for a period of 30 minutes, thus 6 queue traces were obtained. The comparison between the predicted values and each of the 6 runs, presented in Figure 8, shows that the calculations were considerably accurate.

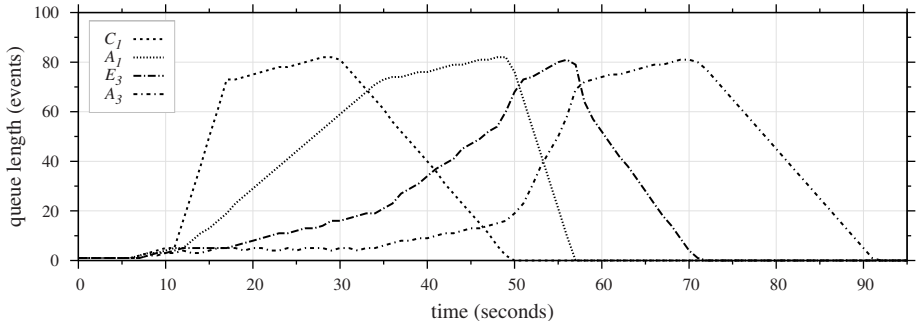
Next, we discuss the more general case where multiple interacting services operated sequentially on incoming events. For space reasons, we consider here only the event path II.1, which involved the sequence of services $C_1 \rightarrow A_1 \rightarrow E_3 \rightarrow A_3$. The ECA rule engine was designed with a single real queue for all the incoming events. The resulting behavior is difficult to calculate analytically. Therefore, we wrote a small script that simulated it. On the measurements side, the Event Generator was configured to run over a 40 minutes period. We compare the simulated and empirical measurements in Figure 9 (a) and (b), respectively.

These two plots are considerably similar, except at $t \geq t_d$. This difference revealed a relation between the rules R_1 and R_3 which was unforeseen at the time we designed the queue length simulator. The issue arises when an EPC event for a particular ASN must wait too long in the Broker queue. When this wait exceeds a predefined timer, an `ASNTimeOut` event is triggered, which sends the (incomplete) ASN document back to the repository and thus has to be re-fetched. This also explains the higher amount of events on the queues of R_3.

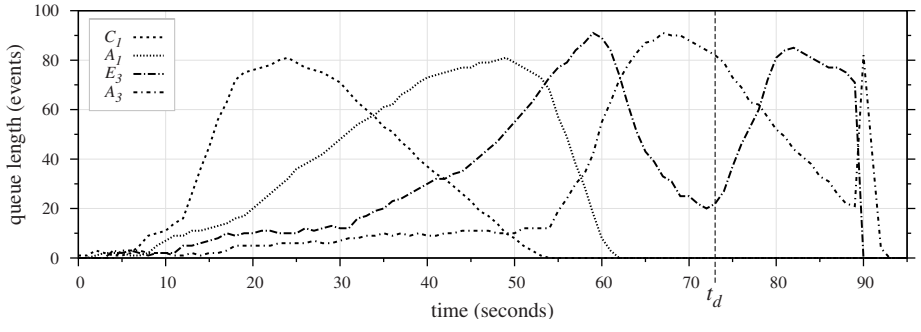
4.4 CPU Utilization

We now present the results on CPU utilization. In this supply chain scenario, the workload is distributed across 5 EPC types, to which we assigned a percentage in Table 4(a). This selection covered the four rules of the scenario.

In order to calculate the CPU utilization using Equation 1, however, the probabilities of each *event path* (and not the EPC types) are needed. For this purpose, we fixed the number of EPCs per ASN for this experiments to 100 EPCs. With



(a) Prediction of queue length by means of simulation



(b) Measured queue length

Fig. 9. Predicted (a) and measured (b) queue lengths for event path II.1

this consideration, the probabilities of each event path were shaped. First, the 25% assigned to Checked EPCs were mapped to event path I.1, I.2, II.1 and II.2, because a Checked EPC always causes the triggering of R_1, followed (sometimes) by R_3 if the ASN document is completed. Second, the 10% of Unexpected EPCs were mapped to event path III. Finally, the remaining EPC types (which accounts for 65%) were mapped to event paths IV.1, IV.2, V.1 and V.2. These probabilities are shown in Table 4(b).

The average service time can be calculated using the information from Table 3 and the formula: $\mu = \sum_{i \in paths} p_i * \mu_i = 222.63$ ms, with $\sum_{i \in paths} p_i = 1$. The CPU utilization, in turn, is calculated from: $U_t = \lambda_t / \mu$. The CPU utilization was monitored using the standard Linux `top` command; a script piped it to a file for later analysis. Both for the calculations and measurements, the average number of incoming events started with $\rho = 40$ events/minute, and it was incremented by a factor $\Delta = 0.5$ every $\varpi = 10$ minutes. The prediction and the measurement were executed over a total of 65 minutes.

In Figure 10 we show a plot of the published events over time (right y axis), together with the measured and predicted results (left y axis). It is easy to notice that the utilization remained constant for 10 minutes and then increased slightly. However, the measured utilization drifted significantly from the predicted one.

Table 4. Settings for CPU utilization prediction and measurement

(a) Workload characterized according to event types

EPC type:	Assigned %
Checked EPC	25%
Checked EPC with sensor data	25%
Checked EPC with missing sensor data	20%
Checked EPC with infringing sensor data	20%
Unexpected EPC	10%

(b) Event path probabilities

Event Path	%
I.1	⇐ 23%
I.2	⇐ 1%
II.1	⇐ 0%
II.2	⇐ 1%
III	⇐ 10%
IV.1	⇐ 59%
IV.2	⇐ 3%
V.1	⇐ 0%
V.1	⇐ 3%

At higher λ rates, the difference was about 15%, which turned the prediction unacceptable. The reason for this was that the ECA rules executed several actions that were I/O bound, particularly blocking invocations with large roundtrips. For instance, the fetching of ASN objects (i.e., XML documents) was implemented by an RMI call which took about 160ms. Equation 1, though, relies on the assumption that the CPU is kept busy all the time. Given this consideration, we adjusted the service times by subtracting pure I/O operation times associated to each event path, and recalculated the average throughput. As a result, the (adjusted) CPU utilization prediction, also plotted in Figure 10, resulted a reasonable approximation of the observed one.

