Use of Extended Adaptive Decision Tables on Reconfigurable Operating Systems

S. M. Martin, G. E. De Luca, and N. B. Casas

***Abstract*— Throughout the last decades, many reconfigurable operating systems have been developed in order to let users and programmers make decisions upon the configuration of some of the innermost kernel's aspects. These decisions, however, require an advanced level of skill in order to obtain some performance advantage. Furthermore, they can be detrimental to the system's performance if they are not careful taken. Letting the kernel itself do the decision-making upon the implementation of its aspects is a safe and powerful way to manage re-configurability that does not require interaction with the user nor the need of advanced skills to take advantage of. In this article, we propose the usage of Extended Adaptive Decision Tables as a mean for the kernel to achieve the capability of taking intelligent decisions based solely on the users' process creation behavior.**

***Keywords*— Decision Tables, Operating Systems, Adaptive Device**

I. Introduction

The operating systems have been developed and used for decades to abstract the complexity of the underlying hardware for both end-users, and application programmers. Through them, the computer and its components can be seen as administrable resources that users and their applications can request and use. The list of resources that are administrated by the system kernel includes, but is not limited to: processor usage, memory distribution, permission for input/output operations on actual or virtual devices, among others.

The first operating systems, such as MS-DOS, only allowed the execution of a single process at a time. For that reason, its kernel's only functions were limited to booting, providing a command line, and some simple services. The executing process would take the complete control over the processor, the memory, and the I/O devices, and the system kernel had no administrative responsibilities whatsoever.

With the development of the more advanced UNIX-based operating systems that allowed the execution of more than one process at a time, and the logging of multiple users concurrently, new challenges arose. The issue of which processes or users should have more processor usage time, or greater free memory chunks available, was addressed differently among the different kernel developers. Now, the diversity of resource administration policies among different operating systems provides users and companies a range of problem-specific products to serve their own business necessities.

S. M. Martin, G. E. De Luca, N. B. Casas, Departamento de Ingeniería e Investigaciones Tecnológicas, Universidad Nacional de La Matanza Buenos Aires, Argentina. {smartin, gdeluca, ncasas}@unlam.edu.ar

Most of the now available public operating systems are non re-configurable. This means that their resource administration policies are built-in and cannot be changed by the users.

We will use the nomenclature used in our previous work [1] in which the processor usage, the memory distribution or any other resource management are said to be *aspects* of the operating system. Each one of these aspects can be administrated using a policy or *mode*. An operating system is said to be *reconfigurable* when one or more of its aspects can change their mode during runtime without rebooting.

Although almost all of the commercial and home requirements for an operating system can be satisfied with a non-reconfigurable conventional kernel, there is a potential for the adaptable capabilities of a reconfigurable kernel.

Some examples of reconfigurable operating systems, such as Kea [2] and Synthetix [3], have shown better results than their commercial counterparts on tests driven under diverse and changing execution conditions and process behaviors. Others, such as SPIN [4], provided an interface for extensibility where the programmer himself could develop new modes for the kernel's aspects.

All of the reconfigurable operating systems available – ready to use, source code, or just in academic bibliography – depend upon the user/programmer to decide both which changes make to the kernel configuration and when to make them. In the majority of the cases, this is achieved by providing an object-model based kernel-process interface [5] that presents the functionality for a certain service, and allows the programmer to define which object instance gets to execute them.

In spite of the great flexibility and adaptive potential that can be obtained with the interface approach, its limitations can be often enough to prevent users, programmers, or even kernel designers, from using it.

At first, it takes an application programmer to know at least something about the kernel's intricacies in order to obtain any advantage. Some programmers may even have to investigate about them before knowing where and what to change.

On the other hand, legacy, standard, or reused programs wouldn't have the opportunity to harness its benefits. They might have to be re-engineered before being able to use the interfaces properly. End-users with programs of their own wouldn't stand a chance on harvesting the potential of the underlying re-configurability.

In this article, we present a different viewpoint for kernel reconfiguration. Since letting the users have the decision-taking responsibility offers great potential for performance improvement, but lacks the portability and demands high skills, we thought that the kernel itself may be able to take charge of the issue. The three main arguments supporting this idea are:

First, the kernel developers already have all the skills and knowledge about its aspects and modes' inner complexities. They can design better and faster mechanisms for mode exchanging so no skills or specific knowledge should be demanded to their users and programmers whatsoever.

Second, all the applications to be run can be kernel-agnostic and still harness the benefits of reconfiguration.

Lastly, the kernel can take decisions upon several more indicators than a programmer could. Some of them may be inaccessible for a process at runtime, and some other may be too complex or too kernel-specific for a programmer to take into account.

We will use Extended Adaptive Decision Tables – from now on, abbreviated as EADTs – as the device that will allow us, as designers, give the kernel the mechanism for decision making based on the behavior of its users.

All the analysis and examples presented in this article were conducted on SODIUM[1], a project for an academic reconfigurable operating system [6] [7] and, more specifically, its reconfigurable process scheduler aspect.

