
	FP7-ICT-2013- 10 (611146)		CONTREX
	Design of embedded mixed-criticality CONTRol systems under consideration of EXtra-functional properties		
Project Duration	2013-10-01 – 2016-09-30	Type	IP

	WP no.	Deliverable no.	Lead participant
	WP3	D3.4.3	POLITO
Implementation of extra-functional Property monitoring (final)			
Prepared by	Massimo Poncino, Andrea Acquaviva, Enrico Macii, Alberto Macii, Andrea Calimera, Sara Vinco (POLITO), Gianluca Palermo, Carlo Brandolese (POLIMI), Alexander Agethen (IX), Fernando Herrera (UC), Paolo Azzoni, Matteo Maiero (EUTH), Philipp A. Hartmann, Sven Rosinger, Kim Grüttner (OFFIS) John Favaro (INTECS), Sylvain Kaiser (Intel), Luca Ceva (VA).		
Issued by	POLITO		
Document Number/Rev.	CONTREX/POLITO/R/D3.4.3/1.5		
Classification	CONTREX Public		
Submission Date	2016-09-30		
Due Date	2016-09-30		
Project co-funded by the European Commission within the Seventh Framework Programme (2007-2013)			

© Copyright 2016 OFFIS e.V., STMicroelectronics srl., GMV Aerospace and Defence SA, Vodafone Automotive SpA, Eurotech SPA, Intecs SPA, iXtronics GmbH, EDALab srl, Docea Power, Politecnico di Milano, Politecnico di Torino, Universidad de Cantabria, Kungliga Tekniska Högskolan, European Electronic Chips & Systems design Initiative, ST-Polito Società consortile a r.l., Intel Corporation SAS.

This document may be copied freely for use in the public domain. Sections of it may be copied provided that acknowledgement is given of this original work. No responsibility is assumed by CONTREX or its members for any application or design, nor for any infringements of patents or rights of others which may result from the use of this document.

History of Changes

ED.	REV.	DATE	PAGES	REASON FOR CHANGES
MP	0.1	2016-07-13	6	Skeleton prepared
SV	0.2	2016-09-01	14	Added POLITO contribuion
RG	0.3	2016-09-12	18	Added OFFIS contribution
SK	0.4	2016-09-15	23	Added INTEL contribution
AG+MP	1.0	2016-09-22	24	Added iXtronics Contribution, re-organization of the documenti
MP	1.1	2016-09-23	25	Rewriting of Sections 1 and 2 in a more self-consistent way.
PA	1.2	2016-09-23	37	Added ETH Contribution
CB	1.3	2016-09-23	55	Added PoliMi contribution
FH	1.4	2016-09-27	71	Added UC Contribution
MP	1.5	2016-09-28	74	Final polishing

Contents

1	Introduction	4
2	Overview and Mapping on the Estimation Flows	5
2.1	Mapping on the EFP Estimation Flow	6
3	Implementation of extra-functional properties monitoring - Individual partner contributions.....	8
3.1	Stream-based simulation and tracing framework [OFFIS].....	8
3.1.1	Extensions of the existing tracing framework.....	8
3.1.2	Implementation of framework and library	11
3.2	Monitoring performance metrics through parallel native simulation [UC].....	12
3.2.1	Status before CONTREX	12
3.2.2	Status after CONTREX	13
3.3	Extra-functional monitoring at the system-level [iX].....	27
3.4	Application-Level light-weight extra-functional monitoring [POLIMI].....	28
3.4.1	Framework Overview	28
3.4.2	Monitor Metamodel.....	29
3.4.3	Metamodel example	31
3.4.4	Framework configuration	31
3.4.5	Framework C interface.....	37
3.5	Sensor modelling and monitoring [INTEL]	45
3.6	Battery modelling and monitoring [PoliTo]	48
3.7	Extra-functional monitoring at the cloud level [EUTH]	61
3.7.1	Kura pervasive monitoring	61
3.7.2	The telemetry protocol	65
3.7.3	Connecting to the cloud.....	70
3.7.4	The monitoring methodology	72
4	Conclusions	73
	References	74

、

1 Introduction

Scope of this deliverable is to provide the first report on the final **implementation** of the monitoring of the extra-functional properties of interest. This deliverable is the follow-up of D3.4.2, in which a preliminary implementation (or in some cases, a refinement of the specifications given in D3.4.1) of the various approaches envisioned in CONTREX for the monitoring extra-functional properties were surveyed.

However, since this is the first and only public deliverable describing the activities of Task 3.4, we have written it in a self-consistent way so that the general CONTREX approach for property monitoring can be inferred by reading this document only. For this reason, the deliverable contains some re-use of the material of the previous deliverables (and in particular D3.4.2) rather than using an "incremental" style of presentation.

The deliverable is organized into two main sections. Section 2 recaps the overall approach for extra-functional property monitoring as described in D3.4.2 and maps the contributions of the partners of Task 3.4 onto the two extra-functional property estimation flows defined in D3.1.1 and applied to the three use cases. Section 3 contains the details of the per-partner activities described in Section 2.

2 Overview and Mapping on the Estimation Flows

Figure 1 shows the general role of extra-functional (EFP) property monitoring in the overall CONTREX scenario and its relation with the execution platform and the relative models.

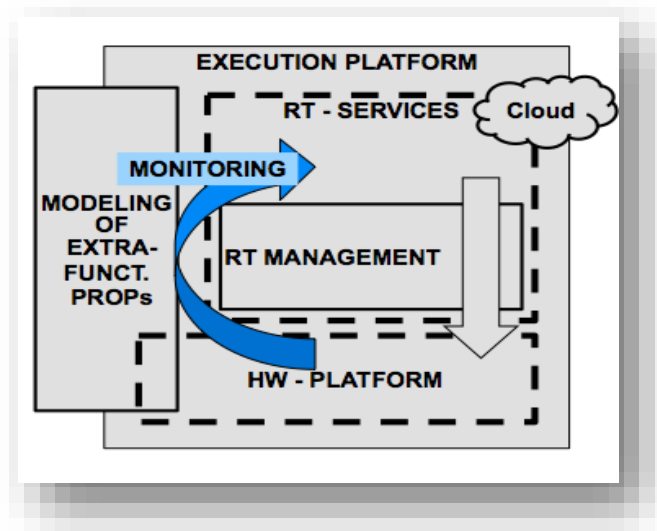


Figure 1. Role of extra-functional properties monitoring.

The general EFP monitoring paradigm that was defined in this task (also based on the specifications presented in D3.4.1) relies on the use of **virtual sensors**, i.e., explicitly (virtual) devices instantiated in the system description at the (SoC) component level; these virtual devices extract the quantity (EFP) of interest and abstract it to higher levels. This approach is simple and flexible at the same time because it is potentially applicable to any EFP, which is “measurable”. The abstraction is implemented **via EFP traces/models to be plugged into functional models properly augmented to accommodate for new metrics**. This is conceptually depicted in Figure 2.

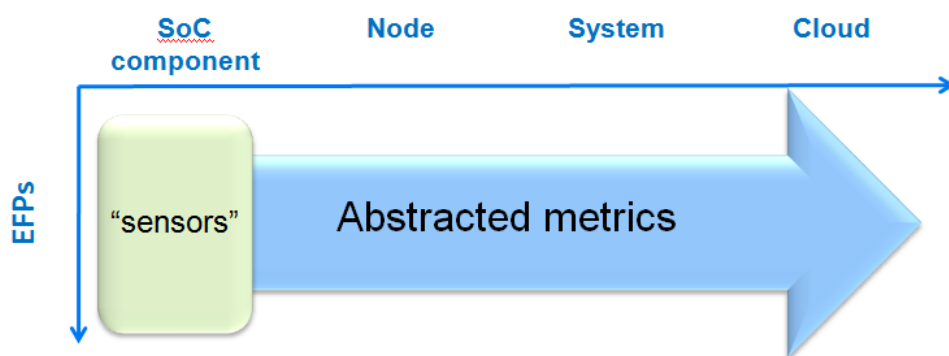


Figure 2. Virtual-sensor paradigm concept for EFP monitoring.

Within this virtual-sensor paradigm, the implementations carried out by the various partners have followed two main variants:

1. An **off-line approach** based on traces derived from functional simulation by monitoring specific quantities that can be related through specific, typically **stateless, models** to the EFP of interest. By eventually running additional simulation runs using these models, the values of the EFPs can be extracted. This approach is generally faster (depending on the complexity of the models) but generally less accurate.

2. An **online approach**, in which the EFPs to be observed **are directly modeled (using state-based models) in the system description**. In other terms, functional and extra-functional behavior is modeled using (possibly) the same language and using a single simulation run. This approach is in generally slower than the off-line one, but guarantees better accuracy. Moreover, as the number of distinct EFPs increases, the online approach becomes more scalable since only a single simulation run is always needed, whereas in the offline one n runs are required for n distinct EFPs.

Although in the first part of the project both approaches have been carried by different partners (as described in D3.4.1), the activities in the task have **converged towards the use of the online approach**, mostly due to (i) its better scalability and (ii) possibility of using an homogeneous description of the system and the EFPs.

2.1 Mapping on the EFP Estimation Flow

As described in D3.1.1, **two distinct EFP estimation flows** are envisioned, depending on the use cases. The implementations of the EFP monitoring described in this deliverable cover different parts of the EFP estimation flows. In the rest of this section, we will show how the various contributions implement this coverage. This mapping represents the evolution of the preliminary specification for EFP monitoring reported in D3.4.1; in that document, the mapping was rather a generic classification of the EFP monitoring approaches with respect to the various abstraction levels (from SoC to cloud). When the implementation was started, the actual mapping became well-defined and it was possible to precisely position all the EFP monitoring "engines" in the most appropriate places in the flow. This step was firstly described in D3.4.2.

Figure 3 depicts the EFP estimation flow applicable to **UC1 and UC3** and the parts covered by the contributions of OFFIS (Section 3.1), UC (Section 0), and iX (Section 3.3)

Figure 4 shows the EFP estimation flow for **UC3** and the sections covered by the contributions of POLIMI (Section 3.4), DOCEA (Section 3.5), and POLITO (Section 0).

