Targeted on-line data extraction with SystemXtract

Johannes Schützel
Modeling and Simulation
Institute of Computer Science
University of Rostock
Rostock, Germany
johannes.schuetzel@uni-rostock.de

Sebastian Stieber
Applied Microelectronics and Computer Engineering
University of Rostock
Rostock, Germany
sebastian.stieber2@uni-rostock.de

Christian Haubelt
Applied Microelectronics and Computer Engineering
University of Rostock
Rostock, Germany
christian.haubelt@uni-rostock.de

Adelinde M. Uhrmacher
Modeling and Simulation
Institute of Computer Science
University of Rostock
Rostock, Germany
adelinde.uhrmacher@uni-rostock.de

ABSTRACT
Complex system-level simulation can produce large amounts of data, of which only portions may be of interest. When experimenting with hybrid prototypes, consisting of physical and simulated components, data logs are generated and inspected in *real-time*. Storing full data logs would not only require much disk space, it would also require much effort to find special events and related system actions afterwards. *Targeted on-line data extraction* helps to instantly provide the data of interest. We present SystemXtract, a powerful specification language and tool for on-line data extraction, supporting origin-, value- and dynamic phase-based constrains on the data. The main contribution of this paper is to show how the language can be mapped to a graph-based processing architecture for executing data extraction as specified. Experiments show that the tool-induced overhead in computation time is insignificant and that the real-time execution of the hybrid prototype is not compromised, while the output is reduced to the interesting data.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging; H.2.3 [**Languages**]: Query Languages; I.6.6 [**Simulation and Modeling**]: Simulation Output Analysis

General Terms
Experimentation, Languages, Algorithms

Keywords
Data Extraction, Specification Language, DSL, Filter Graph

1. INTRODUCTION
Recent embedded systems become more and more powerful and complex. To support the development of such systems and, ultimately, to increase the quality of the final product, the system behavior is modeled at an early design phase in high-level programming languages such as C/C++ and SystemC [10][15]. The created system-level model is used for the design space exploration in terms of testing new functionality and making architectural decisions. One step further, adding already available physical subsystems to those system-level models leads to a full-scale hybrid prototype. As a result, multiple components, whether physically available or simulated, of a complex embedded system can be validated and verified together. Moreover, interfaces and compatibility of the subsystems' functionality can be checked and first experiments, testing the interaction of the embedded system with its physical environment, can be conducted.

Running such hybrid experiments requires specific solutions for synchronization and instrumentation. In most cases, the simulated components have to meet soft real-time constraints for seamless interaction with existing physical components. At the same time, there is a need for logging internal simulation states as well as the system components' interactions and transferred data to analyze the prototype and to debug potential misbehavior or to find corner cases. The debugging of such a complex hybrid prototype can be challenging as it produces a vast amount of logging data, which has to be parsed carefully by the developer to find the important information.

The data volume and analysis are challenging if experiment data is stored in its entirety and interesting data is extracted *off-line*, e.g., using text processing tools such as grep, sed, or awk. However, in debugging hybrid prototypes, test inputs are made via *real-time interaction* with the system's physical part. There, problems are investigated most effectively if the interesting data is available *instantly*. In this case, the off-line analysis approach is unsuitable. Hence, we aim for *on-line* data extraction. *Targeted* on-line data extraction

SIMUTOOLS 2015, August 24-26, Athens, Greece
Copyright © 2015 ICST
DOI 10.4108/eai.24-8-2015.2260916

means to extract only those data items from the running simulation that are of certain interest.

Thus, the challenges are:

- to constrain the outputs in a way that allows the developer to identify relevant system behavior in the debugging process more easily;
- to perform the data extraction on-line, because the developer needs to see results instantly;
- to be sufficiently fast, because simulation and hardware components interact in real-time.

In this paper, we present an approach for on-line data extraction from instrumented system-level models. In Section 2, we start with a discussion on existing data extraction approaches. Section 3 introduces an application example from embedded system development and motivates the development of more powerful domain-specific languages for specifying data extraction. In Section 4, we present our approach, including a novel specification language and its mapping onto a graph-based processing architecture. In Section 5, we present a software tool implementing our approach, and we show how the approach performs in experiments we carried out in the context of embedded systems development. In Section 6, we conclude that our approach meets the requirements of the application domain and we provide an outlook of language capabilities waiting to be realized.

2. RELATED WORKS

A straightforward way of extracting state and trace data for debugging system-level models is the "good old" `printf` method. No need to say that placing `printf`s (and additional helper code) into the model code provides great flexibility, but making changes to such an instrumentation is laborious and error prone [14].

Another way of targeted state and trace data extraction is provided by configurable logging libraries such as log4j [9] and log4c [8]. While it is still required to integrate logging code into the model code, configurability allows for a non-intrusive de-/activation of logging. Targeted logging is configured based on log categories (i.e., a label, typically corresponding to the model structure) and log levels (e.g., debug, info, and warning). Which categories and levels to consider for output and where to store the output is typically configured via Java properties files (e.g., in log4j v1.2) or XML files (e.g., in log4c and log4j v2).

Since logging libraries such as log4j and log4c were developed for general program debugging, log messages are filtered only with respect to their category or log level, but not with respect to *time*. However, restricting the log output to certain time periods can dramatically ease debugging simulation prototypes. This is because, for instance, some problems are known to occur any time but not before the simulation has passed an eventful initialization phase which would clutter up the output with a mass of uninteresting log messages. Moreover, for finding problems that are potentially related to physical inputs one might want to restrict data extraction to phases that are adjacent to such events (cf. Fig. 1). Another shortcoming of general purpose logging libraries is that they consider log messages as closed entities, the content of which has to be interpreted by the debugging person in order to be identified as being of interest or not. In debugging hybrid prototypes, log messages arrive at high frequency and problems might have to be sighted in real-time. Therefore, it is considered useful to filter log messages also by their content, i.e., by contained numerical values or other information.

