A Runtime Environment for Online Processing of Operating System Kernel Events

Michael Schöbel, Andreas Polze
Hasso Plattner Institute at University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
{michael.schoebel, andreas.polze}@hpi.uni-potsdam.de

ABSTRACT

Different approaches were proposed for the logging of operating system kernel events. In general, the resulting logfiles are huge and have to be analyzed by administrators, who try to identify problems and derive adequate actions. The idea of autonomic computing is to automate such tasks.

As an important step towards this vision, computer systems have to be self-aware, i.e. they must be able to detect their runtime state and react if certain problems are detected. In contrast to control-theory based approaches for autonomic computing, the processing of discrete event streams offers the possibility of detecting singular events such as attacks or failing components directly.

Our proposed runtime environment (1) processes event pattern descriptions, (2) combines events generated by user-mode applications and the operating system kernel, (3) can be integrated into the operating system kernel to handle the events as close to their source as possible, (4) adaptively chooses relevant events to keep system disturbance low, and (5) provides an API for the implementation of ideas of autonomic computing in context of reactions to event patterns.

In this paper, the event pattern specification language and the runtime environment are described. The described prototype implements the envisioned runtime environment in user-mode and is able to look for event patterns in pre-recorded event logfiles. Additionally, an outlook on the planned operating system kernel integration is given.

Categories and Subject Descriptors
D.4.1 [Operating Systems]

General Terms
Algorithms, Measurement, Performance

Keywords
Operating Systems, Event Stream Processing, Pattern Detection

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WODA '09, July 20, 2009, Chicago, Illinois, USA.
Copyright 2009 ACM 978-1-60558-656-4/09/07 ...$5.00.

1. INTRODUCTION

Different approaches were proposed for the logging of operating system kernel events. In general, the resulting logfiles contain a huge amount of data. Administrators analyze such logfiles (utilizing various tools) and derive adequate actions if certain problems were able to be identified.

Tools for monitoring the system state can be classified as sampling-based and tracing-based. Sampling-based tools (e.g. Windows Management Instrumentation [14] or VTune [2]) periodically gather system information and use statistical methods for analysis. Tracing-based tools (e.g. Magpie [10] or WPT [3]) gather and analyze (complete) streams of events. Both classes of tools have specific advantages and drawbacks. Tracing-based tools require event logging - the code for event creation has to be present in the system (and may be activated at runtime) or has to be inserted into the system dynamically. Furthermore, the number of events cannot be predicted beforehand. Therefore, the (performance) impact of the analysis can not be predicted or limited. Sampling-based tools require measurement points in the system which can be polled periodically. The analysis impact can be tuned by choosing the polling period. The main disadvantage is that system state changes may be missed if they occur between two measurements.

The idea of autonomic computing is to automate the cycle of system analysis and adaptation. Therefore, the system needs the ability to perceive its state (self-awareness) and to adapt its configuration (self-* properties, such as self-healing, self-optimizing, or self-defending).

In this context, the appropriate way to implement self-awareness depends on how the system adaptation should happen. One approach is using concepts from control theory: the system state is read via sensors, compared to an ideal state and adapted via actuators. Such an approach corresponds to system analysis using sampling-based tools: Metrics such as latency, throughput or memory consumption are monitored. These values are compared to specific goal-values. Using a system model which models dependencies between sensor values and system parameters, the parameters are adapted to match the goal-values.

Automating the usage of trace-based tools offers advantages in some usage scenarios: special system states, failing components or attacks can be detected more easily while scanning a complete event trace. In a sampling-based approach, the underlying system model must be extended to detect singular events such as attacks. Such an extension might be not possible: How does an attack or a failing component affect latency and throughput? And, how can it be
distinguished from other causes such as “regular” bugs?

The transfer of tracing-based approaches to autonomic computing requires techniques for pattern matching in large event streams. Rules assigned to such patterns are executed as a reaction to a successfully detected pattern and may result in a parameter adaptation, in an alert email to the administrator, or in a log file entry.