The contribution of EUTH, which concerns the cloud-level monitoring, does not fit into the two flows since the cloud actually overlaps with the execution platform (as shown in Figure 1), and is therefore treated separately in Section 3.7

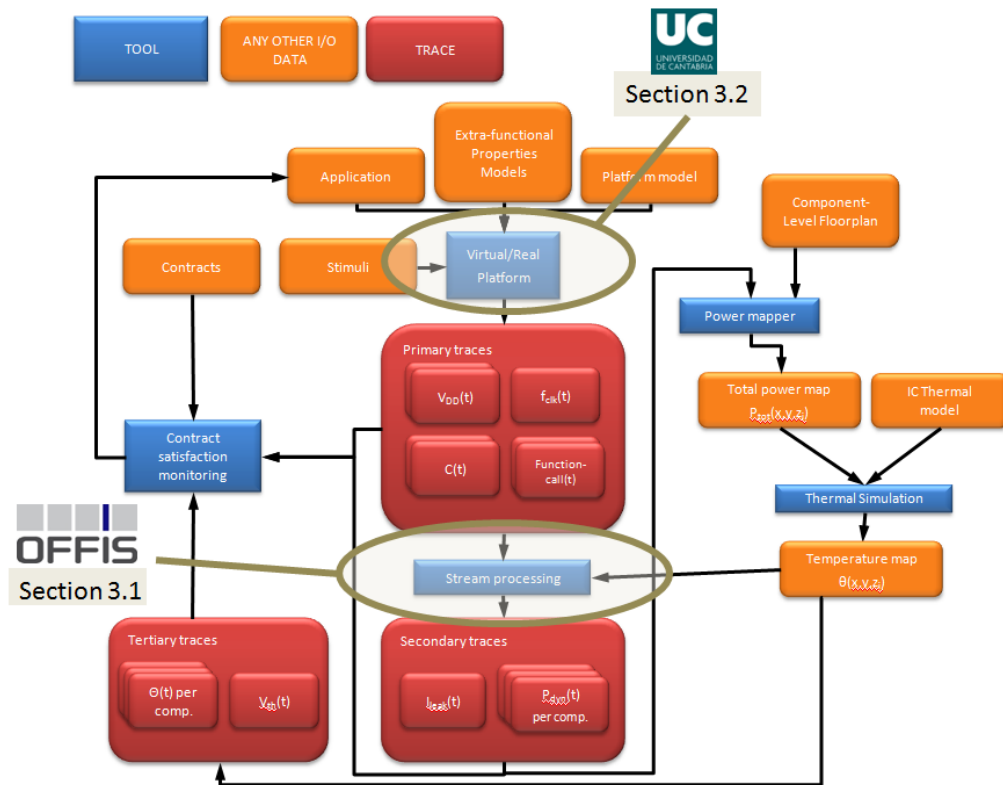


Figure 3. Mapping of EFP Monitoring on the EFP Estimation Flow of UC1 and UC3.

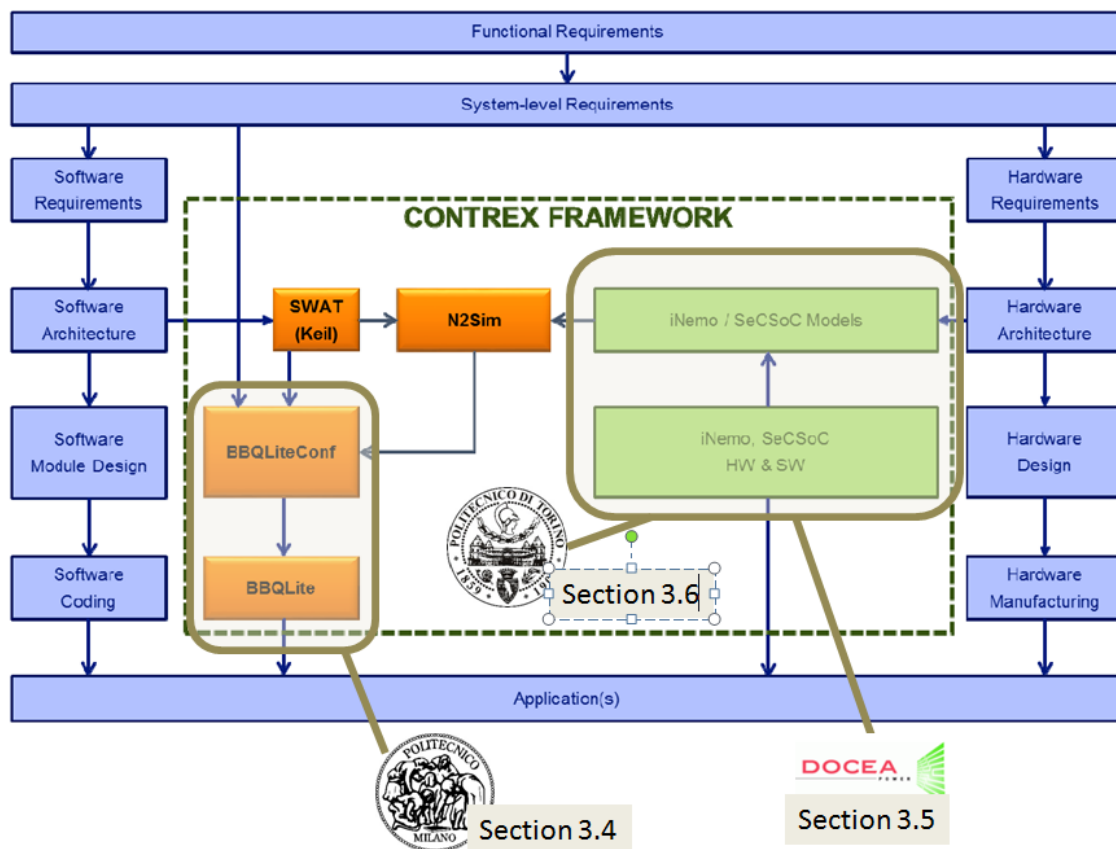


Figure 4. Mapping of EFP Monitoring on the EFP Estimation Flow of UC2.

3 Implementation of extra-functional properties monitoring - Individual partner contributions

3.1 Stream-based simulation and tracing framework [OFFIS]

The goal of the stream-based simulation and tracing framework developed by OFFIS within the CONTREX project is to provide the enabling simulation technology for seamless specification, analysis, and verification of multiple applications running on complex platforms considering extra-functional properties. By sharing the same computing platform, multiple applications can interfere through extra-functional properties (power, energy, temperature; see Figure 1).

The analysis of such parasitic interference requires the correct attribution of platform resource usage across all service layers of the platform (processing elements, software runtime layers) to the currently running application for analysis and verification.

The underlying models for the extra-functional properties of the various platform elements are defined in D3.1.1. In this section, a composable simulation framework for the simulation of these extra-functional property models running in parallel to the functional behaviour as part of the same virtual platform simulation is presented.

In order to address the need to support multiple applications and the attribution of their induced platform activities across the various hierarchical service layers (as defined in D3.2.1, but with a focus on the on-chip services), several extensions to the original COMPLEX model have been identified and are being developed by OFFIS within the CONTREX project.

3.1.1 Extensions of the existing tracing framework

The basic idea of the stream-based simulation and tracing framework was presented in D3.4.1. Timed streams enable the temporal decoupled tracing of any quantity especially including physical quantities for tracing extra-functional properties. Therefore, different semantics of streams have to be considered and the timed stream including, timed writer, and timed reader have to be adapted to these semantics. Physical quantities emerge as both state quantities and process quantities.

State Quantities describe the state of a system at a given time. Without external influences, this state does not change. In the context of system-level modelling, this can include extra-functional properties like the cache hit/miss rates, power consumption, (ambient) temperature and of course the functional state. If tuples of a timed stream have to be split in two, they can simply be split into separate tuples with the same value and the same total duration. Empty periods can be completed by extending the previous tuple in the stream. A reduction of the stream length can be performed by joining consecutive tuples with the same value without loss of information.

Process Quantities describe a state change with an associated duration. Examples include the amount of cache hits/misses (in contrast to the hit/miss rate), the energy needed for a dynamic frequency/voltage switch, or number of transferred bytes in a transaction. Timed stream tuples of such quantities cannot be split, joined or completed in the same way as state quantities. Instead, splitting requires the distribution of the state change into two (or more) intermediate steps with a combined value equivalent to the original value. Empty periods can be filled with

a dedicated “silence value” (usually 0) and a lossless reduction of the stream length is not possible without reducing accuracy of the temporal resolution.

In D3.4.1 it was explained that if multiple SystemC processes push into the same stream it may come to conflicting pushes if they overlapping intervals. In this case, a conflict resolution happens in terms of a `MergePolicy` Template. This behaviour was already explained in detail in D3.4.1. In this context also the `SplitPolicy` was mentioned which splits a (value, duration) tuple at a defined offset. Since several further policies were introduced in the past, instead of multiple policy templates a single traits template is used to define the behaviour of the stream. This traits template manages the behaviour of the `MergePolicy`, `SplitPolicy`, `JoinPolicy`, `EmptyPolicy`, and `ZeroTimePolicy`.

- **MergePolicy**

As already defined, the merge policy defines the conflict resolution behaviour in case of multiple pushes with overlapping intervals of the tuples. In all cases (except in case of errors), the overlapping tuples are split into disjoint duration according to the `SplitPolicy`. For the conflicting cases either one of the conflicting values can be taken (`timed_override_policy` for the last value or `timed_discard_policy` for the first value) or an (potentially user-defined) arithmetic operation can be executed on the values. The arithmetic operation may be the accumulation or averaging of the values but also more complex operations. The behaviour strongly depends on the stream semantics and thus may be defined by the user, too.

- **SplitPolicy**

The `SplitPolicy` like the `MergePolicy` strongly depends on the stream semantics. For state-based streams it suffices in the most cases if the split tuples get the values of the origin tuple. The state holds for the complete interval of the tuple. Therefore, the split values can be assigned with the origin values without changing the result. For process quantities, the splitting process is more complex. Process quantities represent an ongoing consumption or processing of values. This means that a tuple of value and duration describe that in the time interval of duration the amount of value is consumed or processed, respectively. However, it is not known in which way the amount is consumed. For instance, the complete amount could be consumed at begin, at the end, or equally distributed over the interval (linear time dependent). Furthermore, there could be an exponential dependency or a stepwise consumption. Only the provider of stream (and thus the user, too) knows the exact semantic of the process. For that reason, the split tuples cannot get the values of the origin tuple because this leads to a duplication of values in the same interval and thus a modification of the stream. However, a split must never change the stream. Therefore, the `SplitPolicy` has to be adapted depending on the distribution of fractions of the overall amount in the interval. If the value evolves linearly over the interval, the tuple can be split linearly to the points in time where the splits occur. For example, if a tuple is split in the ratio 70% to 30% then the value is split into fractions of 70% to 30% and assigned to the two new tuples. Other `SplitPolicies` can be used in timed streams to adapt this behaviour to the evolution or distribution of values in a tuple.

- **JoinPolicy**

When pushing new tuples into a stream it may happen that two consecutive tuples have the same value. In this case, it may be desired that the two values are merged into one tuple with the aggregated intervals of both origin tuples. For the `JoinPolicy` the rules are the same as for the `SplitPolicy`. If two tuples of a state quantity are merged, the resulting merged tuple gets the value of one origin tuple (it does not matter which one is taken because both are the same). If two tuples of a process quantity are joined, the resulting tuple has again the accumulated interval of the origin tuples and also the accumulated value of the origin tuples. If no joining is desired the tuples stay unchanged. For complex stream semantics, also user-defined `JoinPolicies` can be defined and applied.

- **EmptyPolicy**

Streams consist of a list of tuples that denote a value and a duration. Therefore, the relative point in time of a tuple is given by the accumulated interval of all preceding tuples. When pushing tuples with an offset, this can lead to empty intervals. Since no empty intervals are possible in timed streams, this empty interval is filled by a *silence value* provided by the `EmptyPolicy`. Again, this value strongly depends on stream semantics. In most cases, this value behaves like a neutral element in the value range when merging states. For example, if merging means accumulating the values, the returned value by the `EmptyPolicy` would be 0. For a stream of process states, the silence value would be the *idle* state.

- **ZeroTimePolicy**

Sometimes it happens that tuples with a zero time duration are pushed in the stream. This happens when the initiating process changes states without let time pass. In this case, this value may be deleted by the stream if a zero duration makes no sense for the stream semantics. In the other case, the value stays in the stream and is forwarded to the reader. It has to paid attention when accessing tuples with zero duration. For example if the preceding tuple's duration is until time t and the reader reads until time t , the tuple with the zero time duration is not included. The zero time duration is included not before reading until a time $> t$.