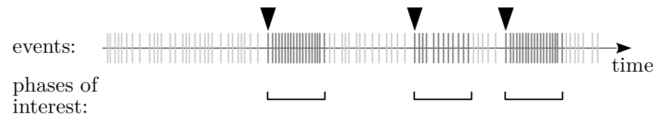


Figure 1: If specific events (marked by arrows) trigger interesting behavior (depicted as phases of increased event frequency), data extraction that is restricted to respective phases of interest can assist in investigating those behavior.

The logging framework CULT enhances the functionality of common logging libraries by time-based and value-based output restrictions [12, 13]. The framework is applicable to programs written in C, C++, or SystemC—the latter being a C derivative for system-level modeling and simulation [1]. CULT provides logging support for ordinary log messages, for standard C++ types such as `int` and `double`, and for common SystemC types such as `sc_signal` and `sc_port` [13]. For the numerical types, value-based output restriction can be accomplished by defining `log_max_value` and/or `log_min_value` as thresholds for the logged value. Time-based output restriction can be accomplished by defining `log_begin_time` and `log_duration`. These definitions refer to operating system time or to simulation time, dependent on whether logging is set into C++ mode or SystemC mode. The logging can be configured globally for the program/simulation or separately for SystemC modules. To handle the complexity of simulation- and module-level configuration, which may include restrictions on log levels, time periods, and/or values and includes the output location, the CULT developers choose XML as the host language for external configuration (programmatic configuration is also possible).

Albeit being capable of representing complex configurations through nested structures and being—technically—human-readable, XML is "verbose and not always ideal for human use" [5]. For instance, an XML file for configuring CULT requires 12 lines at the very minimum (using the common line break policy for readability). When defining value- and time-based restrictions specifically for a certain module in a SystemC model, the configuration file easily grows to 27 lines and 6–11 additional lines per definition for each further SystemC module. This verbosity arises from the opening and closing tags required to enclose configuration values and configuration blocks. Java properties files, as used in log4j [22], log4j-inspired log4cxx [21], and log4cpp [3], store configurations in the form of key-value pairs. Hierarchical configuration structures are represented in a flat style using keys of the form `log4j.appender.stdout`. Writing and understanding such configurations is not perceived to be as intuitive as natural language-inspired languages such as SQL [6] or others with bracket-based structuring.

3. A MOTIVATING EXAMPLE

An application for data extraction from hybrid prototypes is the development of inertial sensor systems being integrated in smartphones. Those inertial sensors constantly provide orientation and motion information, including to which extent the phone is accelerated along its X, Y, and Z axes. Such information may be used, e.g., to track the user's sporting activity or to detect motion-based input gestures such as joggling the phone. Another application, especially of interest for elderly users, is to detect the event of falling, e.g., to initiate an emergency call. The free fall of a smartphone can be detected by the fact that acceleration values almost go to zero because of the lack of net gravitational pull during the fall. Detecting motion events is a computationally intense and, thus, power consuming task if performed by the smartphone's CPU, what is often the case. With hardware developing rapidly, there is a trend to move this task from the phone's CPU to the sensor chip. Now being equipped with digital subsystems, sensor chips get increasingly complex. During the development of such sensor systems, algorithms and intercommunication have to be tested and possibly debugged. This can be done by building a hybrid prototype, experimenting with it, and observing its behavior.

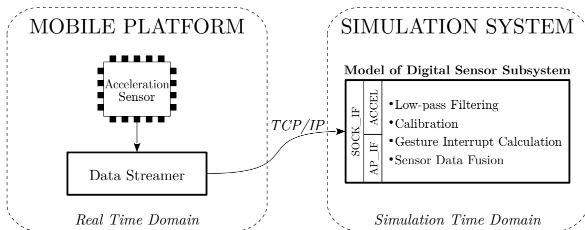


Figure 2: Schematic of a hybrid prototype of a sensor system.

Figure 2 shows the structure of an exemplary hybrid prototype. The physical part of this prototype is a mobile hardware development platform, hosting an acceleration sensor. The sensor is continuously read out and data is streamed to the other part of the prototype—the simulation system. The simulation system inputs the sensor data into a simulation model of the digital sensor subsystem to be developed. This way, new features of the digital sensor subsystem can be simulated—for instance gesture interrupt calculations and sensor data fusion, as done by current sensor products [4].

In our example of experimenting with an inertial sensor prototype, the developer is interested in the system's behavior during and after free fall, e.g., to check whether the "falling" event is detected properly. More specifically, let us assume, the developer is interested in data from the inertial sensor (called `ACCEL` in the simulation model), from the application interface (`AP_IF`), and from the socket interface (`SOCK_IF`), but from the latter only those data transmitted via the I^2C bus (`I2C`). As stated, the developer wants to see data only after measuring free fall conditions, i.e., `ACCEL` reporting values near to zero for X, Y, and Z. However, the developer does not need to see data that is produced later than 6000 nanoseconds after a free fall is detected. A corresponding specification for extracting data from a running hybrid prototype could look like that in Figure 3.

```
1  extract to stdout {
2    targets [       /* defines _what_ to output */
3      ACCEL,
4      AP_IF,
5      SOCK_IF with (msg~"I2C*")
6    ]
7    start at (
8      ACCEL.msg~"data $INT:ax$ $INT:ay$ $INT:az$",
9      -5<ax and ax<5 and
10     -5<ay and ay<5 and
11     -5<az and az<5
12   )
13   stop after (6000)
14 }
```

Figure 3: Example of how data extraction can be specified for the "free fall" experiment.

4. THE SystemXtract APPROACH

In recent years, an increasing interest in the process of modeling and simulation has led to domain-specific languages being designed for specifying experiments as well as data extraction *explicitly* [19]. For many years, the problem of how to deal effectively and efficiently with the data produced by simulation has not been given the attention it deserves. Only recently, with the discussion about Big Data and the identification of potential data sources—one of those being simulation [16]—the situation has started to change.

Our approach is to provide a domain-specific language for specifying data extraction in the domain of system-level modeling. In developing the approach, we initially abstracted from technical aspects in favor of focusing on the extraction capabilities and the ease of specifying them. Regarding this, we build on prior work [11, 17, 19]. Therein, we recognize the following common points:

- **P.1** Which data items are of interest (targets) can be defined through constraints on the origin (model entity) and/or other properties of the data items.
- **P.2** How to address targets (referring to the origin) is affected by how the model entities are organized.
- **P.3** When data extraction should be active during simulation can be defined in terms of simulation time and/or observations (events). There may be different activation schemes.