In this paper a system is proposed which allows to analyze event streams by searching for specific patterns. A pattern specification language allows to describe complex event patterns and (optional) reactions to these patterns. A programming interface enables applications to register patterns in the system and to react if such patterns occur.

The main contributions of this paper are: (1) We describe an event pattern definition language for usage in operating system kernels. (2) We design a runtime environment which processes such event pattern definitions. (3) We describe how the runtime environment can be integrated into the operating system kernel.

The proposed system can be used for the dynamic runtime analysis of operating system kernel events without writing the events to a log file, i.e. the events are processed online as opposed to the more common offline analysis.

The remaining part of the paper is structured as follows: in section 2 the intended system environment is described, afterwards in section 3 a language for the description of event patterns is specified. Section 4 introduces examples in order to clarify the usage of the proposed approach. The prototypical runtime environment is described in section 5. Sections 6 and 7 describe the planned kernel integration and show preliminary evaluation results. After reviewing related work in section 8, the paper concludes in section 9 with a summary and conclusions.

2. SYSTEM MODEL

The proposed approach for event processing relies on assumptions about the underlying system environment. This section describes these assumptions.

In general, an event logging infrastructure is required. The infrastructure consists of two main components: (1) event provider and (2) event logger. The providers are responsible for generating specific events at appropriate points in the system. These events are recorded by the event logger. Furthermore, the event logger provides the recorded events to any application interested in processing the events.

The actual implementation of an event provider and an event logger is not relevant for our event processing approach. Such providers might be integrated in the operating system kernel, might be loaded dynamically at runtime, or might reside at a user-mode application. The event provider notifies the event logger of new events in a way, specific to the event logger implementation.

The event logger collects the occurring events and provides them to any event processing application. Such an event processor could write events to a log file, use the events to generate a notification to an administrator, or process the event in any other arbitrary way. Our runtime environment is implemented as such an event processor.

In general, there is a synchronous way and an asynchronous way to deliver events to an event processor. In the synchronous mode, events are delivered to the event processor when they occur and the event generating application is blocked until the event processor has finished processing. In the asynchronous mode, the event processor reads events from a buffer and event generating applications are not blocked.

Blocking applications during event processing makes sense in scenarios where (1) events can be directly mapped to an application, e.g. by evaluating the process/thread identifier, and where (2) the processing of generated events is relevant for the execution of the application, e.g. a security monitor might terminate an application if specific malware patterns occur in the execution flow.

Our prototype runtime environment supports (synchronous) processing of event logfiles. Support for asynchronous processing will be integrated in the runtime environment which is developed for operating system kernel integration.

Some assumptions are made about processable events:

**Definition 1 (Event).** An event \( E \) is described by a timestamp \( T \), an execution context \( C \), and a set of event data \( D \). The event data describes the state change which is caused by the event.

The way of representing and capturing events depends on the specific instrumentation framework, i.e. the event logger. Timestamps can be assigned using different kinds of time sources. The quality of the time source determines timestamp resolution.

The execution context of an event is defined as follows:

**Definition 2 (Execution context).** An execution context \( C \) of an event describes the execution environment of an occurring event. Therefore, possible parameters of \( C \) are determined by the execution model of the underlying operating system. At each point in time at most one activity can be executed in a specific execution context. All events in the same execution context are ordered in a “happens-before” manner.

Many operating systems use a process and thread based execution model. In these cases, an event occurs in the context of a specific CPU core, a specific process, and a specific thread. At each point in time only one activity (= thread of execution) can be executed in this context.

Other kinds of execution context which are not based on the notion of processes or threads, are e.g. interrupt service routines (ISRs), or (in a Windows-based operating system environment) deferred procedure calls (DPCs).

The concept of an execution context allows limiting the number of events to be analyzed to a specific subset, i.e. all events which occur in a specific execution context. Furthermore, the execution context of events adds flexibility to the specification of event patterns as described in the next section.