A traits template for process semantics could be defined as follows:

```
template<typename ValueType> struct timed_process_traits {
    typedef ValueType value_type;
    // provide default value for empty periods
    typedef timed_silence_policy<value_type>empty_policy;
    // distribute alues proportionally to duration
    typedef timed_divide_policy<value_type>split_policy;
    // keep tuples separate
    typedef timed_separate_policy<value_type>join_policy;
    // resolve conflicts by accumulating values
    typedef timed_accumulate_policy<value_type>merge_policy;
    // keep zero time tuples
    typedef timed_keep_policy<value_type>zero_time_policy;
}; // timed_process_traits<ValueType>

timed_stream<energy_quantity, timed_process_traits>energy_stream;
timed_stream<power_quantity>power_stream; // state traits by default
```

As can be seen, the state traits are used by default and therefore do not have to be given when creating the stream.

3.1.2 Implementation of framework and library

The implementation and documentation is now complete. A detailed documentation of the library and API can be found in Attachment A. The publication of the library implementation as an open source project is in progress. We are currently evaluating different open source licenses and community platforms such as GitHub to share the project. Our goal is to complete the publication of the library by end of 2016.

3.2 Monitoring performance metrics through parallel native simulation [UC]

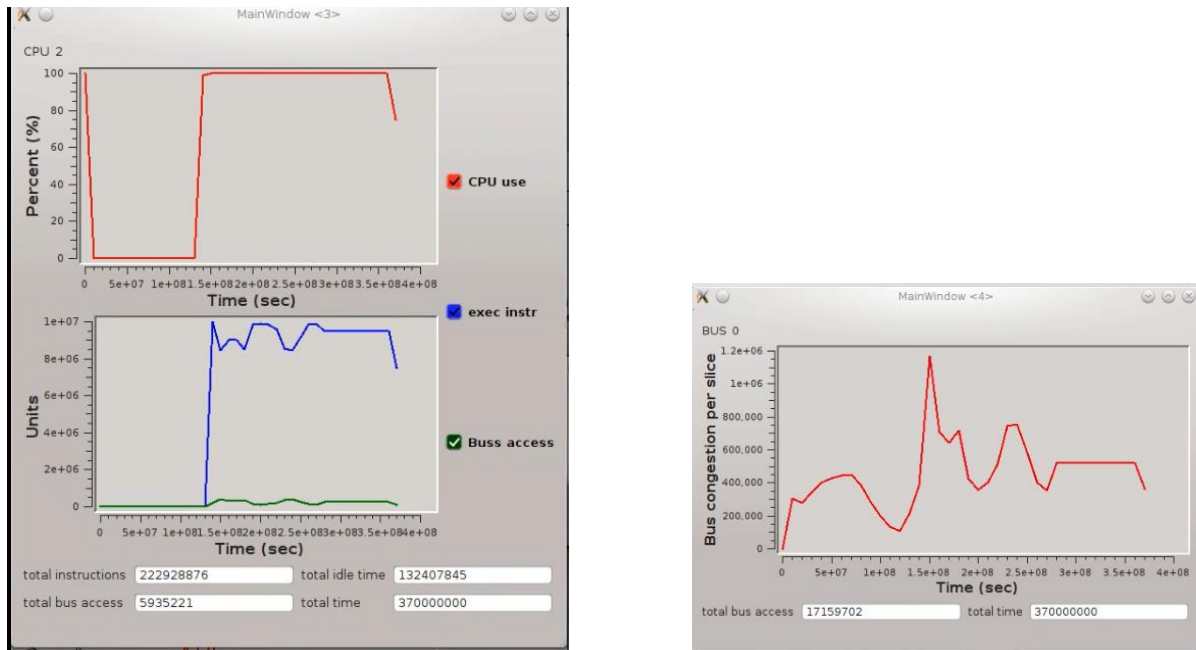
3.2.1 Status before CONTREX

The Virtual Parallel Platform for Performance Estimation (VIPPE) tool is a fast simulation tool. Before CONTREX, VIPPE was able to perform a fast simulation of the functionality and of the time performance. VIPPE fast simulation capability relies on native simulation, on the abstract models employed for modelling crucial elements on performance (caches, RTOS APIs), and the capability of the simulation kernel to exploit the underlying parallelism of the host machine. The modelled systems consist of concurrent applications mapped onto multi-core platforms, with the consideration of almost arbitrary bus-based architectures and hardly predictable elements, such as cache memories, and shared resources with contention effects, e.g. buses. A rich variety of attributes at different levels of the system, i.e. application, SW platform and HW platform, can be considered, in order to enable a holistic exploration in the search of the most efficient design solutions.

Before CONTREX, VIPPE was able to provide time performance metrics of different natures, e.g., end time of a thread, metrics about the activity of the system, e.g., number of instructions executed, amount of cache misses, etc. However, the assessment of these metrics relied on annotations on LLVM intermediate code, which despite being a quite generic annotation method, it lacks enough accuracy, e.g. for considering target dependent optimizations of the compiler. All these reports were available “offline”, once the simulation is over, in the shape of a console report (reported in D3.4.2 and D3.4.x reports).

Before CONTREX, VIPPE had already a capability for a runtime report via a specific graphical report front-end (report GUI). VIPPE report GUI consists of a set of specific graphical windows which, before the beginning of the simulation, allow to select the metrics to visualize; and that, during the simulation, dynamically show an updated evolution of the metric across a simulated time axis. Therefore, by the end of the simulations, all VIPPE report GUI windows report a time domain function for each metric.

For instance, Figure 5 shows the progress in time of some metrics of a processor instance, i.e. utilization, # executed instructions, and # bus accesses (Figure 5a), and the congestion of the bus along time (Figure 5a).



(a) Run-time report of CPU metrics (b) Run-time report of a bus metric

Figure 5. Example of run-time graphical report of VIPPE.

In order to feed the VIPPE graphical front-end, the sampling of the values of those performance figures is directly hardcoded in the kernel code. That is, the internal data structures employed to hold the data are directly read and transferred via sockets to the VIPPE report windows.

3.2.2 Status after CONTREX

Before focusing on the improvements on the monitoring capabilities, the consideration of other related improvements done in CONTREX and reported in other deliverables is convenient. A summary is given in the following subsection. Then, next subsections are devoted to summarize CONTREX contribution and to the details on them.

3.2.2.1 Improvements on EFP estimation in VIPPE

First, CONTREX work has enabled a number of improvements on the estimation of performance figures in VIPPE. They have been reported in D3.1.x, and can be summarized in that VIPPE has been enhanced for enabling:

- A more accurate and efficient time report, by enabling a more precise annotation mechanism, capable to consider the target processor and compiler optimizations.
- A more efficient report of caches, enabled for multi-core targets on multi-core host simulation.
- The accounting and report of energy and power consumption metrics

These enhancements and extensions have enabled the enhanced monitoring and reporting capabilities of VIPPE. They are comprised in the scope of the own VIPPE tool.

In addition to that, CONTREX has served to link VIPPE with a UML/MARTE front-end. The CONTREX Eclipse plug-in (CONTREP) (reported in D2.5.x), enables the automatic generation

of a VIPPE performance model, including the application platform independent and platform dependent code, and a detailed description of the platform, of the mapping of the application to the platform and of the extra-functional attributes associated to the platform.

In addition, to that, the UML/MARTE methodology enables the description of performance requirements and thus, of the performance metrics to be accounted and reported. This information has to be translated into the intermediate text-based representation, which serves as an input for VIPPE to know which metrics to monitor and account to latter on, enable their report at the end of the simulation. This is necessary and crucial enabling the user a focused inspection of the performance, but also for automated requirements checking and for design space exploration. As for the aforementioned code generation infrastructure for the generation of the performance model, the code generation infrastructure and the formats for stating performance requirements and output performance metrics, from the UML/MARTE level to the XML representation serving as input to VIPPE was not present before CONTREX.

3.2.2.2 CONTREX achievements

The following enhancements have been done to VIPPE with respect to monitoring and reporting capabilities:

- Support of XML description of output performance figures and requirements, supporting also the Multicube XML format [9] for DSE
- Support of response times report
- Exporting power traces, able to link in turn with the Intel tools for Temperature assessment.
- Introduction of a monitoring API for modular extension and for connection to third party tools.

All these extensions are explained with further detail in the following subsections.

3.2.2.3 XML description of output performance figures and performance requirements

As advanced, the UML/MARTE methodology enables the description of the output performance figures to be analysed and moreover, of the performance requirements relying on them. Examples on the quadcopter model are shown where the modeller states the desire to report some overall, system-independent metrics (

Figure 6), of performance metrics (#instructions, load) related to a platform component (Figure 7), performance metrics (response time) associated to an application component (Figure 8), and again, performance metrics (power consumption) associated to several platform components (Figure 9).

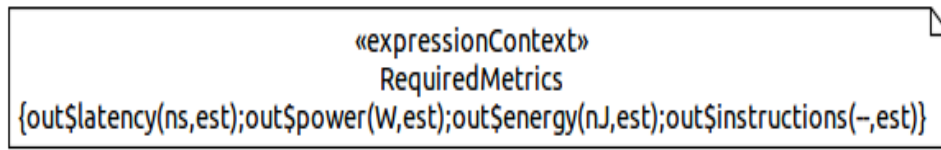


Figure 6. Example of overall, system independent metrics to be reported.

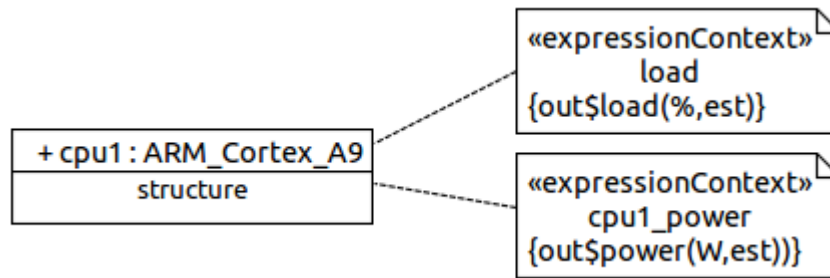


Figure 7. Example of metrics associated to a platform element to be reported.

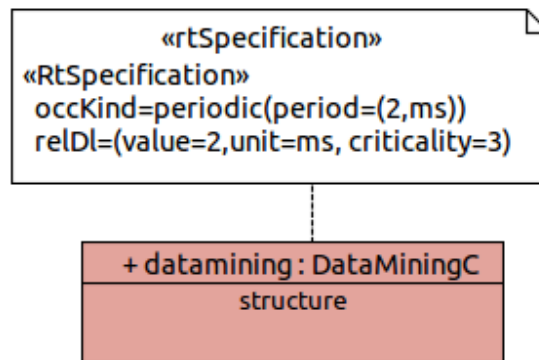


Figure 8. Example of output performance metric associated to an application. The construct also reflects the specification of an application-level performance requirement (deadline).