In the present work, we aim at supporting both, the origin- and the property-based definition of targets (cf. P.1). Since we are dealing with system-level models, which are built of named discrete modules, targets are addressable through the modules' names (cf. P.2). This is similar to the filtering as in logging libraries/frameworks discussed in Section 2. In fact, our starting point is that simulation data is available in the form of log entries. Those log entries are typically structured and contain the following information:

- a timestamp,
- a module name or a special category,
- a log level, and
- a log message, possibly containing numerical values.

Thus, *origin-based target definition* refers to the module names or special categories, and *property-based target definition* refers to the log levels and/or the log messages' contents. Referring to activation schemes (cf. P.3), we aim at supporting time-based, event-based, and relative definitions

of when to start and when to end data extraction. With event-based activation, we provide novel support for online debugging, where the phases of interest may depend on external interaction and, thus, are not known in advance (cf. Fig. 1). In certain circumstances, some simulation data might be only available in an unstructured form, partly or entirely missing timestamp, origin, or level information. We want to support the extraction of such unstructured data items as well.

In Section 4.1, we will present a specification language covering the above data extraction capabilities. However, being able to specify data extraction using an appropriate language is one thing, actually extracting simulation data as specified is an other thing. In simulation, data can be viewed as streams, which can be filtered as they are generated [2]. The possibility to reduce the amount of simulation-generated data by prompt filtering motivated the development of stream-based processing architectures such as described in [18]. The main elements of that architecture are processing nodes, called FunctionStreams. FunctionStreams allow to implement arbitrary filter functionality on data streams. The capability of realizing complex filtering by graphs of FunctionStreams leads us to our mapping approach for putting extraction specifications into effect. In Section 4.2, we present how the specification language's elements are mapped to the processing architecture's nodes in order to extract log entries from log streams.

4.1 Specification Language

The SystemXtract language is an *external* domain-specific language, i.e., it does not build upon an existing general host language, what an *internal* domain-specific language would do. External languages benefit from not being bound to the syntactic and semantic provisions of a host language. The design of SystemXtract as an external domain-specific language allowed us to realize a feature-rich language for writing specifications that are supposed to be concise, self-explanatory, clean, and easy to read. In the following, we introduce the SystemXtract language by examples. At the end of this section, the language's comprehensive grammar definition will be given.

A simulation experiment may be executed to answer multiple questions at once. The SystemXtract language supports that by allowing a specification to contain multiple extraction statements. A single extraction statement has the form `extract to` *DEST* `{ ... }`, where for *DEST* one may specify either:

- `file "filename"` for saving the extraction's results to the specified file, or
- `stdout` for writing the results to the system's standard output.

In the statement's body, enclosed by `{` and `}`, one specifies *which* log entries to extract and—optionally—*when* to activate the extraction.

As described before, log entries can be structured or unstructured. Since structured entries are addressable by the origin module's name or a category, the log level, or the log message's content, the language provides corresponding elements to do so (we will describe them soon). Which structured entries to extract is specified within a block headed by the keyword `targets`. The most simple extraction statement, which extracts all structured entries to the system's standard output (stdout), is:

```
1 extract to stdout {
2   targets [ * ]
3 }
```

This example represents a special case, since the `*` sign defines to extract structured entries irrespective of their origin or other properties.

4.1.1 Specification of Targets

To narrow the extraction to log entries from certain origins in the model, one can specify the corresponding names or categories, e.g.:

```
1 extract to stdout {
2   targets [ UNIT1, UNIT2 ]
3 }
```

In this example, all structured log entries from modules named `UNIT1` or `UNIT2` are outputted to stdout. Since unstructured entries can not be addressed in terms of a module name (or other properties), the language provides the keyword `non-targets` for specifying to extract the unstructured entries:

```
1 extract to stdout {
2   non-targets
3 }
```

The specification of `targets` and `non-targets` can be combined, leading to an output in which the temporal order of structured and unstructured log entries is preserved. This can help to interpret unstructured entries in the context of structured entries.

In the `targets` block, which is actually a comma-separated list, each list item specifies a target. A target can be further constrained by specifying a condition using `with (...)`. Herein, one can refer to the log entries' message (using `msg`) or log level (using `lvl`), e.g.:

```
1 extract to stdout {
2   targets [
3     UNIT1 with (msg~"received*"),
4     UNIT2 with (lvl=ERR)
5   ]
6 }
```

This extraction statement specifies to output those log entries that

- originate from the model's `UNIT1` and whose message starts with "`received`" (line 3), or log entries that
- originate from the model's `UNIT2` and whose log level equals `ERR` (line 4).

When specifying a target using `with (msg~"`*STRING*`")`, the *STRING* may contain any number of `*`-signs at any place. A `*`-sign functions as a wildcard for any number of characters, including none. Moreover, *STRING* can include the patterns

- `$INT:`*VAR*`$` for matching an integer value, or
- `$HEX:`*VAR*`$` for matching a hexadecimal value,

where *VAR* can be any non-keyword variable name. The matched value will be bound to the variable. The variable can be used to further constrain the extraction, e.g.:

```
extract to stdout {
  targets [
    UNIT1 with (msg~"received $INT:x$", x < 5)
  ]
}
```

Boolean expressions, such as `x < 5` in the example above, can have any form defined by the rule 'boolexp' of the grammar shown in Figure 4.

When specifying a target using `with (lvl=`*LEVEL*`)`, the log entries' levels are checked for equality with *LEVEL*. Allowing for *equality* checks only makes the language independent from the naming and the order of levels. If—for instance—one wants to extract all log entries of level `WARN` or `ERR`, one may specify:

```
extract to stdout {
  targets [
    * with (lvl=WARN),
    * with (lvl=ERR)
  ]
}
```

This example also shows that targets can be specified without a specific module name or category. This is achieved by using a `*`-sign instead of a module's name or a category.