The rest of this article is organized as follows: Section 2 introduces the reconfigurable kernel design methodology used to analyze each aspect. Section 3 introduces the SODIUM's reconfigurable process administration aspect in more detail. In section 4, that aspect will be analyzed in order to obtain the decision taking criteria necessary for the construction of the initial decision table that will be presented in Section 5. In section 6 explain what adaptive elements were used in order to obtain the adaptive decision table. In section 7, we analyze the extended mechanism that allows for multi-criteria decision taking in order to generate the final extended adaptive decision table. Finally, section 8 discusses the future work and tests to be conducted, and the conclusions of this investigation.

II. Reconfigurable design methodology

The SODIUM project was started back in 2005 with the aim to give the operating system class' alumni the opportunity, not only to grasp the theory concepts, but also to let them involucrate actively in the development of a kernel. At the end of each year, all the practices were tested, and the best ones were integrated into the kernel for the next year's alumni to use.

One of the first dilemmas of this methodology emerged when, predictably, many different modes were programmed for the same particular aspect. For example, having a basic fixed partition memory administrator as a base, a practice asked to develop another one based on paging, forced the student to replace the existing one. This forcibly implicated a loss in didactic value since, even though the paging approach was, in overall, better, the former mode was still useful for teaching purposes.

Given this situation, the professorship agreed on implementing a mechanism by which SODIUM could handle multiple modes for any aspect, and that those modes could be exchanged while still on runtime. The efforts on developing a design pattern for this mechanism ended up in the methodology proposed in [1].

In that methodology, four steps had to be conducted upon each aspect to be designed for re-configurability:

1. The current situation must be analyzed for each possible transition between modes – if any –, the kernel mechanism that should be developed, and the timing level.
2. All the currently available modes should be enumerated along with a graph showing new ones, in order to uncover possible missing transitions.
3. Design the reconfigurable aspect indicating implementation details, timing levels, and element to be verified for possible data loss for each transition.
4. Design, document, and publish the kernel/user interfaces for the reconfiguration mechanism.

Although this methodology is useful to design reconfigurable schemes for existing aspects and modes, it does not contemplate the decision-taking process. In fact, it ends at step 4, where it indicates that a kernel/user interface should be designed. In our proposal, we seek to vary this methodology to let the kernel reconfigure itself. In order to do so, we shall replace the former step with a new one:

4. Establish the criteria and events determining the need for mode changes, and the mechanisms to allow them.

The timing levels are also important because they indicate which conditions within the system determine whether a transition can be executed or not. Four timing levels are defined. Level 1, when the transition can be only be executed by recompiling the kernel; level 2, when the transition can only be executed by rebooting the system; level 3, when the transition can be executed in runtime, but globally for all processes; and level 4, when the transition can be executed in runtime, and in particular for each process.

In the next section, we will use the presented methodology in order to analyze one of the main reconfigurable aspects of SODIUM's kernel.

III. SODIUM's Reconfigurable Process Scheduler

Even though this article presents a general proposal for kernel decision-taking on re-configurability using EADTs, it is necessary and much easier to explain its implementation steps by using an existing reconfigurable aspect as base.

No other reconfigurable aspect has been more refined and researched upon in SODIUM than the process scheduler. It counts with 6 different modes available: Round-Robin (RR), Round-Robin with Priority Queues (RRPR), Round-Robin with Variable Quantum (RRQV), First-Come-First-Serve (FCFS), Shortest-Finishing-Job-First (SJFS), and Best-Time-Of-Service (BTS). Many of these modes specification are

[1] Sistema Operativo del Departamento de Ingeniería de la Universidad Nacional de La Matanza. Web: http://www.so-unlam.com.ar/

standards and can be found in specialized operating system's books. We used a general specification found in [8] as a base for almost all of them.

This aspect has the particularity that all of its 6 modes can transition to all of the others without any limitation. This means that it counts with 30 different transitions to evaluate. Also, the timing level for all transitions is 3. This is because any changes in the process scheduler will forcibly affect all the process at the same time. No changes can be made individually for any process.

Since straightforwardly programming 30 different functions –one for each transition– could result in a bloated and difficult to understand kernel code, we decided to analyze, at depth, which actions were to be shared between the different transitions' procedures. We identified 8 common actions that could be normalized and reutilized between all the procedures. Briefly, these actions are: *setbasealg(alg)*, to set the basic scheduling algorithm; *initPriorities()*, to initialize priorities for RRPR; *initQTime()* to initialize quantum times for RRQV; *setNonPreemptive()* and *setPreemptive()* to set a preemptive or non-preemptive scheduler behavior, respectively; *queueMode(mode)* to set a unique or level based ready process queue; and *quantumMode(mode)*, to set a unique or priority based quantum evaluation. In Table 3.1, different groups of actions are set and identified with an alphabet letter codification.