Events are collected in streams which are defined in the following way:

**Definition 3 (Eventstream).** An eventstream \( S \) collects all events occurring in a system. The eventstream orders the events and (virtually) assigns them a unique index number \( n \): \( S = \{ E_0, E_N \} \).

The ordering depends on the event logging infrastructure and might be independent from the event timestamps. We assume that the event logging infrastructure ensures the following property for an eventstream \( S \): If we consider two
events \( E_x \) and \( E_y \) from \( S \) which occurred in the same execution context \( C \), then: \( x < y \rightarrow E_x.t \leq E_y.t \), where \( E_x.t \) denotes the timestamp of event \( E_x \). Additionally, if \( x < y \) then \( E_x \) happened before \( E_y \), even if \( E_x.t = E_y.t \).

With these basic definitions for events, execution contexts, and eventstreams the remaining parts of the paper describe our approach for online processing of eventstreams in an operating system kernel.

### 3. EVENT PATTERN SPECIFICATION

In this section, we introduce the general concepts of our prototypical event pattern definition language. Examples are given in section 4. Event patterns can be specified using the grammar shown in table 1. The language uses concepts from EventScript [6] and SASE+ [7].

An EPC (event processing) file contains a sequence of epcstatements, which are either an eventdefinition or a ruledefinition.

An eventdefinition is used to define the available event types and the structure of the event data. Each event contains event header information (such as timestamps or execution context) and event data fields (arbitrary information about the occurring event). Currently, the EPC compiler parses a regular C header file with event definitions and extracts the event structure information. The datatypes of the event data fields are saved to ensure typesafe usage of event data fields in other parts of the EPC file.

The event definitions depend on the event logging infrastructure. Currently, the compiler supports our research instrumentation framework called Windows Monitoring Kernel [13].

After defining available event types, a set of rules can be specified in an EPC file. Each rule definition contains at least a name and a pattern definition. Additionally, the mode of rule processing (either synchronous or asynchronous) and additional rule modifiers can be specified. The mode of a rule defines whether events are processed in the execution context in which they occur (synchronous processing, blocks the activity which causes the event) or not (asynchronous processing, no blocking). Rule modifiers are used to fine-tune the rule definition, e.g. by specifying additional relations between certain events or by setting a time limit for the pattern occurrence.

The basic elements of each pattern definition are either single events or arrays of events. Single events are specified by giving an event type name, arrays are specified by an event type name and “\[\]”. Optionally, length limitations can be specified in the brackets, e.g. \[<5\] defines that less than five events of type \( a \) are expected to match the pattern. Single events and arrays can be named by using a colon: \( type\[\]:name \) specifies a named single event and \( type\[\]\] specifies a named array of events. These names can be used e.g. in rule modifiers to compare specific event data fields.

Event patterns can be constructed analogous to regular expressions:

- An event sequence \( [a,b,c] \) defines that events of types \( a \), \( b \) and \( c \) have to occur subsequently in an eventstream to match the pattern.
- An alternative \( (a|b) \) defines that either an event of type \( a \) or an event of type \( b \) is expected.
- A negation \( \neg X \) specifies that a certain pattern \( X \) does not occur in the eventstream.

These structure elements can be nested in arbitrary ways, e.g. \( [a,\neg[b,\neg c],[d,e]] \) is a valid pattern specification. Repetitions of composed sub-patterns (kleene star operators) are currently not supported - for repetitions of single events arrays can be used.
Additionally to the event pattern structure, the pattern semantic has to be specified. There are three different semantic modes for a pattern specification:

- **STRICTSEQUENCE**: all specified pattern elements occur in a strict sequence in the combined eventstream containing all occurring events in the system.

- **STRICTPARTITION**: all pattern elements occur in a strict sequence in a sub-eventstream (partition) containing only events of a specific execution context.

- **SKIPTILLNEXT**: no strict sequence of pattern elements is required, irrelevant events are skipped.