4.1.2 Specification of Activation

Until now, we presented SystemXtract's language elements for specifying *what* to extract. The foregoing examples lead to log entry extraction being active during the whole run time of the simulation because no activation is specified. As motivated, the language also provides elements to specify *when* to activate and/or deactivate the extraction of log entries. SystemXtract supports various activation schemes covering the most relevant use cases in the domain. One can specify:

- nothing (then, extraction is always active),
- a start point only (extraction is active from then on),
- a stop point only (extraction is active until then),
- a start point and a stop point,
- a start event and a duration afterwards, or
- a duration ahead of a stop event.

A start point is specified by `start at (`*POINT*`)` and a stop point is specified by `stop at (`*POINT*`)`, where *POINT* may be specified in terms of an absolute point in simulation time or in terms of an event. A duration following a start event is specified by `stop after (`*DURATION*`)`, where *DURATION* is an integer meaning units of simulation time. A duration ahead of a stop event is specified by `start ahead (`*DURATION*`)`.

For *POINT* to be an absolute point in simulation time, one specifies `time=`*TIME*, where *TIME* is an integer meaning units of simulation time. For *POINT* to be an event, one can specify one of the following:

- *ID*`.msg~"`*STRING*`"`
- *ID*`.msg~"`*STRING*`",` *BOOLEXP*
- *ID*`.lvl=`*LEVEL*

Instead of *ID*, one can put a `*`-sign to not restrict the `msg` or `lvl` constraint to log entries of a certain module or category otherwise identified by *ID*. An event is considered detected when a log entry matches the event definition. This is analog to matching a target definition. An active phase which is defined through a start and stop event or through a duration is considered to be (re-)entered whenever the start event—or in the case of a duration ahead an event, the stop event—is detected. This allows for extracting log entries from *multiple* instances of a phase of interest.

```
extract to stdout {
  targets [ * ]
  start at (UNIT1.msg~"received $INT:x$", x < 5)
  stop after (100)
}
```

The extraction statement above specifies an active phase that starts whenever the event being defined in line 3 is detected and stops 100 time units afterwards.

4.1.3 Syntax of the SystemXtract Language

Figure 4 shows the grammar of the SystemXtract language in the form of a parsing expression grammar (PEG [7]). For brevity, we omitted to define whitespace (spaces, tabs, and line breaks are accepted between all parts of the lowercase rules) and block comments (enclosed by `/* */`). The parsing expression 'specificatn' is the start expression.

```
specificatn <- statement*
statement   <- 'extract' 'to' output '{' what when '}'
output      <- 'file' FNAME / 'stdout'
what        <- targets nontargets? / nontargets
when        <- startpoint? stoppoint?
               / startpoint 'stop' 'after' '(' PINT ')'
               / 'start' 'ahead' '(' PINT ')' stoppoint
targets     <- 'targets' '[' ('*' / list ) ']'
nontargets  <- 'non-targets'
startpoint  <- 'start' 'at' '(' (time / event) ')'
stoppoint   <- 'stop' 'at' '(' (time / event) ')'
list        <- target (',' target)*
target      <- '*' property / ID property?
property    <- 'with' '(' (message / level) ')'
time        <- 'time' '=' PINT
event       <- ('*' / ID) '.' (message / level)
message     <- 'msg' '~' MSTR (',' boolexp)?
level       <- 'lvl' '=' ID
boolexp     <- val comp val
               / '(' boolexp ')'
               / 'not' boolexp
               / boolexp ('and' / 'or') boolexp
comp        <- '<' / '<=' / '==' / '>=' / '>' / '!='
val         <- VAR / INT / HEX
VAR         <- [a-z]+
ID          <- [a-zA-Z_][0-9a-zA-Z_]*
PINT        <- '0' / [1-9][0-9]*
INT         <- '0' / '-'? [1-9][0-9]*
HEX         <- '0x' [0-9a-fA-F]+
FNAME       <- '"' DRV? [0-9a-zA-Z_/\-.␣]* '"'
DRV         <- [a-zA-Z] ':'
MSTR        <- '"' (!'"' (ESC / MINT / MHEX / .))* '"'
MINT        <- '$INT:' VAR '$'
MHEX        <- '$HEX:' VAR '$'
ESC         <- '\"' / '\$'
```

Figure 4: Grammar of the SystemXtract language.

4.2 Mapping to a Processing Architecture

In the previous section, we presented the syntax of the SystemXtract language and described its semantics. In order

to put specifications into effect on actual log data streams, we map the language's elements to processing nodes of a graph-based architecture presented in [18].

Our approach differs from that of traditional logging libraries such as log4* or CULT. While traditional logging libraries produce log entries depending on decisions made at compile time or decisions made at run time through inserted code, we pursue a filtering approach, making decisions exceptionally at run time. On the one hand, the filtering approach requires to temporarily produce an extensive stream of logs; on the other hand, this approach fosters a clear separation of simulation and data extraction concerns while adding dynamic phase-based activation schemes.

4.2.1 *The Processing Architecture Being Used*

The processing architecture bases on a set of configurable, special purpose node types. Nodes can be interconnected via input and output ports to form a directed acyclic processing graph that realizes the desired processing tasks (cf. Fig. 5).

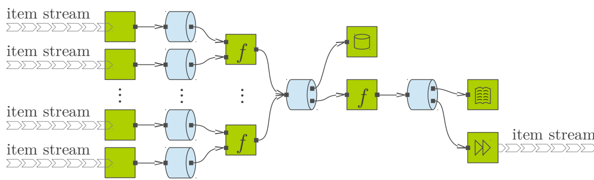


Figure 5: Example of a processing graph. The green boxes represent Transporters (different types); the blue pipes represent Queues.

Within a processing graph, data is uniformly transported and processed in the form of DataItem objects. A DataItem object holds a timestamp plus a Boolean, integer, or decimal value or a string, list, set, map, or object. Data is processed by propagating DataItems from the graph's inputs to the graph's outputs.

The architecture provides so called Transporter node types for input management (InputStream), processing (FunctionStream), and output management (StoragePoint, ReadingPoint, OutputStream). A FunctionStream is configurable with a user defined function (UDF) and as many input and output ports as it needs. A FunctionStream's UDF is applied to the DataItems arriving at the input port(s); the results are passed on via the output port(s).