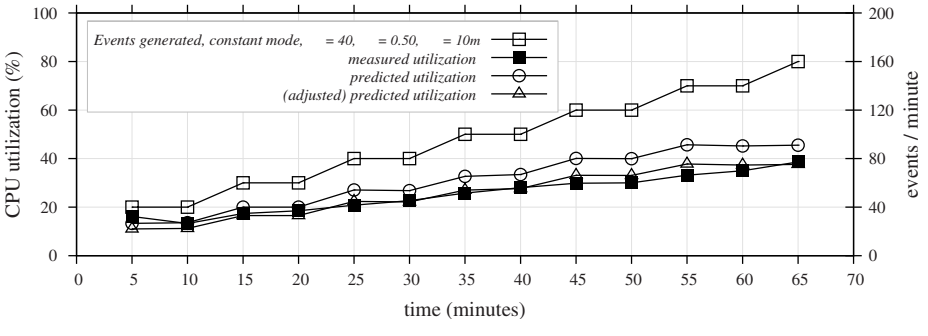


Fig. 10. Measured and predicted CPU utilization using the event path model

5 Conclusions and Future Work

We presented a performance evaluation of an ECA rule engine on an embedded device. This area has not been well explored because it deals with two complex domains: the resource constraints of embedded devices and the complex reactive nature of ECA systems.

The model we developed eliminates, to a certain extent, the problems of rule independence and revisiting events. The proposed solution, based on identifying

event paths, has shown to be reasonably accurate in predicting performance. Furthermore, the presented work helped understand the entire system more deeply and enhance it in different ways. Queueing behavior analysis exposed timing dependencies between rules that were not evident before.

The model's simplicity can be offset by the effort required to find manually all event paths and obtain their probabilities. It might be useful to integrate a tool that, by statically analyzing the ECA rules, automatically identifies the paths that the events may take. By dynamically tracing incoming events, the relevant paths could be identified and their probability determined by the frequency with which the path was traced. Finally, we are working on developing a comprehensive methodology for performance evaluation of ECA rule engines, independently of the underlying platform under test. This requires the application of the steps described in this paper to further projects to confirm the validity of the method based on event paths.

References

1. Bornhövd, C., Lin, T., Haller, S., Schaper, J.: Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In: 30th VLDB, Toronto, Canada (2004)
2. Guerrero, P.E., Sachs, K., Cilia, M., Bornhövd, C., Buchmann, A.: Pushing Business Data Processing Towards the Periphery. In: 23rd ICDE, Istanbul, Turkey, pp. 1485–1486. IEEE Computer Society, Los Alamitos (2007)
3. Cilia, M.: An Active Functionality Service for Open Distributed Heterogeneous Environments. PhD thesis, Dept. of Computer Science, Technische Universität Darmstadt, Germany (August 2002)
4. Mühl, G.: Large-Scale Content-Based Publish/Subscribe Systems. PhD thesis, Dept. of Computer Science, Technische Universität Darmstadt, Germany (September 2002)
5. Widom, J., Ceri, S. (eds.): Active Database Systems: Triggers and Rules for Advanced Database Processing. Morgan Kaufmann Series in Data Management Systems, vol. 77. Morgan Kaufmann, San Francisco (1996)
6. Paton, N.W. (ed.): Active Rules in Database Systems. Monographs in Computer Science. Springer, New York (1999)
7. Dittrich, K.R., Gatziu, S., Geppert, A.: The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In: Sellis, T.K. (ed.) RIDS 1995. LNCS, vol. 985, pp. 3–20. Springer, Heidelberg (1995)
8. Cilia, M.: Active Database Management Systems. In: Rivero, L.C., Doorn, J.H., Ferraggine, V.E. (eds.) Encyclopedia of Database Technologies and Applications, pp. 1–4. Idea Group (2005)
9. Zimmermann, J., Buchmann, A.P.: Benchmarking Active Database Systems: A Requirements Analysis. In: OOPSLA 1995 Workshop on Object Database Behavior, Benchmarks, and Performance, pp. 1–5 (1995)
10. Geppert, A., Gatziu, S., Dittrich, K.: A Designer's Benchmark for Active Database Management Systems: 007 Meets the BEAST. In: Sellis, T.K. (ed.) RIDS 1995. LNCS, vol. 985, pp. 309–326. Springer, Heidelberg (1995)
11. Geppert, A., Berndtsson, M., Lieuwen, D., Zimmermann, J.: Performance Evaluation of Active Database Management Systems Using the BEAST Benchmark. TR IFI-96.01, Dept. of Computer Science, University of Zurich (1996)

12. Konana, P., Mok, A.K., Lee, C.G., Woo, H., Liu, G.: Implementation and Performance Evaluation of a Real-Time E-Brokerage System. In: 21st IEEE Real-Time Systems Symposium, pp. 109–118 (2000)
13. Baralis, E., Bianco, A.: Performance Evaluation of Rule Execution Semantics in Active Databases. In: 13th ICDE, pp. 365–374. IEEE Computer Society, Los Alamitos (1997)
14. Menasce, D.A., Almeida, V.A.F.: Capacity Planning for Web Services: Metrics, Models, and Methods. Prentice Hall PTR, NJ (2001)