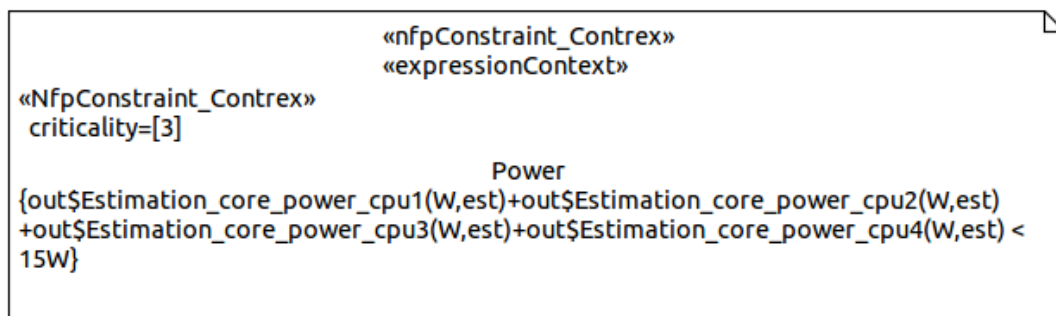


Figure 9. Example of performance requirements specification involving the report of several output performance metrics associated to several platform components (processors).

Moreover, Figure 8 illustrates a specific case of the UML/MARTE modelling methodology where the performance metrics is implicitly inferred from the performance requirement (deadline on the response time of the periodic functionality of the application component). Figure 9 illustrates a case where a performance requirement is built-up through an expression

which relies on 4 performance figures (power consumption of each CPU core of the platform), and thus which implicitly requires their report.

As well as enabling the modelling of the performance metrics to be monitored, accounted and reported in UML/MARTE, CONTREP has enabled the automatic generation of the text-based description of these metrics in XML, suitable for DSE, and serving as front-end for VIPPE. In fact, the XML front-end of VIPPE has been enriched to read the aforementioned XML description.

Figure 10 shows a snapshot where the CONTREX Eclipse Plug-in has automatically produced out of a UML/MARTE model including the constructs shown in

Figure 6, Figure 7, and Figure 8, an XML file (*design_space_4simulation.xml*). Among other information, this file contains the text-based counterpart describing the output performance metrics to be accounted and reported in an XML file called *output.xml* at the end of the simulation. These files abide the Multicube specification, in order to allow the automated connection with an exploration tool.

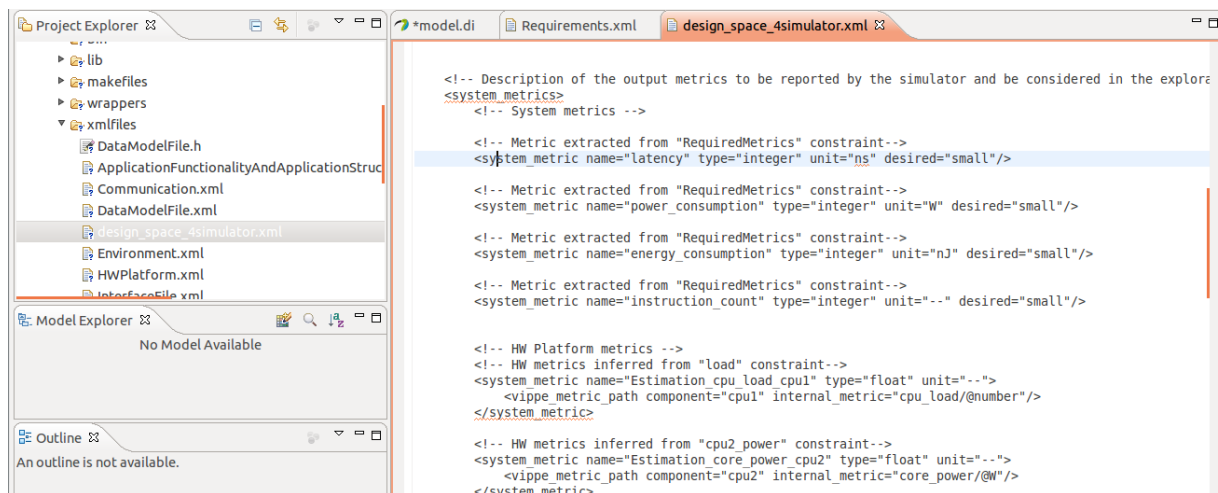


Figure 10. CONTREP generates the *design_space_4simulation.xml* file, where output metrics to be reported are captured in a Multicube compliant format.

The excerpt of code of the *design_space_4simulation.xml* file shown in Figure 10 shows the description of the overall, system-independent metrics and also of the some of the metrics related to platform components (CPU load and power). There it can be noticed a particularity of the *design_space_4simulation.xml*, with respect to the Multicube interface. Specifically, for each *system_metric* tag, a tool specific metric description, in this case, for VIPPE (tag “*vippe_metric_path*” is inferred by the tool set). This follows a solution which was defined in the COMPLEX project [10], in order to enable a traceable translation between the strict naming convention in Multicube (expression of the type [A-Za-z_][A-Za-z0-9_]*) and more flexible naming systems at each specific tool. This is an example showing how the tooling developed in CONTREX has re-used (whenever possible) interface definitions and solutions developed in previous related projects (Multicube, COMPLEX).

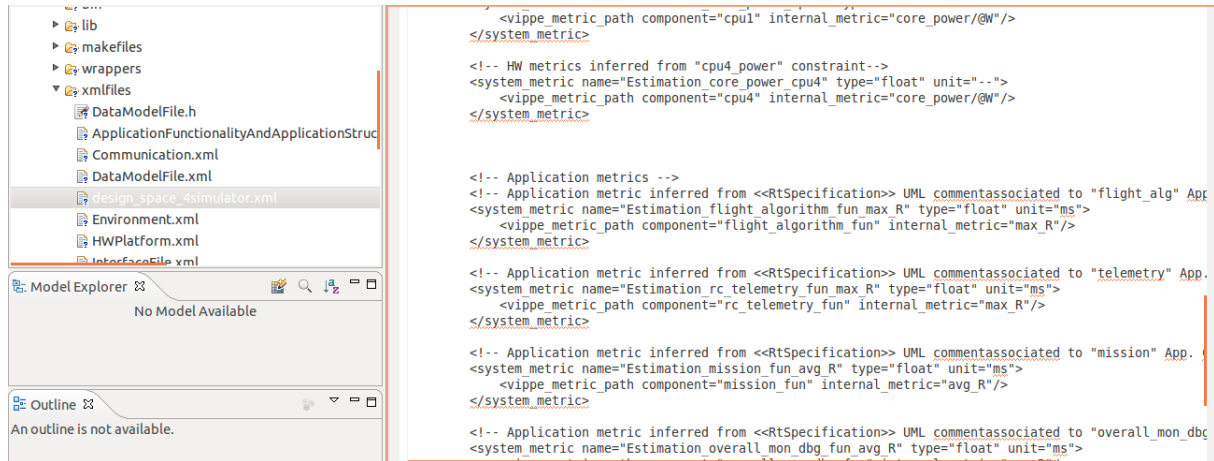


Figure 11. Excerpt of the *design_space_4simulator.xml* file within the CONTREP framework, showing also the description of application metrics.

The excerpt of code of the *design_space_4simulator.xml* file shown in Figure 11 shows also the inference of tags stating the output of performance metrics, specifically response times. This type of report is a novel feature enabled by CONTREX, as reported in the next sub-section.

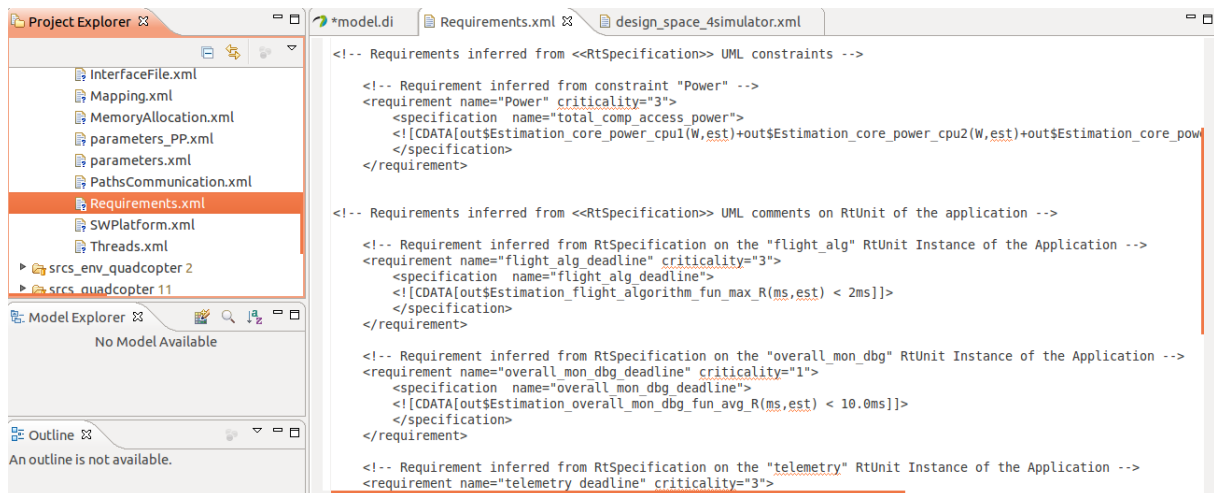


Figure 12. Excerpt of the *Requirements.mxl* file generated by the CONTREP framework.

The CONTREP framework is also capable to automatically generate a XML description of the performance requirements. It is dumped in the *Requirements.xml* file, illustrated in Figure 11 for the quadcopter case. This file is specifically used for the interactive validation of a simulated solution. Interactive means that the user simulates (launches the simulation) of a performance model generated with CONTREP, for a specific design solution, and then tests if the different performance requirements are fulfilled. This check is automatically done by CONTREP. CONTREP integrates a checker application, which reads both, the *output.xml* file (with the performance figures estimated after the VIPPE simulation of a specific solution) and the aforementioned *Requirements.xml* file. This checker application reports the fulfilment or violation of the performance constraints to the “Error Log” window of the CONTREP framework. A remarkable particularity is that the severity (error, warning, info) of the report associated to each performance requirement checked is defined in terms of the criticality associated to the requirement (notice that criticalities associated to the requirements in the UML/MARTE model (e.g. power requirement in Figure 9) are also translated into the XML counterpart Figure 12).

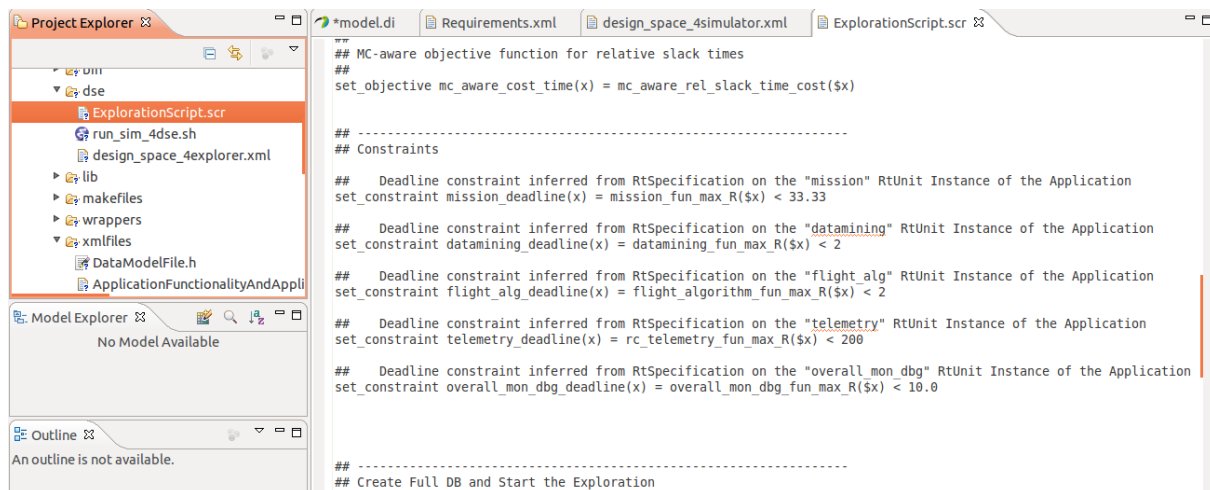


Figure 13. Excerpt of an *ExplorationScript.scr* file generated by the CONTREP framework.