Additionally to the aforementioned Transporter node types, there is the Queue node type for buffering, merging, and sharing data. Transporters and Queues are interconnected in alternating fashion (cf. Fig. 5). For more details about this architecture, we refer to [18].

With this processing architecture at hand, the task of extracting log entries from a stream of log entries corresponds to *filtering* the stream. Filtering can be realized through FunctionStreams being configured with appropriate UDFs. We implemented UDFs for filtering log entries with respect to their timestamp, origin, level, and message.

To make the filtering more efficient, we created the LogParser UDF that transforms log entries into LogEntry objects. As explained, log entries can be structured or unstructured. If being structured, they have the form:

```
[ 13125000 ns ] INFO: FIFO: new data stored. fill_level=0x9A
```

timestamp — level — origin — message

A LogEntry object representing a structured log entry holds the log level, origin, message, and the original log string in separate fields. A LogEntry object representing an unstructured log entry holds just the original log string. The LogParser UDF packs each LogEntry object together with the timestamp into a DataItem, which can be handled by the processing graph.

4.2.2 *Mapping of Specifications*

As a prerequisite for efficient graph-based filtering of the log entries, the processing graph is set up as shown in Figure 6.

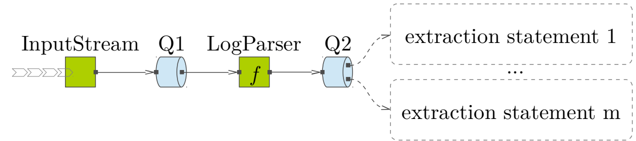


Figure 6: Input side of a processing graph for implementing SystemXtract specifications.

Extraction statements, of which there can be several ones in a specification, are mapped to separate parts of the graph. In the following, we will explain the mapping of *individual* extraction statements.

(a) Outline of an extraction statement:

```
1 extract to DEST {
2   targets [ TARGETS ]
3   non-targets
4   ACTIVATION
5 }
```

(b) Graph structure for an extraction statement:

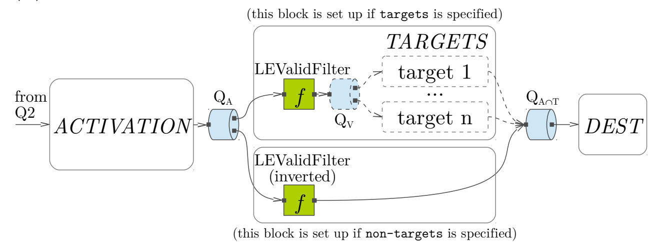


Figure 7: Mapping of an extraction statement to a graph structure. The specification's parts *DEST*, `targets[`*TARGETS*`]`, `non-targets`, and *ACTIVATION* in (a) are mapped to corresponding blocks in (b).

Figure 7a shows the outline of an extraction statement. Note that either the specification of `targets` or of `non-targets` is optional. The part *ACTIVATION* stands for `start` and `stop` specifications, which are also optional as introduced in

Section 4.1.2. According to the language's semantics, only those log entries are passed to *DEST* that appear in a phase defined through *ACTIVATION* and additionally fit to one of the *TARGETS* definitions or to `non-targets`. Using corresponding filter blocks (whose structure will be presented next), an extraction statement can be set into effect by a graph as shown in Figure 7b.

If *DEST* in Figure 7a is specified as `stdout`, the graph's part *DEST* in Figure 7b is built with an OutputStream called LEStdoutWriter. If *DEST* is specified as `file "filename"`, the graph is built with a StoragePoint called LEFileWriter.

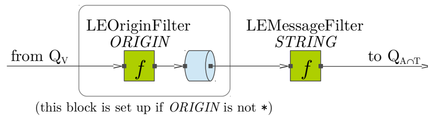
4.2.3 Mapping of Targets

The lines 2 and 3 in Figure 7a are mapped as follows. If `targets[`*TARGETS*`]` is specified, the graph is set up with the upper middle block shown in Figure 7b. If `non-targets` is specified, the graph is set up with the lower middle block shown in Figure 7b. The LEValidFilters at the entrance of those blocks forward only DataItems containing "structured" or "unstructured" LogEntries, depending on being set into normal or inverted mode, respectively.

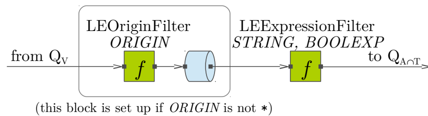
If *TARGETS* is a single `*`-sign, the dotted parts in Figure 7b are omitted; the LEValidFilter is then directly linked to $Q_{A \cap T}$. If *TARGETS* is a list of target specifications, each of those is mapped to a separate filter block as shown in Figure 7b. Because of the or-semantics of the targets list, the filter blocks are set up in parallel; their results are merged by $Q_{A \cap T}$. Each individual target filter block is built up as shown in Figure 8a–d.

(a) *ORIGIN*

(b) *ORIGIN* `with (msg "`*STRING*`")`



(c) *ORIGIN* `with (msg "`*STRING*`",` *BOOLEXP*`)`



(d) *ORIGIN* `with (lvl=`*LEVEL*`)`

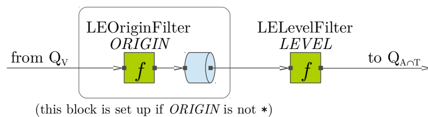


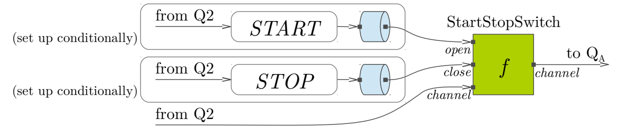
Figure 8: Graph structures for different forms of individual target specifications.

The LEOriginFilter, LEMessageFilter, LEExpressionFilter, and LELevelFilter forward only those DataItems whose LogEntry's origin, message, in-message values, or log level match the filter's parameters; unmatched DataItems are absorbed. Thus, the shown filter chains put the combined target constraints into effect correctly.