After specifying the event pattern, additional conditions can be defined. These conditions allow to define relations between event data fields or to define join fields. In the current prototypical implementation a single event data field or a pair of event data fields can be combined and compared to another single event data field or pair of fields. The common arithmetic and relation operators can be used. Special array operators provide access to the number of array elements and the minimum value, the maximum value, or the average value of the array elements. These basic operations are sufficient for our proof of concept implementation. Support for arbitrary calculations might be added in future versions.

Event data fields can be used as join fields by using square brackets, e.g. `WHERE { [name] }`. This example specifies that all events used in the `PATTERN` statement must contain a data field `name` and all matching events must have the same value assigned to this field.

Join fields which are part of the execution context define a eventstream partition that is considered in the `STRICTPARTITION`-semantic.

A rule can have a timeframe which defines the maximal difference between the timestamp of the first detected event of a pattern and the final event. In our prototype, the timeframe can be defined based on the CPU cycle counter which is used for event timestamping. Alternatively, time can be specified in seconds. The runtime environment calculates the corresponding value for the CPU cycle counter.

The last element in a rule description is the `RETURN` statement. There are two possible results a rule can provide: (1) the rule can return the whole event sequence which matched the pattern, or (2) the rule can return a set of event data fields.

Concepts that are planned for extensions of the language are: (1) A `DO`-statement for action specification which are executed immediately after a defined pattern is detected. The set of useful actions and security aspects has to be investigated. (2) String operations, e.g. for file name comparisons.

4. USAGE EXAMPLES

To demonstrate how the proposed rule definition language can be applied to different problem areas, this section presents basic examples for dynamic analysis using rule definitions.

Listing 1 shows an example of a rule description to detect long running system service calls. Such a rule could be used to compare the behaviour of the system with the expectations, or for system analysis in general.

```
Listing 1: Detect long running system service calls
1 ASYNCHRONOUS RULE longsyscalls
2 STRICTPARTITION PATTERN
3 { [syscall:a, syscallexit:b] }
4 WHERE
5 { [ProcessId],
6 [ThreadId],
7 b.TimeStamp - a.TimeStamp > 2s }
8 RETURN
9 { a.SyscallNr }
```

For system analysis, we do not need to take action immediately. Therefore, the `longsyscalls` rule is asynchronous. We want to compare pairs of system service call events and system service call exit events - the pattern description `[syscall:a, syscallexit:b]` specifies such a sequence. These sequences only make sense in a specific execution context, the `WHERE` statement defines the context: both events must have the same process identifier and the same thread identifier. Furthermore, we are only interested in “long running” system service calls, therefore, the last condition (line 7) specifies that the timestamp difference between system service call entry and exit has to be greater than two seconds. Finally, the `RETURN` statement specifies that we are interested in the system service call number as return value.

We assume that the fields `ProcessId`, `ThreadId` and `TimeStamp` are parts of the event header, and therefore part of every possible event type. Additionally, it is assumed that a `syscall` event contains a field definition `SyscallNr` which contains the actual system service call number.

The rule, shown in listing 2, can be used to detect lock contention, i.e. concurrent acquire operations for the same synchronization object.

```
Listing 2: Detect lock contention
1 SYNCHRONOUS RULE lockcontention
2 SKIPTILLNEXT PATTERN
3 { [lock:a, "lock:b", lock:c] }
4 WHERE
5 { [ObjAddress],
6 a.Operation == 1,
7 b.Operation == 0,
8 c.Operation == 1 }
```

This rule analyzes `lock` events which belong to the same synchronization object (= the event field `ObjAddress` has the same value). The first `lock:a` event tries to acquire the synchronization object (`Operation` is 1), the second `lock:b` event releases the object (`Operation` is 0), and the third event also tries to acquire the object.

By negating the second event (i.e. the event must not occur in the eventstream) the pattern can detect lock contention: a synchronization object can only be acquired by one single thread, therefore a release operation can only be generated if the synchronization object was acquired successfully. If there is no contention on the synchronization object, event `lock:a` and event `lock:b` are generated by the same thread and the pattern detection aborts. If the release is not detected and another acquire event occurs, it must be from a different thread and the lock is contented.