Figure 13 shows an excerpt of an exploration script file generated for the quadcopter (targeting an exploration relying on the MOST exploration tool). Such an excerpt illustrates the translation of the performance requirements captured in the UML/MARTE model, and thus of their associated output performance metrics, on the information passes to the DSE infrastructure.

Finally, notice both in Figure 12 and Figure 13 traceability information in the shape of comments (adapted to each specific format). It provides valuable information to the user about possible errors or lacks in the model, as well as for the own debugging of the toolset.

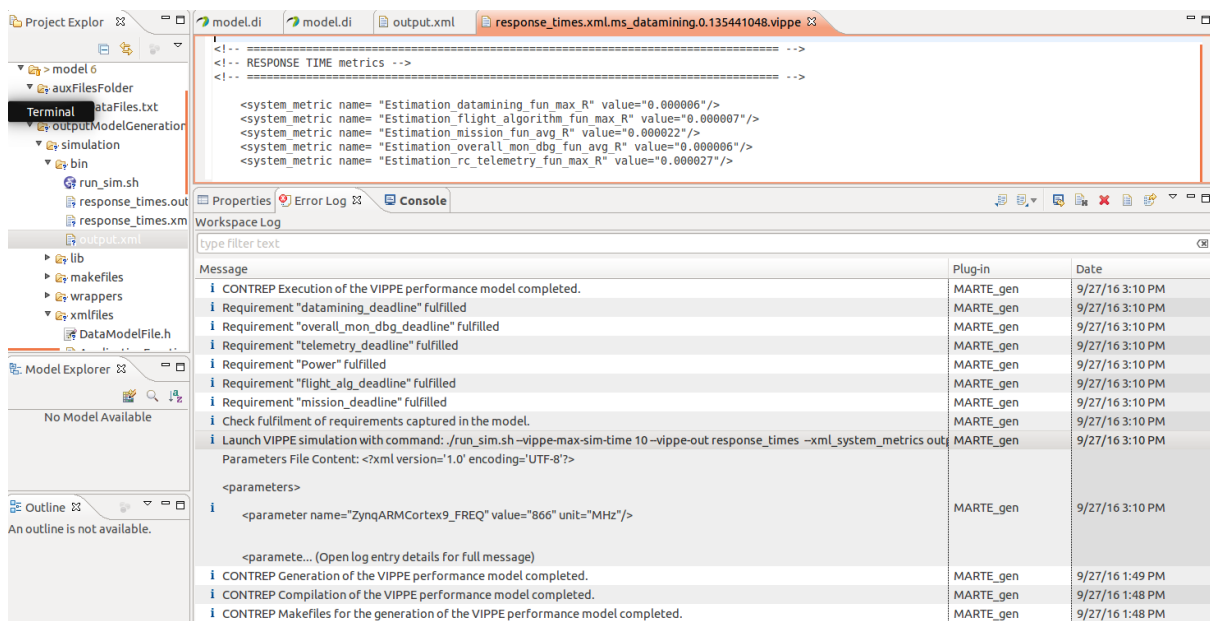


Figure 14. Execution and Validation of the solution. All performance requirements are fulfilled in this example.

Figure 14 shows the launch and validation of the performance vs the performance requirements through the simulation of a single-solution through the VIPPE performance model generated by CONTREP. A “zoom” to the info log reporting the simulation launch command would show:

```
./run_sim.sh -vippe-max-sim-time 10 -vippe-out response_times -xml_system_metrics  
output.xml -xml_metric_definition ../xmlfiles/design_space4simulator.xml
```

Where, the run_sim.sh script, automatically generated by contrex, contains in turn the first part of the VIPPE launch command, i.e. :

```
vippe32.x -vippe-platform-xml ../xmlfiles -vippe-mapping-xml ../xmlfiles -vippe-so-path ../lib
```

With this command configuration, VIPPE is able to assume that the user wants to simulate a “default” solution (configuration), with the default values for the DSE parameters, whose value is stated in a default parameters.xml file, automatically generated by CONTREP too from the default values for the DSE parameters given in the UML/MARTE model. Such a configuration is dumped also in the “Error Log” as an information message. This way, the user knows which configuration the simulation and the validation refers too. Then, the following messages in the “Error Log” window refer to the checks of the performance requirements once the simulation is completed and the output.xml file with the performance metrics corresponding to the simulated configuration is ready. In the Figure 14, all the performance requirements are fulfilled, what it is reported by specific informative cases.

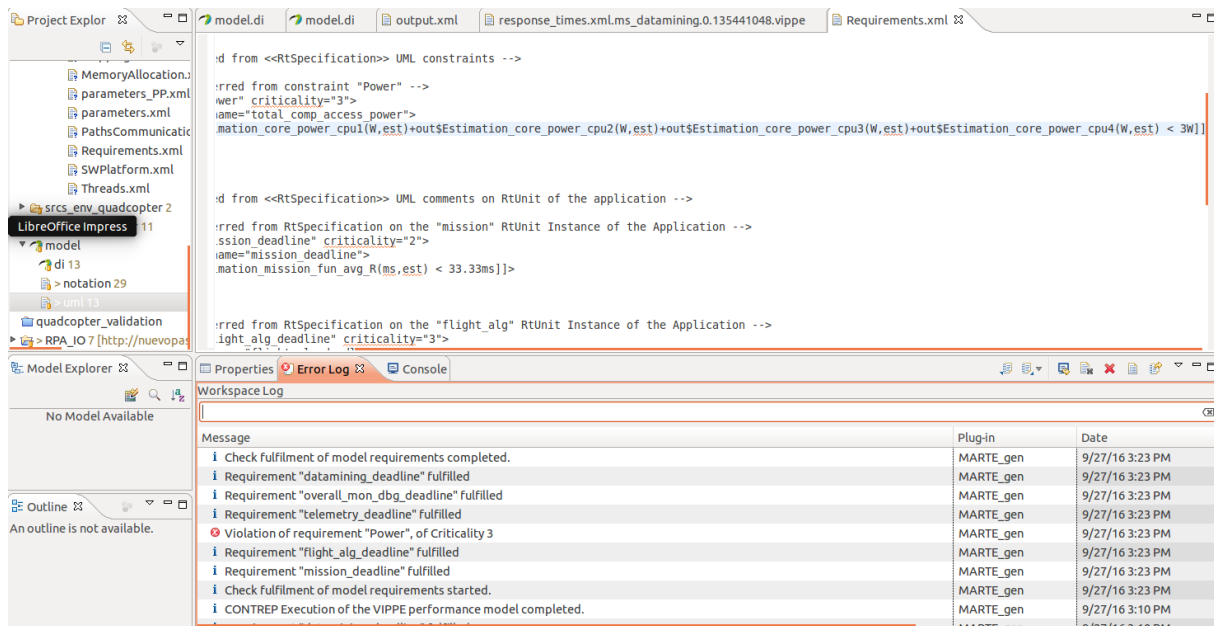


Figure 15. Execution and Validation of the solution. One performance requirements is not fulfilled in this example. The criticality level and CONTREP configuration involve to be reported as an error in the Eclipse “Error Log”.

3.2.2.4 Response time reports

In CRAFTERS, a parallel project overlapping with CONTREX, UC enabled a basic LLVM instrumentation for time stamping entry/exit function at run-time. This enabled an important set of reports for the target platform, which typically where enabled for host platforms (e.g. # function calls, call graph, function call sequence per thread) by tools like *gprof* or *Valgrind*.

In CONTREX, the VIPPE capabilities have been enhanced to support the report of response times. The report of response times is crucial, as many mixed-criticality systems will combine performance requirements focuses on different types of performance metrics (power consumption, throughput, etc) with the need to a more or less good quality on the time response

of one or more components. To accurately assess if deadlines are fulfilled provides good validation mechanisms, specifically for time-critical parts. Moreover, in design contexts where time responses do not refer to safety critical parts, simulation-based analysis of these response-times can be also used for design. For the enabling the report of the response times, two main activities have been completed for that:

- the adaptation of the annotation infrastructure developed in CRAFTERS, which was suitable only for model modelling mono-core platforms,
- the extension of the command interface to enable the user the response time report, which accounts for the support of the `-vippe-out response_times` option (see Figure 14 in previous section), and also the ability in the CONTREP configuration tab to enable the required instrumentation, previously mentioned.
- the development of an additional post-processing functionality to enabling the report of minimum, average, and maximum report times per instrumented function,
- the extension of the XML front-end to support at the output metrics specification of the response metrics (max_R, min_R, avg_R), and the corresponding extension of CONTREP to support the generation of this XML description out of the UML/MARTE model,
- the extension of the XML output interface for the report of the response times in the output.xml file under the Multicube format.

The latter, has the particularity that the response time metrics are metrics accounted in the user processes (while platform and overall metrics are computed in the kernel process). In order to avoid communication overhead, user processes report this information into XML intermediate files (an example shown on top of Figure 14), one per user process,. Then, the kernel process integrates these XML response time reports into the overall output.xml report file.

3.2.2.5 Production of Power Traces

VIPPE is a performance analysis simulation-based infrastructure that, in CONTREX, has been deeply extended and re-factored in order to provide fast time and energy/power assessment. VIPPE perfectly adapts to the scheme for temperature assessment sketched in

Figure 16. In such a scheme, temperature assessment mostly relies on the *Thermal Profiler* tool from Intel¹. Section 3.3 of D3.1.3 provides technical insight on this tool. *Thermal Profiler* tool is capable to consider a rich set of technological parameters and capture the geometrical characterization of the platform (sketched as .dpd file in Figure 15). *Thermal Profiler* enables several types of thermal analysis (steady state, step, dynamic).

¹ Previous versions of the Intel toolkit for dynamic analysis of temperature relied on two tools, Thermal Profiler and AceExplorer. The piece of work reported in this section relies on Thermal Profiler a more recent unified version of the aforementioned tool pair.