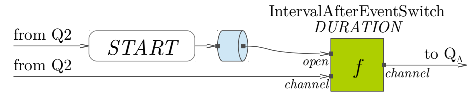
4.2.4 Mapping of Activation Schemes

The placeholder *ACTIVATION* in Figure 7a stands for one of various activation schemes (cf. Section 4.1.2). Technically, we differentiate schemes using start/stop points from schemes using a duration relative to a start/stop event. The activation schemes are mapped as shown in Figure 9a–c.

(a) `start at (`*START*`)  stop at (`*STOP*`)`



(b) `start at (`*START*`)  stop after (`*DURATION*`)`



(c) `start ahead (`*DURATION*`)  stop at (`*STOP*`)`

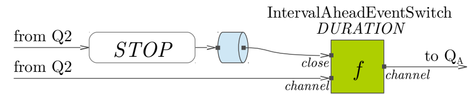


Figure 9: Graph structures for different activation schemes.

The behavior of the StartStopSwitch, IntervalAfterEventSwitch, and IntervalBeforeEventSwitch corresponds to that of a channel that can be opened or closed, hence the names of the ports. The StartStopSwitch opens and closes upon arrival of a DataItem at the corresponding port; repeatable opening is activated if *START* and *STOP* are both specified as events. The IntervalAfterEventSwitch opens whenever a DataItem arrives at the *open* port and closes its channel for DataItems with a timestamp greater than opening time + *DURATION*. The IntervalBeforeEventSwitch utilizes the channel's upstream Queue to buffer a *DURATION*-dependent amount of DataItems that are released whenever a DataItem arrives at the *close* port.

Specifications of *START* or *STOP* through `time=`*TIME* are mapped to a FunctionStream with a TimeCrossingDetector UDF. A TimeCrossingDetector, which has one input and one output port, passes only the first DataItem whose timestamp crosses the UDF's *TIME* parameter.

Specifications of *START* or *STOP* in the form of events are mapped in the same way as an individual target specifications (cf. Fig. 8). This is possible because event detection is semantically equivalent to target filtering.

5. EXPERIMENTS WITH SYSTEMXTRACT

Nowadays, system-level models can be used to support the whole specification, validation, and verification process in developing complex embedded systems. The functional representation of a system, implemented in SystemC as an ex-

ecutable specification, can help making design decisions by simulating different use cases. Furthermore, an executable specification can act as a reference model in verification [10].

In the case of designing novel processing subsystems around existent sensors, combining a simulation model of the processing subsystem with the physical sensors allows experiments under real world inputs. In Section 3, we gave an example where a hybrid prototype with an acceleration sensor is used for investigating the prototyped system's behavior in free fall situations. A similar prototype is presented by Stieber et al., who also discuss how to achieve proper time synchronization between the physical and the simulated system components [20]. A more detailed view on the hybrid prototype developed by Stieber et al. is given in Figure 10.

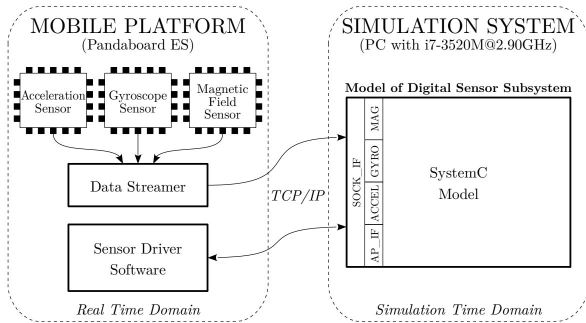


Figure 10: Schematic of a hybrid prototype used in the development of an inertial sensor system.

5.1 Experiment Setup

The prototype depicted in Figure 10 is used for developing the sensor driver software which will later control a new sensor system. The driver software (lower left in Fig. 10) runs on a mobile platform, which in this case is a Pandaboard ES running Android 4. The Pandaboard also hosts an acceleration sensor, a gyroscope sensor, and a magnetic field sensor (upper left in Fig. 10), which provide measurements via the board's I^2C-bus. The measurements are taken up by a data streamer program on the Pandaboard (left in Fig. 10) and streamed to the simulation system, where the digital sensor subsystem—later to be implemented on-chip—is simulated with SystemC (right in Fig. 10). The Pandaboard and the simulation system communicate via local link Ethernet in a transaction-based I^2C protocol over TCP/IP.

The SystemC model of the digital sensor subsystem consists of different modules, each simulating the functionality of a certain component of the sensor system. Three of those components are the sensors at the Pandaboard, which are represented in the model by modules called `ACCEL`, `GYRO`, and `MAG` (cf. Fig. 10). An other system component is the power management unit, which is represented in the model by a module called `PMU` (not shown in Fig. 10). There are also a `SOCK_IF` (for socket interface) module handling the TCP/IP communication and an `AP_IF` module representing the application interface.

To use the hybrid prototype for investigating the developed system's internal behavior, the SystemC model has been instrumented with logging code, which produces messages at different log levels. During experiments, a lot of those log messages provide information about sampled sensor data (from `ACCEL`, `GYRO`, and `MAG`) and internal state changes such as power mode switches (in the `PMU` module). Further log messages inform about intermediates and results of the modeled gesture detection algorithms and sensor data filters. Additionally, log messages inform about inter-module communication (e.g., via the `SOCK_IF` and `AP_IF` modules), internal transactions, configuration changes, performance values, and errors. Next to the message, log entries include a timestamp, a module name, and a log level. The format of those entries was described in Section 4. For immediate inspection at experiment run time, the log entries are streamed to the simulation system's standard output. The built-in logging is configurable to a certain extent, i.e., log message generation can be restricted to certain log levels. Thus, the experimenter has to choose between fully enabled logging, producing vast data volumes, and restricted logging, potentially missing important information.

5.2 The SystemXtract Tool

For targeted log data extraction with SystemXtract, we developed a Java application (in the following called the tool), implementing the presented approach. Due to a requirement of the particular application context, the SystemC model is kept independent of the SystemXtract tool. That is, the pre-instrumented model just streams log entries via its standard output and the tool reads log entries via its standard input. The connection between the executed model and the SystemXtract tool is established by a software pipeline—in our case a UNIX pipe: <simulation> | <tool>. Figure 11 gives an overview of the tool's components and chain of control.