The exclusion of recursively acquired locks would require a more complex rule.
The rule is executed in a synchronous way, therefore the execution of the activity that generates the lock:c event is blocked if the described pattern occurs. In such cases the locking strategy might be adapted or an entry might be logged for later analysis.

Listing 3 shows a short example for the processing of application specific events. We assume that an interactive system generates (1) login events if a user (identified by the event field UserName) logs into the system, and (2) operation events which encapsulate the name of the current user and the operation type (type 0: regular, uncritical operations, and type 1: critical, privileged operations). The operation event is logged before the actual execution, i.e. an operation event occurs if the specific operation “is about to happen”.

### Listing 3: Detect invalid operations
```
SYNCHRONOUS RULE invalidoperation
SKIPTILLNEXT PATTERN
{ {login:u, operation:o] }
WHERE
{ {UserName],
u.Type < o.Type }
```

It should be ensured that users can only execute operations at their corresponding privilege level. In the example login and operation events are monitored. The pattern matches if the user type is not allowed to execute a specific operation, i.e. the Type of the user is less than the Type of the operation.

Again, this rule is marked as synchronous. The execution context of the operation gets blocked when generating the (invalid) operation event. The system can handle the exception e.g. by aborting the operation.

These examples show that there are different application domains for online processing of generated events. Rules can be loaded when needed, to analyze certain problems. Additionally, rules can be used to implement ideas of autonomic computing: a pattern description implements the self-awareness of the application - system adaptation happens when a particular pattern is detected. More complex case studies are subject to ongoing work.

## 5. RUNTIME ENVIRONMENT

The runtime environment consists of the following components: the compiler for EPC specifications and a user-mode application which loads compiled rule definitions and processes pre-recorded event logfiles. These components and the execution model for rule processing are described in this section.

The compiler was implemented using the Coco/R\(^1\) framework. It compiles a rule specification (compliant with the described grammar) into a deterministic finite automata. Due to space restrictions we omit details about the compilation process. Converting regular expressions into automata is a well investigated topic.

Conditions and actions are assigned to automata transitions. Conditions are checked to determine if the transition is valid at the current state of the automata, considering the current event. E.g. the expected event type and the WHERE conditions are checked via conditions. Actions are executed if the transition is actually chosen. Actions might save specific data fields of the current event for later reference or modify certain aspects of the runtime environment for the specific rule.

The binary representation of the automata is loaded by the runtime system. If an event occurs, the runtime system identifies rules that are interested in the specific event. The transition from the initial automata state to the next state (i.e. when the first event relevant for the pattern occurs) starts an automata run.

Each run is represented in memory by a runtime state representation. The data structure for runtime state representation is generated by the compiler. It contains the current automata state, event field information (which is initialized by transition actions), and information about results that is returned if a pattern is detected. These items can be derived from the EPC script.

When a new event occurs, the event information is stored in an execution context local buffer. That means e.g. that each thread has its own buffer for occurring events. The event information is then fed into the event processor. The event processor manages a global buffer for event data. This buffer contains event data that is relevant for rule results and event data that does not require immediate processing.

First, the event processor checks if there are any synchronous rules waiting for the new event. If this is the case, the event processor evaluates the specific rule automata. Only if the event data is relevant for rule results (e.g. if the rule has to return the whole event sequence), the event information is copied to the global event buffer. Secondly, if there are any asynchronous rules waiting for the new event, the event data is copied to the global event buffer (if not already done as a result of processing a synchronous rule) and a reference is stored in the asynchronous event queue.

Afterwards, the execution context local event buffer can be reused for the next event. Events from the asynchronous event queue are processed by a special thread.

As already described, the runtime state of a specific rule encapsulates all information relevant for a potential event pattern match. Each rule has its own memory pool for such runtime state information. The pool size can be configured to either contain a specific number of state information or to have a specific size. Then, the actual pool size determines how many event pattern matches can be detected parallelly.