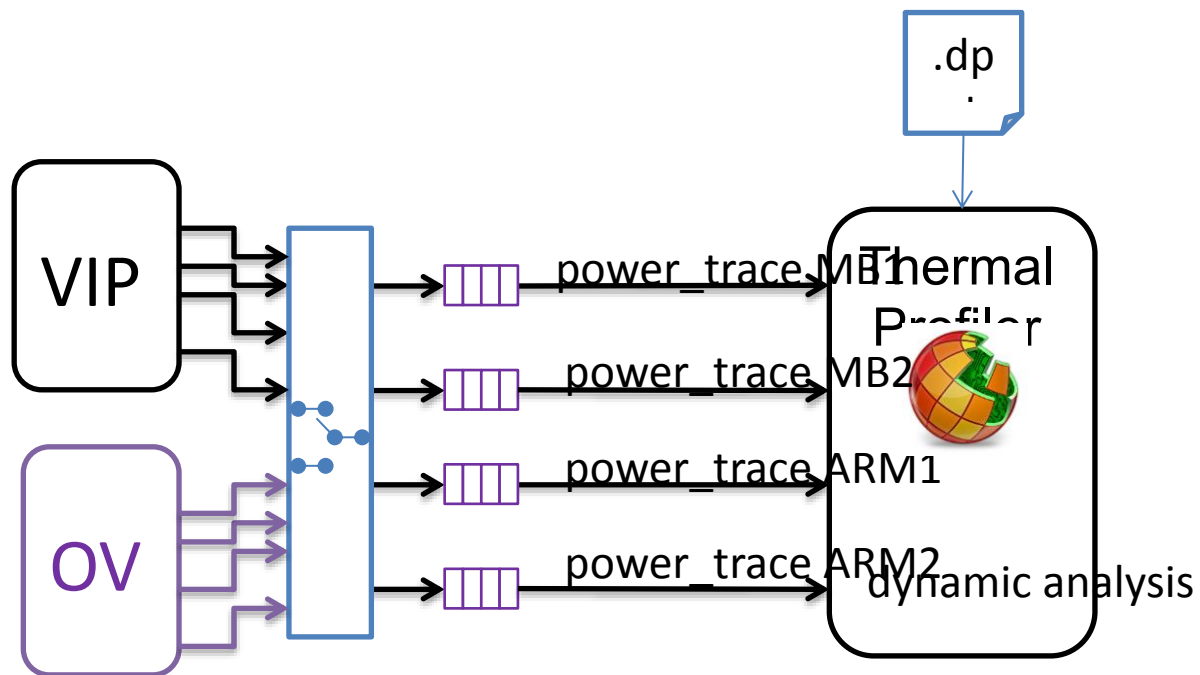


Figure 16. In CONTREX, virtual platforms based on different simulation technologies, i.e. native simulation (VIPPE) and binary translation (OVP) enable the export of power traces for dynamic thermal simulation.

The dynamic analysis enables to assess the progress of temperature along time and across the SoC geometry. Therefore, is a very detailed analysis which allows to assess any problematic hot spot at any specific time among different possible simulation scenarios. As well as the geometrical and technological characterization, *Thermal Profiler* requires another basic input for the dynamic thermal analysis, i.e. power traces. Specifically, a power trace per power in the SoC model captured in *Thermal Profiler*.

Power traces are classified as secondary traces in the extra-functional property estimation flow shown in section 4.1 of D3.1.3. As was mentioned there, the flow enables the combination of several point tools and techniques. Figure 15 highlights a specific synergic combination of tools centered on the production of the power traces. As mentioned in other reports, VIPPE and OVP simulation and estimation complementary technologies. VIPPE technology is suited for fast architectural exploration, enabling fast automated DSE. OVP is remarkable in enabling SW development on top of the selected virtual platform, and in achieving accurate estimates in terms of instructions at least.

Therefore, Figure 15 shows the possibility for an automated architectural DSE where time, energy and power performance figures can be considered. A filtered design space can be analysed based on a detailed OVP-based platform. In both cases, the thermal analysis technology is shared.

Further clarifications on the power traces produced by VIPPE are given. VIPPE handles a number of default accounting variables. They are accumulators which are crucial for dynamic energy and power estimation:

- *# instructions*
- *# instruction cache misses*

- *# bus accesses*
- *# of data cache misses*

These accumulator variables are used at the end of the simulation, in the post-processing associated to either the report to console or to produce the XML output file for automated DSE. This strategy enables to optimize simulation speed.

However, when power tracing is enabled, processing for energy&power calculation is required at the specific times where power is monitored. A decision on when to monitor and thus calculate is required. For VIPPE tracing the decision taken so far was to enable a monitoring period equal to the VIPPE time slice², as this is the minimum period where it is ensured all user processes are synchronized³. But at the same time that means that for each time slice, the computation of the energy for that time slice is performed.

VIPPE is a flexible framework, such it enables to enable/disable the annotation and accounting capabilities. For instance, in the lightest configuration, VIPPE can be compiled to support only instructions estimation, thus to optimize simulation speed. Under this configuration, the dynamic energy estimated in the power traces considers the amount of instructions executed in the time slice. For that, a state variable for the instructions accumulated at the end of the previous slice is required. The frequency configured for the processor is considered. In the implementation at the time of this report, these traces do not consider configurable voltages. However, as reported in D3.1.2, VIPPE can consider the annotation an amount of cycles per instruction (which allows giving a time dimension to instructions executed), and also an energy cost per instruction, which overall allows for a high-level model of specific process architecture.

Moreover, if cache modelling is enabled (and the platform model includes processors with caches) then the power traces exported by VIPPE include also the contributions of level 1 instruction and data caches associated to the processor instance. For this, further variables for storing the state of the account variables (icache misses, bus accesses, dcache misses) in order to compute the amount of icache misses, bus accesses and dcache misses at each time slice is required. With this traces, the dynamic energy consumption contributed by the L1 instruction and data caches at a given slice is computed. Power at the end of the slice is computed after division by the time slice time.

The power tracing has been simply activated through an additional value (“power_traces”) for the “-vippe-out” switch, which can be invoked more than once. It is illustrated in the following alternative command for invoking the vippe simulation illustrated in section 3.2.2.3.

```
./run_sim.sh -vippe-max-sim-time 10 -vippe-out response_times -xml_system_metrics output.xml -  
xml_metric_definition ../xmlfiles/design_space4simulator.xml -vippe-out power_traces
```

This command produces a single power traces file in VCD format. The file is named “power_traces_date_time.vcd”, where *date_time* stands for the date and time when the power trace is exported. The following excerpt of .vcd file illustrates the output of VIPPE in this case.

² An extension to enable the augmentation of the monitoring period to speed-up simulation is possible rela

³ Monitoring in intermediate points is also possible, although the accuracy will in general depend on several aspects, such as how far the point is to the synchronization point, the size of the basic blocks in the time slice, etc.

```
$comment
    Power trace file generated by VIPPE tool
$end
$timescale 10 ms $end
$scope VIPPE_model $end
    $var real 1 a power_ARM1 $end
    $var real 1 b power_ARM2 $end
    $var real 1 c power_MB1 $end
    $var real 1 d power_MB2 $end
$upscope $end
$enddefinitions $end
r0.0 a
r0.0 b
r0.0 c
r0.0 d
#10
r5.91667 a
r3 b
r1 c
r1 d
#20
r3.00005 a
r3 b
r1 c
r1 d
#30
r14.4336 a
r3 b
r1 c
r1 d
#40
r3.00007 a
...
```

Figure 17. Excerpt of VCD file with the power traces associated to the 4 processors of the quadcopter platform.

3.2.2.6 Monitoring API

Moreover, in CONTREX, a monitoring API has been defined, for the run-time access to the output performance metrics. The purpose of this API is two-fold. First, to facilitate the modular extension of the dynamic reporting capabilities in the tool. Second, to facilitate the connection with third party tools which display or exploit that monitored performance data. The API is sketched in Figure 18.

Description
<i>MonitoredMetricHandler import_metric(int MetricContext, char *InstanceName, char* MetricName, int SynchType, const AllMetricsHandler *all_metrics = NULL)</i>
<p>Metric Context: input param defining the context of the metric. Valid values are SYSTEM, PROCESSOR, CACHE,BUS,</p> <p><i>InstanceName</i>. In case it is not a system metric, defines the instance name the metric refers to.</p> <p><i>MetricName</i>: name of the metric to be reported. Meaning depends on the MetricContext</p> <p><i>SynchType</i>. defines the type of synchronization of the metric reader. If there is no value or the value is SYNCH, then the access is blocking. In this case, the getMetricValue (or its alias getNextMetricValue) can be called. Each invocation is consuming and blocking (the caller will block until the next value of the metric on the next slice time is available. If the value is ASYNCH, the access is asynchronous. This means that each invocation peek (sample) the value of the metric (as it is hold by the VIPPE kernel).</p> <p>Return Value: A struct of type MonitoredMetricHandler type. This structure contains also the time associated to the last value read.</p>
<i>AllMetricsHandler import_all_metrics(char *host, char *port)</i>
<p>Import all metrics that can be monitored by VIPPE and thus imported. Used for communications with third party tools.</p>
<i>int updateMetricValue(MonitoredMetricHandler &metric_handler)</i>
<p>Dumps in the metric handler the value of the metric. When the metric is imported synchronously, each call guarantees that each value update refers to an updated advance of the time slice. When the metric is imported asynchronously, the value corresponds to the simulation time until the VIPPE kernel has advanced right at the time of the call (oversampling is possible in this case).</p> <p>Return Value: <0 in case of failure</p>
<i>int& readIntMetricValue(MonitoredMetricHandler &metric_handler)</i>

Returns the value of the metric associated to metric_handler right after the last update of metric value with getMetricValue. Used for metrics of integer type.
<i>double& readFloat/MetricValue(Monitored/MetricHandler &metric_handler)</i>
Returns the value of the metric associated to metric_handler right after the last update of metric value with getMetricValue. Used for metrics of float type.
<i>long long int readTimee(Monitored/MetricHandler &metric_handler)</i>

Figure 18. Basic Monitoring API for VIPPE extension (for connection with 3rd party applications the import_allmet)

Figure 19 sketches a case where VIPPE dynamic reporting capabilities are extended relying on the monitoring API. In that example, a simple import of metrics is feasible. Metric import provides a way to configure the synchronization policy to access the metric and single handler to later update an read metric in a simple way.

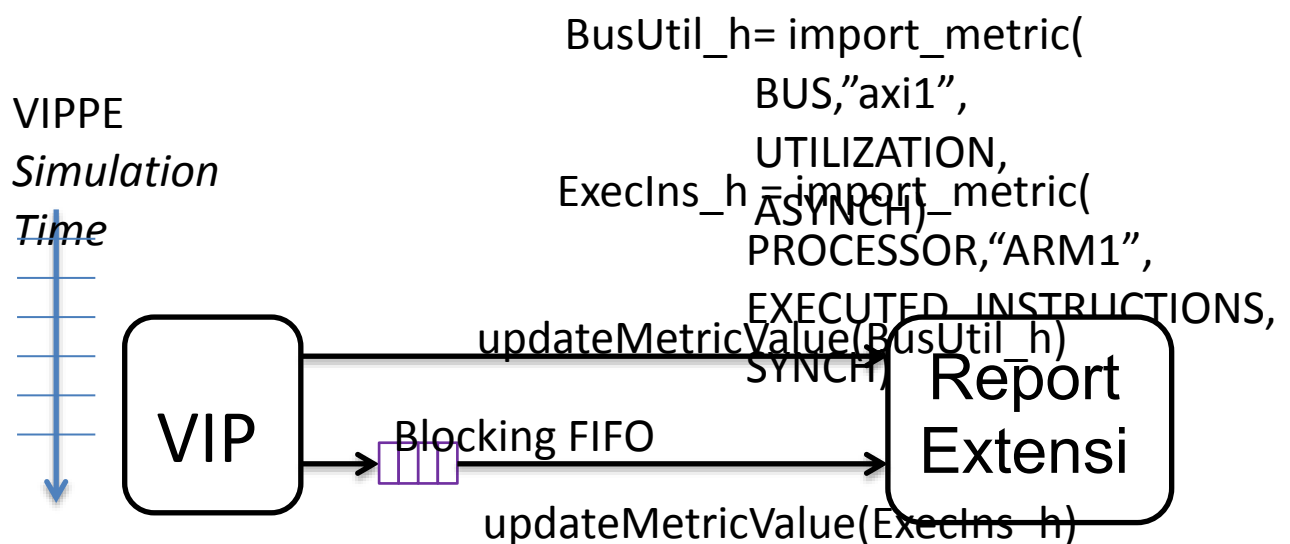


Figure 19. The VIPPE monitoring API can be employed for a modular extension of VIPPE dynamic reporting capabilities.