		a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z
Actions	setbasealg(BTS)					x					x					x					x							
	setbasealg(RR)																							x	x	x	x	x
	initPriorities()	x	x															x	x						x	x		
	initQTime()		x					x											x							x		
	initReorder(CTIME)			x					x					x									x				x	
	initReorder(PTIME)				x					x					x					x								x
	setNonPreemptive()			x	x					x	x			x	x												x	x
	setPreemptive()																x	x	x		x	x						
	queueMode(UN)						x	x	x	x	x																	
	queueMode(PR)	x											x					x							x			
	quantumMode(UN)											x	x	x	x	x												
	quantumMode(PR)		x					x											x							x		
Codification		a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	u	v	w	x	y	z

Table 3.1 – Set of actions and their codification for mode transitions on SODUIUM's process scheduler

These groups of actions represent functions that allow the execution of all the analyzed transitions. The map of actions per transition can be seen in Table 3.2. The *'0'* action means that no procedure should be executed.

In Table 3.2, the SODIUM kernel can query which actions to execute – all of them are commutative – in order to change the scheduling mode to another while still on runtime. This is a key step for an autonomously reconfigurable kernel, and must be designed by the systems developers. This table will be used later again in Section 7, when new rules of transition will have to be created to contemplate new executing conditions.

One of the visible limitations of this approach is that it is not extensible. All the transactions and groups of actions must be determined before the programming of the kernel. Any mechanism that would allow a programmer to enable new modes would be so complex that it would go against its main benefits.

		Next Mode					
		RR	PR	QV	FC	SJ	BT
Current Mode	RR	0	a	b	c	d	e
	PR	f	0	g	h	i	j
	QV	k	l	0	m	n	ñ
	FC	o	p	q	0	r	s
	SJ	t	p	q	u	0	s
	BT	v	w	x	y	z	0

Table 3.2 – Mode transitions and their set of actions

Although this transition-actions relation allow the kernel to execute transitions by itself, it is still not enough to let it take decisions. Having this relation between transitions and actions to be performed is the first step in order to generate a conventional decision table –from now on, abbreviated as cTD–. The cTD will hold the first trivial condition-action mechanism that will indicate the kernel when and what transitions execute.

IV. Decision-Taking Criteria

In order to let a transition-mode-actions table, as the one obtained before, allow the kernel to have decision-taking capabilities, we need to define two new key elements: conditions, which will indicate which rules –as a generalization of what a transition is, in the context of this article– must execute in a given moment; and events, that indicate when the conditions must be evaluated.

Events by themselves can be also considered as conditions, for if they do not trigger, no rules associated to them will be executed. However, from an operating system's view, the distinction is important. Events are codified as trigger functions set in different parts of the kernel while conditions are a part of the cTD to be developed. In the case of our process scheduler, event triggers will be set every time a process is created, killed, interrupted, or released. This includes hardware/software interruptions, syscalls, I/O requests and responses, and process intercommunication routines.

Defining conditions as such will be a little more difficult. In order to define in which executing scenarios we will have to evaluate three different edges of the process scheduling: scheduling metrics, application categories, and algorithm-metrics relations.

A. *Scheduling metrics*

SODIUM's process scheduler counts with a set of 7 metrics to evaluate and report the performance of each scheduling algorithm. Most of them are based on the metrics list present in [8]. Here is a brief enumeration and explanation for each one:

- Processor Usage ($\%_{cpu}$) indicates the ratio between the time the processor spends actually executing processes and the time that it spends idle or in overhead costly procedures.
- Throughput ($\#_f$) indicates the amount of processes finished given a finite differential of time. SODIUM updates this metric every 10 minutes.
- Turnaround Time (t_{cv}) indicates the time that took a process to completely executes, from when it is created until it is terminated. It includes the time spent waiting for enough free memory, to be executed, and I/O operations.
- Waiting Time (t_w) indicates the total waiting time of a process during all its execution from the moment it is created. I/O operations or syscalls are not included in this metric since they represent actual requested operations from the process.
- Response Time ($\overline{t_r}$) indicates the average time, for each process, after which, the first response, such as writing a character on the screen, is produced after a user request.
- Effective Time of Service (t_s) indicates the sum of time that the process has spent in actual usage of the processor.
- Overhead (O_v) indicates the time that the processor spends executing system maintenance or managing routines.

B. *Application Categories*

Since it is not possible to know completely what actions and services will an application request before it is completely executed, the only way to estimate its future behavior is to profile it into well-known categories. We focused that profiling using a technique presented in [9] based on frequencies of system calls, and maintaining a per-process profiling information structure such as the ones specified for the Solaris operating system in [10].

Executing processes fall, after a short period of profiling, into one of the following application categories defined by us: Interaction Intensive Applications (II), such as games or command-line consoles; Multimedia Applications (M), such as video and audio editing tools; I/O Intensive Applications (ES), such as DVD burners or data transfer programs; Internet Applications (WEB), such as web browsers or network programs; Processing Intensive Applications (P), such as compilers or scientific programs; and System Applications (S), such as services or maintenance programs.