This concept of memory pooling was chosen for the following reasons: (1) the amount of memory required to process a specific rule is predictable, (2) allocating and deallocating memory for runtime state information (which in general requires only a few byte) is inefficient, and (3) allocating the memory before the start of rule processing ensures that the required memory is indeed available.

The event processor-managed global buffer is subdivided into smaller buffer units. Each of these units maintains a reference counter which describes how many event data entries are still referenced by runtime state entries. If the reference counter reaches zero, the particular buffer can be re-used to store event data.

We believe that saving the whole event sequence is not required in most cases. The proposed way of using the described event stream processor is to detect patterns which allow immediate action to fine-tune the system. It is highly probable that only few event data parameters are required to configure the action when the patterns actually occur.

\(^1\)http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/
We investigate different heuristics to improve the performance of the runtime environment: (1) Adaptive event activation and deactivation. The runtime environment can determine the set of events that is relevant for the loaded rules and for current automata runs. By interacting with the instrumentation framework, the generation of irrelevant events can be turned off to reduce system disturbance. (2) Evaluation ordering. The compiler generates deterministic automata, i.e. there is always exactly one valid transition for a specific state and a specific event. Transitions and conditions are evaluated sequentially. Based on probabilities of evaluation results, reordering the evaluation sequences can lead to increased performance of rule processing.

6. KERNEL INTEGRATION

The described runtime environment can be integrated into an operating system kernel. Currently, an integration into the Windows Research Kernel \[12\] is being worked on.

The integration of the core runtime environment as described in the last section is possible without major problems. The described rule and buffer management can be implemented in kernel-mode as well. This section will focus on the advantage such a kernel integration offer: the programming interface for applications. If applications can react to detected event patterns, performance improvements might be achieved e.g. by re-configuring the application to prevent specific unfavorable patterns. Another advantage is the possibility to synchronously react to detected mal-ware patterns while the “bad” activity is blocked.

The API provides functions for rule management and result management. Rule management functions are required for loading and unloading rules into the operating system kernel, and for querying the current execution state (e.g. enumerating loaded rules or reading statistics about rule execution). Result management functions are used for registering callbacks and, in general, for defining what to do if a specific pattern is detected in the event stream.

From an application developer point of view, the most important function provided by the API is RegisterRule. An example for using this function is shown in listing 4.

Listing 4: Register rule and callback

```c
int resultCb(RuleInfo r, ResultData d) {
    [...] return CONTINUE;
}
main() {
    RuleHandle h = RegisterRule("rule.epcc", resultCb);
    [...] UnregisterRule(h);
}
```

The rule definition file rule.epcc is loaded with RegisterRule at line 8. If this rule detects the specified pattern, the callback function resultCb is called. This function is defined at line 1. RegisterRule returns an identifier for the loaded rules. A rule can be unregistered with UnregisterRule by passing the rule identifier as parameter.

If the rule specification contains a DO statement (i.e. a compiled action which is loaded into the kernel), the result callback has to be NULL. If no action is specified in the rule definition, the callback has to be defined. Now, the rule mode must be distinguished: If the rule has to be processed asynchronously, the return value of the callback function is ignored by the system. When processing rules synchronously, a decision must be made whether the activity which causes the pattern match can continue its execution or not. This decision is signaled by the callback return value: CONTINUE or ABORT.

The registered rule callbacks are processed in a dedicated execution context, i.e. a dedicated worker thread. These threads do not generate any events to prevent situations in which the callback handler thread gets blocked by another callback handler thread.

Additional points to consider when integrating the runtime environment into the operating system kernel are further safety and security aspects. Furthermore, the kernel integrated runtime environment has to consider multiprocessor issues such as different timebases for different CPUs. These areas require further investigations.