In Figure 19 use case, the API can directly call the “import_metric” functions. In the use case of enabling 3rd party applications to exploit or display VIPPE performance metrics, a previous call to “import_all_metrics” function, which performs the effective link of the client 3rd party function with VIPPE.

Figure 20 illustrates the use of the VIPPE monitoring API for the export of power traces. Figure 21 illustrates the use of the VIPPE monitoring API for the import of activity metrics (instructions, frequency) for estimating the power traces via the C functions with the power model of the Zynq processing system provided by OFFIS.

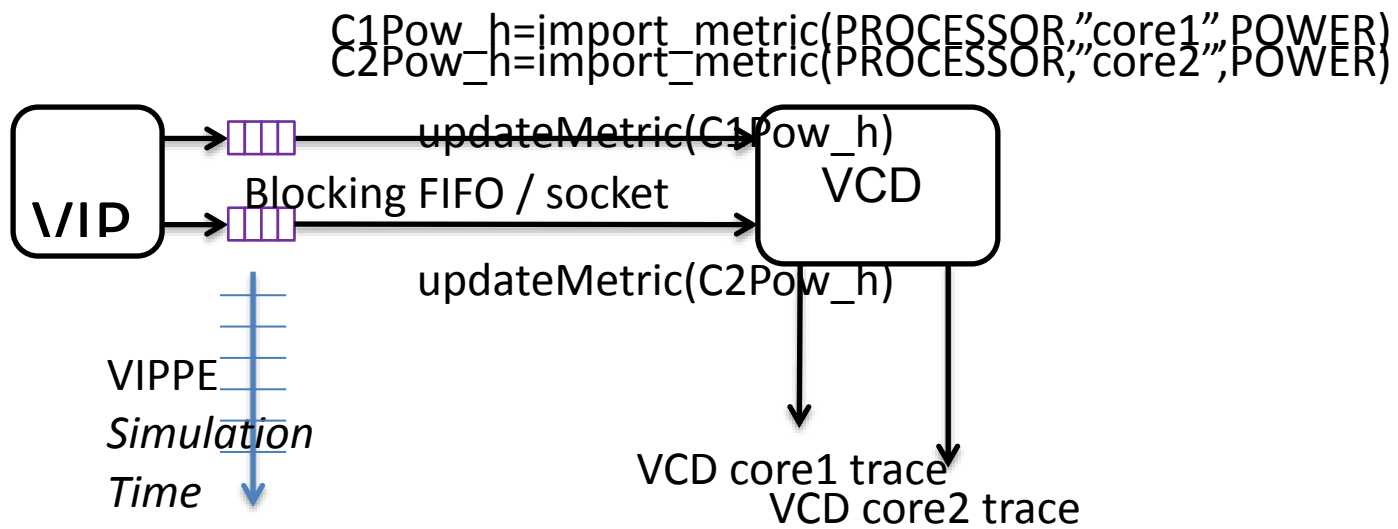


Figure 20. The VIPPE The current support of the VIPPE API ensures the coverage of the activity

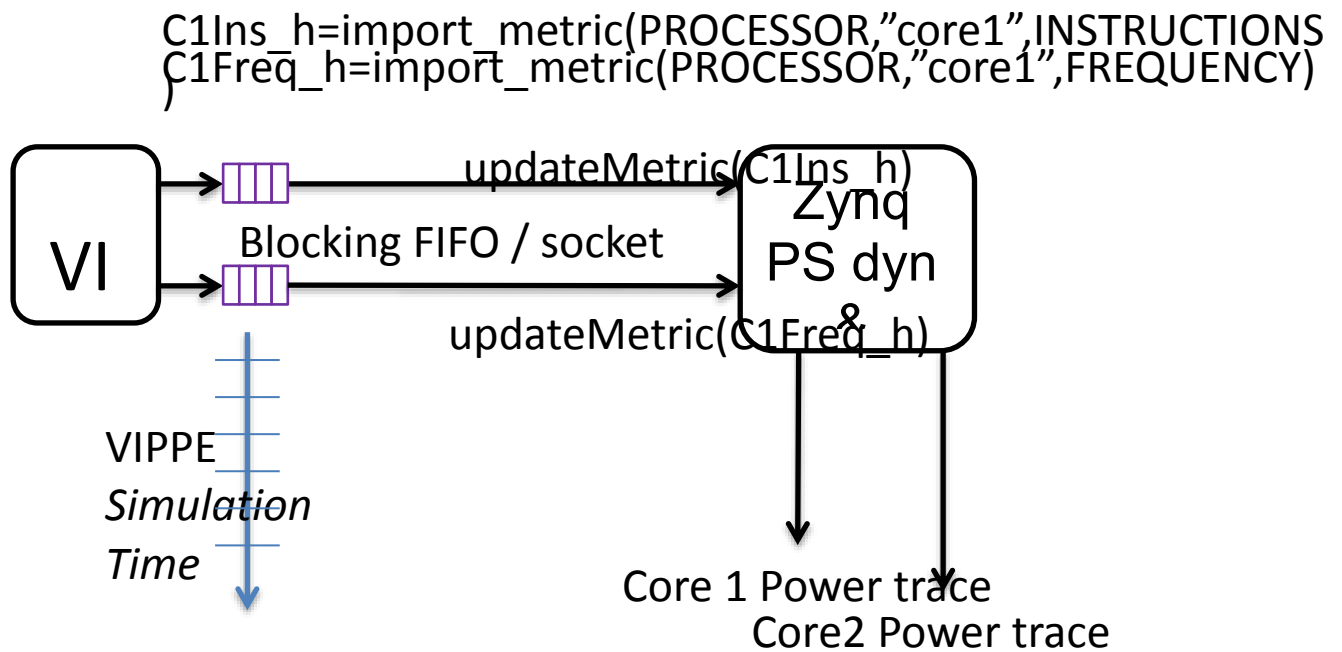


Figure 21. Import of activity traces for the generation of power traces with the specific Zynq processing system power model.

3.3 Extra-functional monitoring at the system-level [iX]

In the last project period the focus of the work package was in the development of a library to model the behavior of extra functional costs like

- Motor temperature
- Battery temperature
- Processor temperature
- Power control unit temperature
- Battery load level
- Turnaround time for real time application

and their drawback to the safety and the reliability of the system in pure simulation driven calculations.

In this project period the work was focussed on the implementation of the library for real-time hardware test environments. This allows more realistic hardware-in-the-loop tests because it enables the user to monitor the extra-functional costs on the system level.

For that purpose the parameters (e.g. the slope of the temperature behavior of the CPU, et.al.) were investigated in more detail and especially in which intervals they vary. This allows parameter studies as well as the investigation of the robustness of the system against parameter changes.

Also the nonlinearity of the extra functional properties and their influence on the systems behavior were investigated. The nonlinearities are the main reason of system failures and therefore in the focus for robustness and stability investigations. This work was performed in close cooperation with UC and OFFIS.

3.4 Application-Level light-weight extra-functional monitoring [POLIMI]

This section describes the specifications of the lightweight framework that will be used to monitor extra-functional properties for applications in microcontroller-based nodes (such as the one used in UC2).

The main purpose of the framework is to monitor properties at runtime that are not directly controllable from the application's code: the extra-functional properties. The most common metrics are the timing ones, such as the response time or the throughput, energy/power related and the quality metrics, whose definition is totally application-dependent. The application can exploit that information at runtime for adapting its functionalities and at design time to optimize its performance.

3.4.1 Framework Overview

The source code of an application enables the system designer to carefully state the functional behavior of the application. Nonetheless, common programming languages are not designed to address also extra-functional properties, such as the amount of data transmitted or received (e.g. through the UART interface), the power consumption of a specific device on the platform, the residual energy of the battery. The purpose of this framework is to fill this gap, enhancing an application with the ability of dynamically observe the interested extra-functional properties, end eventually to react with Run-Time actions as those under analysis in Task 3.3 of the project.

From the framework point of view, an extra-functional property is composed by two information:

- **Metric:** The name of the extra-functional property
- **Device:** The name of the device that the metric refers to.

In order to be general, the framework must be agnostic about the nature of the monitored extra-functional properties, the system architecture and the application structure. To provide such flexibility, the framework relies on models to compute the values of these properties. The main idea of the framework is that on one hand the target application notifies the framework about any interesting *event* that happens in the system. On the other hand, the framework exploits this information to update the models of the system. Relying on models, the framework is able to follow the evolution of extra-functional properties and to provide their values to the application.

The definition of an interesting *event* is tightly tied on the definition of the models themselves, however the followings are examples of *events*: (i) Sent 5 characters through the UART interface; (ii) The microprocessor is going to sleep; (iii) The sensor has gathered 30 samples.

The actual framework that interacts with the application is meant to be used as an external library that must be linked against the target application. However, the C source files that compose the framework implementation are automatically generated from an XML configuration file, that describes the system models and the extra-functional properties to monitor. In this way is possible to have separation of concerns between the application logic and the extra-functional abstraction. The benefits of using this approach are twofold. On one side the XML configuration file is much more easy to read and maintain with respect to C source files, on the other side is possible to deploy a framework tailored on the designer requirements, in order to minimize the overhead introduced by the framework.

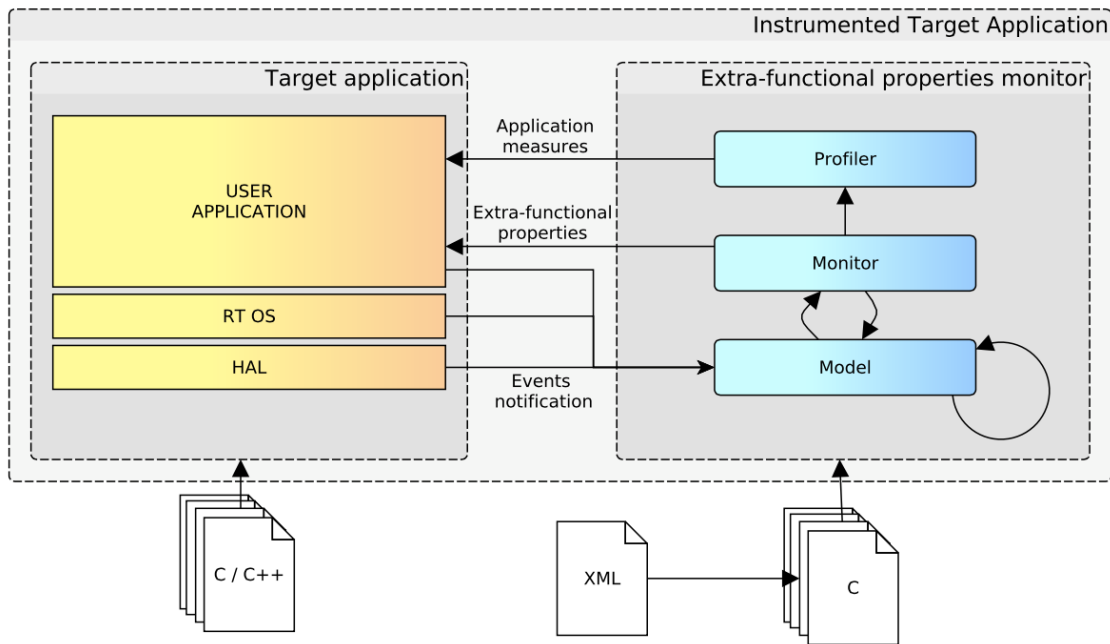


Figure 22 - The framework workflow. Starting from an XML configuration file, a tool automatically generates the C framework. A standard interface is used to interact with the target application.