After conducting several tests on sample programs that were successfully profiled, we could establish which metrics are more important for each application category. In the Table 4.1 we show the most important metrics for each category that resulted from the tests.

Categories	Metrics
Interaction Intensive Applications (II)	Response Time Waiting Time
Multimedia Applications (M)	Waiting Time Processor Usage
I/O Intensive Applications (ES)	Throughput Response Time
Internet Applications (WEB)	Effective Time of Service Response Time
Processing Intensive Applications (P)	Turnaround Time Throughput
System Applications (S)	Overhead

Table 4.1 – High priority metrics for each category

For simplicity reasons, all the other metrics that are not considered as high priority will be considered as indifferent for that given category.

C. *Algorithm-Metrics Relations*

Since we now count with the possibility of profiling a process into a category, keep scheduling metrics up to date, and know which metrics favor each category, we only need to determine which scheduling algorithms (modes) improve those metrics.

We found some of these algorithm-metric relations already documented [8] in the bibliography. However, we conducted additional tests to confirm them, and also figure out those that were lacking. From these tests, we obtained the results shown in the Table 4.2. In this table, the beneficial relations in which the algorithm improves the measures from each metric are marked with a (+); the neutral relations in which the algorithm doesn't affect the metric are marked with a (0); the negative relations are marked with a (-), and the (x) marks indicate that the algorithm is extremely negative to the metric.

	Metric						
Algorithm	%cpu	$\#_f$	t_{cv}	t_w	t_r	t_s	o_v
RR	+	0	-	0	0	0	0
RRPR	+	0	0	+	0	x	0
RRQV	0	0	0	0	+	x	0
FCFS	-	0	+	x	x	x	+
SJFS	-	+	+	x	x	x	-
BTS	-	0	-	-	0	+	-

Table 4.2 – Relations between metrics and algorithms

We can now combine Tables 4.1 and 4.2 in order to obtain an algorithm-category rating in which the score will indicate how much each algorithm benefits/handicaps each application category. The amount of (+) marks per each high priority metric that the algorithm beneficiates will score 1 positive point for that category; (-) will rest 1 point; (0) will produce no effect; and (x) marks will completely disqualify the algorithm. The results from the combination are presented in the Table 4.3.

	Algorithm					
Category	RR	RRPR	RRQV	FCFS	SJFS	BTS
(II)	0	1	1	X	X	0
(M)	1	2	0	X	X	0
(ES)	0	0	1	X	X	0
(WEB)	0	X	X	X	X	1
(P)	-1	0	0	1	2	-1
(S)	0	0	0	1	-1	-1

Table 4.3 – Rating relation table between algorithms and application categories.

V. Conventional Decision Table

The rating information obtained in the Table 4.3 is enough to decide which scheduling algorithm to use when the processes in execution pertain to the same application category. However, this scenario doesn't cover all the execution possibilities.

It could happen that only one process is in the ready queue. This case may be important for batch processing systems. Using a overhead costly algorithm in these cases is not convenient. Therefore, we can establish that in case of mono-processing (*mono*), the algorithm used will be FCFS.

Also, a more common scenario is that when different processes of different categories try to execute concurrently. In this case, using the rating from Table 4.3 can be misleading, because we are using simple heuristic and empiric information on complex cases. In fact, there are 720 different combinational scenarios of mixed categories. This complexity cannot be addressed a priori, by analyzing each case in particular. This case of multi-category (multi) will be resolved using an adaptive decision table –from now on, abbreviated as ADT– to be developed in the next section.

Rules			1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Conditions	Current Mode		RR	RR	RR	RR	RR	PR	PR	PR	PR	PR	QV	QV	QV	QV	QV	FC	FC	FC	FC	FC	SJ	SJ	SJ	SJ	SJ	BT	BT	BT	BT	BT
	Current Scenario	Mono			x					x					x											x					x	
		II		x					x											x					x					x		
		M	x											x					x					x					x			
		ES		x					x											x					x					x		
		WEB					x					x					x					x					x					
		P				x					x					x					x											x
		S			x					x					x											x					x	
		Multi						x					x					x					x					x				
Actions		func()	a	b	c	d	e	f	g	h	i	j	k	l	m	n	ñ	o	p	q	r	s	t	p	q	u	s	v	w	x	y	z

Table 5.1 – Conventional Decision Table for SODIUM Process Scheduler

By now, we can contemplate the general case of multiple categories with the most generally acceptable of the scheduling algorithms: RR.

The *mono, multi,* and the pure categories processes can be interpreted as the conditions that, without combining with each other, determine the whole universe of possible execution scenarios. Also, we know which algorithm to use for each condition, we can establish the transitions –and their actions– to execute in each case by combining Tables 4.3 and 3.2.

We now have all the elements to elaborate the first cDT for the SODIUM kernel to decide which algorithm use at each moment. It will consist in a set of 30 rules combining conditions –current mode and current scenario– and their transition actions. The cDT for the SODIUM process scheduler is shown as the Table 5.1.