7. EVALUATION

Due to the early stage of the kernel integration of the proposed runtime environment, preliminary evaluation results can be given that are based on our prototype user-mode implementation. Our instrumentation framework, the Windows Monitoring Kernel (WMK) \[13\], is integrated into the kernel and allows logging of operating system kernel events.

For test purposes we used an event logfile that contained 1200000 events. These events were recorded using the Windows Monitoring Kernel during one single boot process of a Windows Server system. Recorded events contained (amongst other types) system service calls, context switches, synchronisation operations and file access events. For performance evaluation we used an Intel Core2Duo, 3 GHz system with 4 GByte RAM.

For test purposes the rule longsyscalls was used. Without any additional optimizing heuristics, the runtime environment was able to process 40000 events per second (this includes overhead stemming from reading the logfile from disk). The memory consumption was quite low: a pool of 500 runtime state entries was enough for all long running system calls to be identified.

The memory consumption depends on the number of concurrent automata runs - which in turn depends on the kind of patterns to detect. In the system service call example the upper bound of concurrent runs is set by the number of concurrent threads, i.e. each thread can execute at most one service call at a time.

By applying the heuristics described in section 5, the event processing rate can be increased. Further investigation is required to assess the effects of the proposed heuristics.

Gathering the logfile data using the WMK leads to a system slowdown of 5% at an event rate of 64000 events per second. This is comparable to other instrumentation frameworks for the Windows platform, e.g. ETW \[1\]. The additional overhead caused by online processing of the occurring events needs further investigation when the runtime environment is integrated into the kernel.

8. RELATED WORK

Several different approaches for operating system kernel instrumentation were proposed for different platforms and
different use cases, e.g. DTrace [5], KLogger [8], LTT [15], or ETW [1]. Our work as described in this paper focuses on the online processing of operating system kernel events. The actual instrumentation technique is orthogonal to the proposed concepts. Most of the instrumentation frameworks comply to the system model described in section 2.

The research area of event processing was investigated recently in the context of business applications. Eventstreams with high volumes of data can be generated e.g. by using RFID enabled warehouse management or by monitoring database transactions. Pattern analysis in such eventstreams may be used for theft detection [7] or to find unusual activities in business processes [11]. We apply these concepts to the processing of events generated in the operating system kernel.

Our event pattern specification language combines concepts from EventScript [6] and SASE+ [2]. EventScript proposes that the description of event patterns can be done similarly to regular expressions. SASE+ introduces the concept of different event selection strategies which further reduces the complexity of pattern descriptions in some use cases. Additionally, our language provides explicit support for execution contexts and for the specification of synchronicity.

Few details have been published about runtime environments for event stream processing. The proposed runtime environment for SASE+ [4, 7, 9] describes a result management approach called active instances stack. These stacks provide the possibility of result sharing between different pattern detection runs. Our result buffer management uses similar concepts, but the degree of sharing might be lower. With the concept of runtime states that contains aggregated result information, we think that the recording of complete event streams is not necessary in most cases.

The application of eventstream processing in the domain of operating system kernels differs in another aspect from the business domain use cases: The notion of an execution context allows synchronous event processing - in database systems or RFID scenarios it is hard (or impossible) to block the event generating activity. In our envisioned use-case (dynamic analysis of operating system kernels and applications) these concepts offer additional possibilities for specifying event patterns.

9. SUMMARY AND OUTLOOK

In this paper an approach for online processing of eventstreams is described: A pattern specification language allows to define event patterns and conditions - a runtime environment is able to load such definitions and scan a stream of events for these patterns.

A compiler for the pattern specification language and a prototypical runtime environment were developed. The current prototype is executed in usermode. The kernel-mode integration is work in progress. Concepts of the kernel integrated version of the runtime environment (e.g. programming interface or buffer management) are described in the paper.

The online processing of event streams offers the advantage that the eventstream itself must not be stored to disk. Synchronous event processing allows to block activities if they generate “bad” patterns. Furthermore, the runtime environment API provides functions which can be used to build self-aware applications, and to implement concepts from autonomic computing.

10. REFERENCES