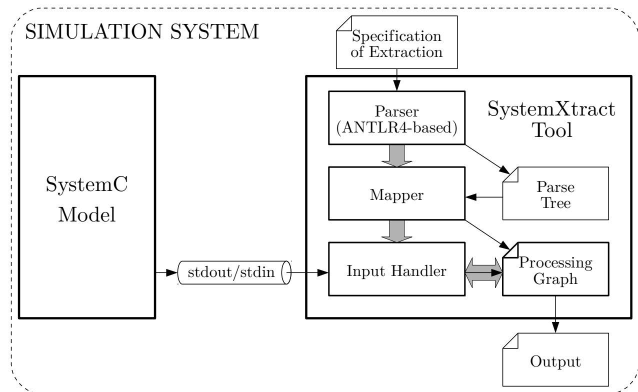


Figure 11: Application of the SystemXtract tool. Wide arrows show the chain of control; thin arrows show the flow of information.

The SystemXtract tool works as follows (cf. Fig. 11). After being called with a specification's filename as parameter, the tool's parser reads the specification and generates a parse tree. Figure 12 shows a specification used in experiments with the hybrid prototype of Section 5.1. Then, the tool's mapper transforms the parse tree into a processing graph—this is done as described in Section 4.2. The processing graph for the specification in Figure 12 is shown in Figure 13. Having built up the processing graph, the Sys-

```
1  extract to file "log.txt" {
2    targets [
3      ACCEL,
4      GYRO,
5      MAG,
6      AP_IF,
7      SOCK_IF with (msg~"I2C*")
8    ]
9    start at (PMU.msg~"Chip status = 2")
10   stop at (PMU.msg~"Chip status = 0")
11 }
```

Figure 12: An extraction specification referring to the components of the SystemC model in Figure 10.

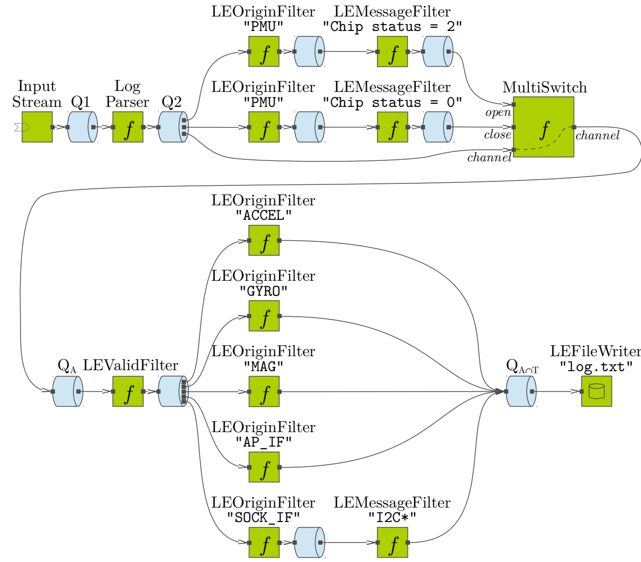


Figure 13: Processing graph generated from the specification shown in Figure 12.

temXtract tool is ready for filtering log entries. To do so, the data handler continuously reads log entries (each is a line of text) from the standard input, wraps them as DataItems and hands them over to the InputStream of the previously generated processing graph. After processing a DataItem, the processing graph processes returns control to the data handler. The data handler reads and forwards log entries to the processing graph until the tool's execution is terminated.

5.3 Experiments and Results

To verify that our model-independent filter-based extraction approach does not induce an overhead in computation time (in the following: overhead) that compromises the hybrid prototype's real-time execution, we performed the following experiments:

E.1 The hybrid prototype is executed using the built-in logging only (configured for "full" logging).

E.2 The hybrid prototype is executed with "full" logging. The log output is piped to the SystemXtract tool being configured using the specification:
`extract to file "log.txt" { targets [ * ] }`

E.3 The hybrid prototype is executed with "full" logging. The log output is piped to the SystemXtract tool being configured using the specification shown in Figure 12.

Since the simulation of the digital sensor subsystem, which runs faster than real-time, is synchronized to the sensors' 625 μs sampling interval (cf. [20]), the overall run time of an experiment can not be used for rating the overhead induced by the data extraction tool. Instead, we derive the overhead from the average total sleep-time spent for synchronizing to the 625 μs intervals. Taking the average total sleep-time determined by experiment E.1 as a baseline allows us to rate the tool-induced overhead in E.2 and E.3.

To be able to relate the experiments' results to each other, we fixed the experiment run time (which includes short start-up phase for the simulation and the hardware sensors) to 3 minutes and repeated each experiment 5 times. The results are shown in Table 1.

Table 1: Experimentation results.

Exp.	sleep-time per interval	# logs	disk space
E.1	503.59 μs ($\sigma = 2.76\,\mu s$)	537850	31.28 MB
E.2	502.19 μs ($\sigma = 1.02\,\mu s$)	529982	30.67 MB
E.3	500.80 μs ($\sigma = 1.84\,\mu s$)	67161	4.91 MB

The experiments E.1 and E.2 with "full" logging show that the simulation produces about 3000 log entries per second. In experiment E.1 without SystemXtract, the simulation sleeps 503.59 μs per 625 μs interval (80.6% of the execution time), i.e., 121.41 μs are used for computing the simulation and outputting log entries. In experiment E.2 with SystemXtract, the sleep-time decreases by 0.2% relative to E.1. This means that the computation time for simulation, log generation, *and* graph-based processing/output increases just by 0.2% relative to E.1. The log data volume is—as expected—nearly the same (unstructured log entries, which are filtered out in E.2, account for less than 1.5% of all log entries). In E.3, with SystemXtract and the specification shown in Figure 12, the computation time for simulation, log generation, *and* graph-based processing/output increases just by 0.4% relative to "full" logging in E.1. Coming at this very low overhead cost, the number of log messages to be sighted by the experimenter is decreased to approximately 12.5% relative to the "full" logs!