VI. Adaptive Decision Table

A. *Normalized Base cDT*

The condition evaluation in the cDT of the Table 5.1 obeys to that of an inclusive OR. This means that, it takes only one condition to be true in order to execute a given rule. For example, rule 2 will execute if a *Mono* scenario is detected and also if it detects an *ES* scenario. However, the formal definition of the cDT, that we must use as a base for developing the more complex ADTs in order to contemplate all the possible scenarios from the generalized *Multi*, requires the conditions to be evaluated with an AND logic. For this, it will be necessary to add duplicated rules that contemplate one of the conditions that will be eliminated from the original one. Also, we need to eliminate *mono* scenario corresponding rules because they can be directly programmed into the scheduler; and those corresponding the *multi* scenario, because the ADT will allow us to contemplate all the particular rules of combining categories scenarios.

Another modification to the original cDT is the addition of the not mark (-) indicating that that condition must be false in order to execute that rule. The resulting fixed cDT is shown in the Table 6.1 (some of the rules have been bypassed in the representation for format purposes).

	1	2	3	4	5	6	7	8	…	24	25	26	27	28	29	30
Current Mode	RR	RR	RR	RR	PR	PR	PR	QV	…	PR	QV	FC	SJ	SJ	BT	BT
II	-	x	-	-	x	-	-	-	…	-	-	-	-	-	-	-
M	x	-	-	-	-	-	-	x	…	-	-	-	-	-	-	-
ES	-	-	-	-	-	-	-	-	…	-	-	x	x	-	x	-
WEB	-	-	-	x	-	-	x	-	…	-	-	-	-	-	-	-
P	-	-	x	-	-	x	-	-	…	-	-	-	-	-	-	-
S	-	-	-	-	-	-	-	-	…	x	x	-	-	x	-	x
func()	a	b	d	e	g	i	j	l	…	h	m	q	q	u	x	y

Table 6.1 – Normalized cDT

simultaneously, we will have to add adaptive mechanisms to our static cDT in order to turn it into a ADT. ADTs formal definition [11] [12] requires the specification of a normalized base cDT such as the one in Table 6.1, and also the addition of adaptive functions to perform rule query, elimination, and insertion actions upon it.

In this investigation, we used a simple adaptive mechanism that consists in the usage of two different adaptive functions. One is used to verify the existence of rules contemplating the current execution scenario. The other is used to add a new rule in case that no such rules were detected.

These adaptive functions called Ad1 and Ad2, execute after and before their calling rules, respectively. Ad1 only sets the value of the *state* variable to D (determined) when an existing rule can manage the current state of conditions. When no rule can handle the current conditions, a new rule (31) is in charge to set the *state* variable to ND (non-determined). This variable is set back by the execution of Ad1, in case that a rule was found. On the next step, if the state is equal to ND, another new rule (32) executes the Ad2 adaptive function.

The Ad2 adaptive function is in charge to add 6 new rules. These new rules will transition to the current mode from any other mode given the current conditions. The result for this is that, whenever in the future, when the same conditions repeat, they will transition to the mode in which those conditions were found initially. The rationale behind this is that, when a new set of conditions is found, is probably because only one new different category process was created, where there is a whole group of processes of an existing category already running. By doing this, we try to maintain the benefits of the current mode for the existing processes.

		Adaptive Functions Declarations									Rules													
		Ad1		Ad2							0	1	2	3	…	26	27	28	29	30	31	32	33
		H	?	H	+	+	+	+	+	+	S	R	R	R	R	R	R	R	R	R	R	R	E
Conditions	Current Mode				RR	PR	QV	FC	SJ	BT		RR	RR	RR	…	FC	SJ	SJ	BT	BT			
	State														…							N	
	II				C_1	C_1	C_1	C_1	C_1	C_1		-	x	-	…	-	-	-	-	-			
	M				C_2	C_2	C_2	C_2	C_2	C_2		x	-	-	…	-	-	-	-	-			
	ES				C_3	C_3	C_3	C_3	C_3	C_3		-	-	-	…	x	x	-	x	-			
	WEB				C_4	C_4	C_4	C_4	C_4	C_4		-	-	-	…	-	-	-	-	-			
	P				C_5	C_5	C_5	C_5	C_5	C_5		-	-	x	…	-	-	-	-	-			
	S				C_6	C_6	C_6	C_6	C_6	C_6		-	-	-	…	-	-	x	-	x			
Actions	func()				$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$	0	a	b	d	…	q	q	u	x	y			
	State=		D												…						N		
Adaptive Functions	Ad1	A			x	x	x	x	x	x		x	x	x	x	x	x	x	x	x		x	
	Ad2			B																		x	

Table 6.2 – Adaptive Decision Table for SODIUM Process Scheduler

B. *Adaptive mechanism*

In order to contemplate new rules for complex scenario conditions, such as II and M category processes running

The implementation of this mechanism onto the subjacent cDT in order to generate the first SODIUM process scheduler ADT is shown as the Table 6.2.

The action to be performed by every new rule is set by the $\$_m$ function that takes the rule's own starting mode and the current mode –as destination mode– as parameters to obtain the specific set of actions for that transition. The result of executing $\$_m$ are exactly those found in the Table 3.2.

An example of its adaptive functions can be illustrated as when a new scenario is detected in which the (M) and the (S) conditions are detected simultaneously. Six new rules are created to handle those conditions in combination with the six possible current modes, and are added to the subjacent cDT. The Table 6.3 shows the effect of their execution, highlighting the new rules in shaded green.

We will recur to the formal definition of the EADT from [13] and [14] in which multiple criteria can be defined in order to determine an alternative.

The original formulation consists on a base ADT such as the one obtained in the Table 6.2 and the addition of auxiliary functions (FM) that execute prior any other action and define values for variables that those actions will use.

In our case, we want to define the destination mode parameter for the function $\$_m$ using a multi-criteria method. The required steps for defining the method consist in three modules:

	Adaptive Functions Declarations								
	Ad1		Ad2						
	H	?	H	+	+	+	+	+	+
Current Mode				RR	PR	QV	FC	SJ	BT
State									
II				C_1	C_1	C_1	C_1	C_1	C_1
M				C_2	C_2	C_2	C_2	C_2	C_2
ES				C_3	C_3	C_3	C_3	C_3	C_3
WEB				C_4	C_4	C_4	C_4	C_4	C_4
P				C_5	C_5	C_5	C_5	C_5	C_5
S				C_6	C_6	C_6	C_6	C_6	C_6
func()				$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$
State=		D							
Ad1	A			x	x	x	x	x	x
Ad2			B						

	Rules																			
	0	1	2	3	…	26	27	28	29	30	31	32	33	34	35	36	37	38	39
	S	R	R	R	R	R	R	R	R	R	R	R	E	R	R	R	R	R	R
Current Mode		RR	RR	RR	…	FC	SJ	SJ	BT	BT				RR	PR	QV	FC	SJ	BT
State					…							N							
II		-	x	-	…	-	-	-	-	-				-	-	-	-	-	-
M		x	-	-	…	-	-	-	-	-				x	x	x	x	x	x
ES		-	-	-	…	x	x	-	x	-				-	-	-	-	-	-
WEB		-	-	-	…	-	-	-	-	-				-	-	-	-	-	-
P		-	-	x	…	-	-	-	-	-				-	-	-	-	-	-
S		-	-	-	…	-	-	x	-	x				x	x	x	x	x	x
func()	0	a	b	d	…	q	q	u	x	y				c	h	m	0	u	y
State=					…						N								
Ad1		x	x	x	x	x	x	x	x	x		x		x	x	x	x	x	x
Ad2												x							

Table 6.3 – Example of an ADT creating new rules for a formerly non-contemplated (M) and (S) Scenario

However basic, this mechanism actually learns from the users' behavior, and will converge into a complete 720 rules cDT differently for each user, or each run. On the other hand, it shows some limitations on the fact that, once created, the new rules can't be modified, even if the reaching scenario should indicate a new transition for that rule. Also, it doesn't provide the ability to contemplate multiple criteria for the decision making process. These problems are addressed by using the extensions shown in the following section.

VII. Extended Adaptive Decision Table

The ADT obtained in the previous section allows the creation of new rules that contemplate conditions that were not included in the original cDT. However, the only criterion used for determining the transition actions was that of maintaining the original current mode. This criterion, doesn't contemplate the usage of direct indicators of performance such as the metrics, nor a mechanism to alter the already created rules. Therefore, it is necessary to extend the definition of our ADT in order to include specific functions that could decide which mode to utilize based in the relation between the processes categories and the maintained metrics.

Module I consists in the identification of the different criteria –metrics, in our case– and alternatives –modes– for the decision problem. Their quantitative relation will define, in each case what alternative will be better for each scenario taking the metrics as reference. In our case we define the a C set of conditions, and an A set of alternatives as the following:

- $C = \{\%cpu, \#_f, t_{cv}, t_w, t_r, t_s, o_v\}$
- $A = \{RR, RRPR, RRQV, FCFS, SJFS, BTS\}$

	Criterion Preference						
Category	%cpu	$\#_f$	t_{cv}	t_w	t_r	t_s	o_v
(II)	1	1	1	3	3	1	1
(M)	3	1	1	3	1	1	1
(ES)	1	3	1	1	3	1	1
(WEB)	1	1	1	1	3	3	1
(P)	1	3	3	1	1	1	1
(S)	1	1	1	1	1	1	3

Table 7.1 – Criterion preference by category

Module II consists in obtaining a Z matrix of performance for each combination of criteria and alternatives. Using the Saaty fundamental scale for comparing the relative importance of each criterion with each category we could elaborate the Table 7.1.