6. CONCLUSIONS

To support the debugging of hybrid prototypes we developed the language SystemXtract for specifying *what* to extract (i.e., which log entries) and *when* to extract (i.e., in which phases). The language is mapped to a graph-based stream processing architecture for an effective targeted on-line data extraction. To test the developed tool, we selected a project, in which digital subsystems of "intelligent" inertial sensors, e.g., for smartphones, are developed. The number of log entries (about 3000 produced per second), hours lasting tests, as well as real-time constrains made it a perfect case study for the developed tool. Our first tests show,

- that the embedded systems' developer had no difficulties to specify the constrains for narrowing down the potentially interesting data,
- that the data could be significantly reduced, and
- that the on-line filtering by the stream processing architecture added only a negligible overhead on the simulation (about 0.4 % in our studies).

Particularly, the possibility of event-driven activation and deactivation of the data extraction proved useful in reducing the risk of missing important data due to too restrictive filtering or too cluttered outputs. In comparison to other approaches for data logging, such as CULT, our approach provides more flexibility in expressing constrains that are executed on the fly. Those constrains refer to *what* data to extract, e.g., by string matching and Boolean expressions, and *when* to extract those data, e.g., by event-driven activation and deactivation of data extraction. In addition, due to being a domain-specific language that does *not* base on XML, SystemXtract allows for more compact specifications (lines of code). However, it should be noted that SystemXtract does not address the problem of how to instrument the hybrid prototype—SystemXtract works on the data that is already outputted by the hybrid prototype.

Future work will be dedicated to extending the language such that activation/deactivation can be triggered not only by simple events referring to single log entries, but also by complex events, putting (temporal) constraints on the sequence of log entries. Further work may be dedicated to a hybrid approach, producing a minimum of temporary log data by more efficient static logging, while applying graph-based filtering only for dynamic phase-based extraction.

7. ACKNOWLEDGMENTS

Johannes Schützel is funded by the German Research Foundation (DFG) through GRK 1424.

8. REFERENCES

[1] IEEE Standard for Standard SystemC Language Reference Manual. *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, pages 1–638, Jan. 2012.

[2] G. Abdulla, T. Critchlow, and W. Arrighi. Simulation data as data streams. *SIGMOD Rec.*, 33(1):89–94, Mar. 2004.

[3] B. Bakker and Log4cpp Contributors. Log for C++ project. `http://log4cpp.sourceforge.net/`. Accessed on February 2nd, 2015.

[4] Bosch Sensortec. Data sheet: BNO055 — Intelligent 9-axis absolute orientation sensor. `http://ae-bst.resource.bosch.com/media/products/dokumente/bno055/BST_BNO055_DS000_12~1.pdf`, 2014.

[5] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, June 2008.

[6] D. D. Chamberlin and R. F. Boyce. SEQUEL: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74, pages 249–264, 1974.

[7] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 111–122, 2004.

[8] C. L. Goater. Log4c: Logging for C Library. `http://log4c.sourceforge.net/`. Accessed on February 2nd, 2015.

[9] C. Gülcü. *The complete Log4j manual: The reliable, fast and flexible logging framework for Java.* QOS.ch, Lausanne, Switzerland, 2003.

[10] C. Haubelt, J. Teich, and R. Dorsch. Entdecke die Möglichkeiten. *Design&Elektronik*, 8:22–27, 2008.

[11] T. Helms, J. Himmelspach, C. Maus, O. Röwer, J. Schützel, and A. M. Uhrmacher. Toward a language for the flexible observation of simulations. In *Proceedings of the 2012 Winter Simulation Conference*, WSC '12, pages 418:1–418:12, 2012.

[12] W. Hong, J. Joshi, A. Viehl, N. Bannow, A. Kramer, H. Post, O. Bringmann, and W. Rosenstiel. Advanced features for industry-level logging and tracing of C-based designs. In *2013 Forum on Specification Design Languages (FDL)*, 2013.

[13] W. Hong, A. Viehl, N. Bannow, C. Kerstan, H. Post, O. Bringmann, and W. Rosenstiel. CULT: A unified framework for tracing and logging C-based designs. In *System, Software, SoC and Silicon Debug Conference (S4D)*, 2012.

[14] L. Murphy, G. Lewandowski, R. McCauley, B. Simon, L. Thomas, and C. Zander. Debugging: the good, the bad, and the quirky—a qualitative analysis of novices' strategies. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '08, pages 163–167, 2008.

[15] P. Panda. SystemC—a modeling platform supporting multiple design abstractions. In *Proceedings of the 14th International Symposium on System Synthesis*, pages 75–80, 2001.

[16] M. Parashar. Big data challenges in simulation-based science. In *Proceedings of the Sixth International Workshop on Data Intensive Distributed Computing*, DIDC '14, pages 1–2, 2014.

[17] J. Schützel, R. Ewald, and A. M. Uhrmacher. Towards a general foundation for formalism-specific instrumentation languages. In *Proceedings of the 2013 Winter Simulation Conference*, WSC '13, pages 4008–4009, 2013.

[18] J. Schützel, H. Meyer, and A. M. Uhrmacher. A stream-based architecture for the management and on-line analysis of unbounded amounts of simulation data. In *Proceedings of the 2nd ACM SIGSIM/PADS Conference on Principles of Advanced Discrete Simulation*, SIGSIM-PADS '14, pages 83–94, 2014.

[19] J. Schützel, D. Peng, A. M. Uhrmacher, and L. F. Perrone. Perspectives on languages for specifying simulation experiments. In *Proceedings of the 2014 Winter Simulation Conference*, WSC '14, pages 2836–2847, 2014.

[20] S. Stieber, J. Wolff, C. Haubelt, and R. Dorsch. Hybride Prototypisierung eines Sensorsubsystems. In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, MBMV 2014, pages 209–212, 2014.

[21] The Apache Software Foundation. Apache Log4cxx. `http://logging.apache.org/log4cxx/`. Accessed on February 2nd, 2015.

[22] The Apache Software Foundation. Apache Log4j. `http://logging.apache.org/log4j/2.x/`. Accessed on February 2nd, 2015.