On the other hand, we define the importance of each criterion pair will vary regarding the amount of processes of each category that is ready to execute multiplied by the criterion preference shown in the Table 7.1. For example, for an scenario where 2 (II) category processes, and 1 (S) category process are ready, the importance of the t_w and t_r metrics will be equal to 6, and that of the o_v will be 3.

With those values in mind, a criteria pair preference can be elaborated for the example as the one shown in the Table 7.2.

	Criterion Preference						
Criteria	%cpu	$\#_f$	t_{cv}	t_w	t_r	t_s	o_v
%cpu	1	1	1	6	6	1	3
$\#_f$	1	1	1	6	6	1	3
t_{cv}	1	1	1	6	6	1	3
t_w	1/6	1/6	1/6	1	1	1/6	3
t_r	1/6	1/6	1/6	1	1	1/6	3
t_s	1	1	1	6	6	1	3
o_v	1/3	1/3	1/3	1/3	1/3	1/3	1

Table 7.2 – Relative preferences between criterion pairs

	Criterion Preference							Total Preference
Category	%cpu	$\#_f$	t_{cv}	t_w	t_r	t_s	o_v	
Multiplier	x 1	x 1	x 1	x 6	x 6	x 1	x 3	
RR	0,35	0,13	0,05	0,18	0,16	0,23	0,15	0,17
RRPR	0,35	0,13	0,15	0,54	0,16	0,03	0,15	0,28
RRQV	0,12	0,13	0,30	0,02	0,02	0,03	0,45	0,25
FCFS	0,06	0,13	0,30	0,02	0,02	0,03	0,45	0,11
SJFS	0,06	0,38	0,30	0,02	0,02	0,03	0,05	0,06
BTS	0,06	0,13	0,05	0,06	0,16	0,68	0,05	0,13

Table 7.3 – Matrix Z of alternatives preference

It can be seen on Table 7.3 that, for the example with two (II) processes, and one (S) process, results in a better preference for the RRPR mode, just above that of the RRQV mode. The Z matrix will be regenerated completely based in the amount of ready processes per category each time that an adaptive function calls to a *z_gen()* named function. Another *z_get()* named function will be used to obtain the most preferred scheduling mode from the current newest Z matrix.

Module III consists in the development of the functions for the insertion of new rules for the non-contemplated scenarios in the moment that they are detected. This was already achieved in the TDA presented in the Table 6.2.

It will only take to add calls to the *z_gen()* and *z_get()* functions along with a new variable *m* to hold their obtained value. In the Table 7.4 the final form of the EADT for the SODIUM process scheduler is shown.

		Adaptive Functions Declarations									Rules												
		Ad1		Ad2							0	1	2	3	…	26	27	28	29	30	31	32	33
		H	?	H	+	+	+	+	+	+	S	R	R	R	R	R	R	R	R	R	R	R	E
Conditions	Modo Actual				RR	PR	QV	FC	SJ	BT		RR	RR	RR	…	FC	SJ	SJ	BT	BT			
	Estado														…							N	
	II				C_1	C_1	C_1	C_1	C_1	C_1		-	x	-	…	-	-	-	-	-			
	M				C_2	C_2	C_2	C_2	C_2	C_2		x	-	-	…	-	-	-	-	-			
	ES				C_3	C_3	C_3	C_3	C_3	C_3		-	-	-	…	x	x	-	x	-			
	WEB				C_4	C_4	C_4	C_4	C_4	C_4		-	-	-	…	-	-	-	-	-			
	P				C_5	C_5	C_5	C_5	C_5	C_5		-	-	x	…	-	-	-	-	-			
	S				C_6	C_6	C_6	C_6	C_6	C_6		-	-	-	…	-	-	x	-	x			
Extended Adaptive Actions	z_gen()																					x	
	z_get()																					m	
Actions	func()				$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$	$\$_m$	0	a	b	d	…	q	q	u	x	y			
	Estado=		D												…						N		
Adaptive Functions	Ad1	A			x	x	x	x	x	x		x	x	x	x	x	x	x	x	x		x	
	Ad2			B																		x	

Table 7.4 – Final form of the EADT for the SODIUM process scheduler

The values of the Table 7.2, obtained for this particular example, will vary depending on the current conditions scenario and the amount of processes per category. Nevertheless, continuing with the example, a normalized final Z matrix of performance can be obtained as the one shown in the Table 7.3.

Until now only brief tests and simulations for testing the EADT performance regarding different simple scenarios has been conducted due to the initial complexity of the implementation. Their results were satisfactory although yet not sufficient to determine its full potential. We estimate that in the next few months, new developments will support the usage of this powerful tool.

VIII. Conclusions and further work

With the usage of EADTs we were able to figure out alternatives for execution conditions of an aspect of an operating system that were too complex or inaccessible to figure out a priori. It also provided the kernel with the capability of changing the existing rules based on the new process usage behavior of each user. While these features may be possible to attain otherwise, none of the other existing adaptive devices provide such an intuitive mechanism for specifying rules and adaptive functions.

Although we are still lacking actual results from tests conducted on a variety of complex scenarios, the preliminary results on simple executions show that the decision tables converged into ideal solutions in each case, and that the kernel was actually learning from the process scheduler events. This actually serves as a demonstration that, so far, it is possible to create an automatic adaptive reconfiguration mechanism for a kernel without the supervision or explicit interactions with the users and their application.

There is still much potential to be harnessed from the EADTs. For example, we are not yet using the measures from the different metrics to evaluate each algorithm benefits from the actual execution. Doing this would converge and replace the initial heuristics on the algorithm-metrics relation tables.

Regarding SODIUM, there are other aspects of its design that are yet to be analyzed and converted into adaptively reconfigurable. That work should be done in the following months, during which we would still testing the results of the adaptive process scheduler.

Perspectives on the usage of adaptive mechanisms for automatic non-interactive reconfiguration are promising. These could be applied on any other home or enterprise operating systems in the market without the need of re-engineering their existing applications. An initial cost should be paid, nonetheless; mechanisms for automatic reconfiguration must be developed and provided as inputs for the EADTs, conditions must be analyzed, and events must be set.

References

[1] S. Martin, N. Casas, G. De Luca, “Diseño de un sistema operativo reconfigurable para fines didácticos y prácticos”. 6° Workshop de Tecnologia Adaptativa. San Pablo, Brasil, 2012.

[2] A. C. Veitch, N. C. Hutchinson, “Kea – a dynamically extensible and configurable operating system kernel”, 3rd International Conference on Configurable Distributed Systems. Vancouver, Canada, 1996.

[3] C. Cowan, T. C. Autrey, C. Krasic, C. Pu, J. Walpole, “Fast concurrent dynamic linking for an adaptive operating system”. 3rd International Conference on Configurable Distributed Systems. Vancouver, Canada, 1996.

[4] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, S. Eggers, “Extensibility: Safety and Performance in the SPIN Operating System”. 5th Symposium on Operating Systems Principles. ACM, New York, United States, 1995.

[5] R. Lea, Y. Yokote, J. Itoh. “Adaptive operating system design using reflection”. Object-Based Parallel and Distributed Computation. Volume 1107, Springer Berlin, 1996.

[6] N. Casas, G. De Luca, M. Cortina, G. Puyo, W. Valiente, “Implementación de distintos tipos de memoria en un sistema operativo didáctico”. XIV Congreso Argentino de Ciencia de la Computación. La Rioja, Argentina, 2008.

[7] H. Ryckeboer, N. Casas, G. De Luca, “Construcción de un Sistema Operativo Didáctico”. X Workshop de Investigadores en Ciencias de la Computación. La Pampa, Argentina, 2008.

[8] A. Silberschatz, P. B. Galvin, G. Gagne. “Operating System Concepts”. 8va Edición. John Wiley & Sons, New Jersey, United States, 2012.

[9] S. M. Varghese, K. P. Jacob, “Process Profiling Using Frequencies of System Calls”. The Second International Conference on Availability, Reliability and Security (ARES'07). Viena, Austria, 2007.

[10] R. McDougall, J. Mauro, “Solaris Internals”. Second Edition. Prentice-Hall. California, United States, 2007.

[11] J. J. Neto, “Adaptative rule-driven devices - general formulation and a case study”. Sixth International Conference on Implementation and Application of Automata. Pretoria, South Africa, 2001.

[12] T. Pedrazzi, A. Tchemra, R. Rocha, “Adaptive Decision Tables A Case Study of their Application to Decision-Taking Problems”. Adaptive and Natural Computing Algorithms, Springer. Vienna, Austria, 2005.

[13] A. H. Tchemra, “Tabela de decisão adaptativa na tomada de decisões multicritério”. Phd Thesis. Escola Politécnica, USP, San Pablo, Brasil, 2009

[14] A. H. Tchemra, “Adaptatividade na Tomada de Decisão Multicritério”. 4° Workshop de Tecnologia Adaptativa. Escola Politécnica, USP, San Pablo, Brasil, 2010.

[15] T. L. Saaty, “Método de Análise Hierárquica”. McGraw-Hill. San Pablo, Brasil, 1991.

Sergio Miguel Martin is a Software Engineer from Universidad Nacional de La Matanza (UNLaM), Buenos Aires, Argentina since 2010. He performs as a teaching assistant on the operating systems class since 2010; and on the automata and formal languages class since 2011. He is currently finishing his master thesis on Software Engineering on the same university. His main investigation fields are operating systems and high-performance computing.

Graciela Elisabeth De Luca Is a Systems Analyst from the Universidad Tecnológica Nacional, and Bachelor of Computer Science from the Universidad Católica de Salta. Since 2005, belongs to the SODIUM research group of the Universidad Nacional de La Matanza. Also performs as professor for the Operating Systems Class.

Nicanor Blas Casas, Is a Software Engineer from the Universidad the Universidad Católica de Salta. Since 2005, belongs to the SODIUM research group of the Universidad Nacional de La Matanza. Also performs as the associate professor for the Operating Systems Class.