The previous figure sketches the workflow of the framework. Starting from a system configuration that defines all the used models and how they interact with each other, with the system and with the application; a tool generates a tailored framework that implements the required features. Using a standard interface, the framework integration in the target application requires a minimal effort in terms of number of lines of code to be changed.

It is possible to logically partition the framework interface in three services:

- **Model** - This service is the back-end of the framework: it exposes the functions that update the models (*event* notification) in order to track the evolving system.
- **Monitor** - This service exposes to the application the extra-functional properties that the models compute, such as the energy consumption or the amount of received data.
- **Profiler** - This service is the front-end of the framework: it collects the measures of the extra-functional properties between a defined region of code and provides to the application their average values. It might also compute indirect extra-functional properties, such as the power consumption or the receiving speed.

3.4.2 Monitor Metamodel

Since the framework is agnostic about the nature of the monitored extra-functional properties, the framework consider a model as an abstraction of a characteristic of the system to exploit for computing the value of the interested extra-functional properties. The metamodel is in a higher level of abstraction with respect to the models since it is used to define any model that the system designer may require. Using a metamodel to define a specific model enables the framework to use a standard interface, no matter how the system is modeled.

3.4.2.1 Metamodel definition

The metamodel is logically defined by four elements:

1. **State** - Since the main purpose of a model is to represent the evolution of a characteristic of the system, the value of a modeled extra-functional properties may depends on the history of the system. In this case the model must be stateful, thus the metamodel enables the system designer to define the state of the model as a list of variables and constants. In the generated C implementation, the constants are represented as macros, while the state variables are represented as standard C variables.
2. **Finite State Machine** - If a model requires a state, the metamodel must provide a general mechanism to update it. For this reason the framework enables the system designer to define a Finite State Machine (FSM) to follow the system evolution. In particular on every transaction of the FSM is possible to define a list of formulas that describe how the transaction affects the state. Using this abstraction scheme, the *event* notification is defined by a transaction of the FSM of a model. In the generated C implementation, a single transaction is represented as a C function that updates the state variables. In fact, each formula in the transaction is represented as a C assignment, where the left hand expression is a state variable and the update formula is the right hand expression of the assignment.
3. **Input** - Even if the model represents a characteristic of the system, in order to compute the value of the modeled extra-functional properties, the model may require to interact with others element of the system. In particular, the metamodel enables the system designer to define a model that can have the following kinds of input: (i) The value of an extra-functional property computed by another model; (ii) The return value of a C function declared in an external header, for instance the OS function that provides the current time value; (iii) A local parameter with respect to a transaction of the FSM.
4. **Output** - This is the most important information of the metamodel, since it defines how the value of the modeled extra-functional property is computed. In particular, the formula that compute each property might involve: (i) The internal state of the model; (ii) Extra-functional properties computed by other models; (iii) The return value of external functions. In the generated C implementation, the output of the model is represented as a function that computes and returns the value of the modelled extra-functional property.

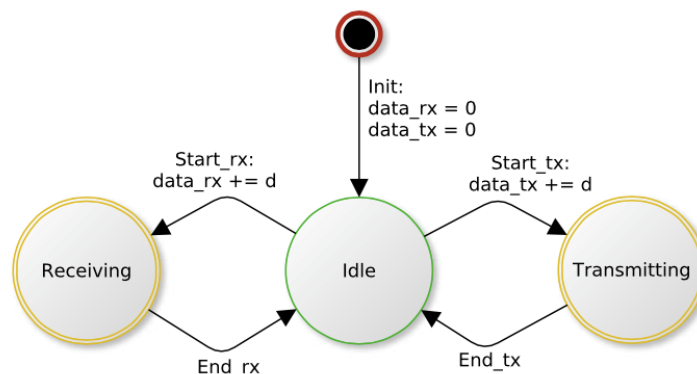


Figure 23 -The UART device model that aims at tracking the amount of sent and received data.

3.4.3 Metamodel example

To better describe the metamodel, we want to use a simple example where the system designer is interested on monitoring the amount of data transmitted and received by the UART interface. In this scenario the model that provides those extra-functional properties may be defined as follows:

- **State**-The state is composed by two variables that count how many data are sent or received, respectively “data_tx” and “data_rx”.
- **FSM** - The FSM of the model is showed in
- Figure 23.
- **Input** - This model doesn't requires to know the value of any additional extra-functional properties nor external functions, however it does require a local input on the transactions “Start_tx” and “Start_rx” to know how many data are sent or received (“d”).
- **Output**-In this example the value of the state variables is exactly the value of the state variables.

Since the UART operations are asynchronous, a real monitor should observe the physical device. However, the purpose of the framework is to compute the value of extra-functional properties using software models. In this example, the key idea of the model is to mimic the behavior of the physical devices, tracking the interested extra-functional properties.

In particular, the UART device will be in the “Idle” state if the application is not sending or receiving any data. When the application wants to send some data, it must notify the framework about the “Start_tx” event, specifying also the amount of sent data “d”. This notification has two effects:

- It changes the current state of the FSM to “Transmitting”.
- It increments the state variable “data_tx” by the amount “d”.

Once the transmission is finished, an interrupt will occurs and the application must notify the framework about the event “End_tx” and the FSM will move in the state “Idle”. A symmetric behavior is required when the application needs to receive some data. Since the state variables “data_rx” and “data_tx” are initialized to zero, their values is the amount of received and transmitted data through the UART interface.

The metamodel provides to the system designer a wide range of possibility to model the system. However, the recommended approach is to model each device that composes the platform, tracking the extra-functional properties of interest for each device. If the system designer is interested in platform-wide properties, he might use a model that collects the information from all the models of the devices.

3.4.4 Framework configuration

The framework implementation is automatically generated from an XML configuration file that defines the system models and states the interested measures (the model service and the profile service). With these information the tool automatically generates the framework implementation tailored on the system designer requirements.

This section aims at describing the semantic of the configuration file. An XML scheme has been defined as formal description of the XML configuration file syntax. The basic structure of the configuration is composed by two lists, as showed in the following XML snippet:

```
<monitor>
  <models>
    <!-- The list of the models definition -->
  </models>
  <profile>
    <!-- The list of the profiling measures definition -->
  </profile>
</monitor>
```

The first is the list of all the models of the system, while the second is the list of all the required measures. Since the monitor service of the framework provides to the target application the value of all the extra-functional properties exposed by the models, it is not required to explicitly state these information in the configuration file.

3.4.4.1 Model definition

The first section of the configuration file is the list of all the system models. Every model targets a single device and it is composed of four optional sections: the input, the output, the state and the FSM.

State

If the model requires a FSM, in this section is possible to define the list of variables and constants that compose the state. Every state variable may be updated in each FSM transaction. A state variable requires defining the following attributes:

- The name of the state variable.
- Optionally, a string that may be used as reference of this state variable in a transaction function or in an output function. The default reference string for a state variable is `<device_name>.<variable_name>`, where the device name is the value of the device attribute of the model.

Since a model may use constants in a function definition, beside the state variables is possible to define also state constants that act as an alias for a numeric value. A state constants requires to define the following attributes:

- The numeric value of the constant.
- A string that represents the alias for the value.

Input

This section states all the input that the model requires. The framework enables the definition of three kinds of input:

- **Metric** - This input is an extra-functional property computed by another model, for instance the current time value of the microprocessor. This input requires to define the following attributes:
 1. The name of the required extra-functional property
 2. The name of the target device
 3. Optionally, a string that may be used as reference of this input in a transaction function or in an output function. The default reference string for an external metric is `<device_name>.<property_name>`.
- **Local** - This is an input local with respect to a transaction of the FSM. For each transaction that requires this input in order to update the state, the framework assumes that the application will provide the value of this input as a parameter of the macro. This input requires to define the following attributes:
 1. The name of the input that will be used as identifier for the parameter for the generated function.
 2. Optionally, a string that may be used as reference of this input in a transaction function. The default reference string for a local input is its name.
- **External** - This input is typically a function declared in an application header, for instance the function that retrieves from the Operating System the time value. However the framework is able to use also variables declared as external. This input requires to define the following attributes:
 1. The name of the function or variable, it must be a valid C identifier.
 2. The header that declare the function or variable.
 3. Optionally, a string that may be used as reference of this input in a transaction function or in an output function. The default reference string for an external identifier is its name.

Output

Every “output” tag of the XML configuration file, defines a single extra-functional property that the model provides. Thus, each output must define the name of the exposed property and write the formula to compute its value as text information inside the “output” tag. The formula may be function of the state, of external inputs and function of metrics computed by others models. In the formula might be used any mathematical C function that is declared in the *math.h* header file.

When the framework generates the framework, it first tokenize the formula and tries to figure out the meaning of each terms. Then replace every reference to an input or to a state variables or constant with the appropriate C code. For this reason, the formula must be written using the C syntax.

FSM

The last section of a model definition, describes the Finite State Machine of the model. In particular, every transaction of the FSM must be defined. Each transaction requires to define the following attributes:

1. The name of the transaction
2. The name of the starting state of the FSM
3. The name of the ending state of the FSM

Inside every transaction, it is possible to define how the state variables are updated. In particular for each update, must be defined the target state variable and the formula to use to compute the new value. The framework process the formula as described before, however in the transaction formula is possible to include references also to local input.

3.4.4.2 Profile section

This section defines the list of measures that the system designer requires. Each measure is composed by two sections: the observed extra-functional properties and the derived metrics.

- **Observed extra-functional properties** - Every extra-functional property that is exposed by a model may be used as a metric to profile a region of the target application. Each metric requires to define the following attributes:
 1. The name of the device
 2. The name of the extra-functional property
- **Computed metric** - Using the values of the observed extra-functional properties is possible to compute indirect metrics, such as the power consumption of a sensor or the receiving speed on a the UART device. Each computed metric requires to define a name and the formula that computes the value, which takes into account only the value of the observed extra-functional properties.

3.4.4.3 Metamodel example

This example aims at showing how to write the XML configuration file that implements the model of the UART device described in Section 3.4.3. In the previous example the system designer is interested on monitor the amount of data exchanged through the UART device where its model is defined as in

Figure 23. In this example we assume that the system designer is also interested on monitoring the transmitting and receiving speed as an indirect measure.

Since the speed is defined as the amount of data sent, or received, over a certain amount of time, we also need to model this last extra-functional property. Thus, in this example we define also a very simple model of the microcontroller that provides as output the time elapsed since the system is started. This information will be provided by an external function exposed by the OS. For this reason the microcontroller model doesn't require a state.

The following is the XML that represents a possible solution for the problem where the XML is divided in two sections:

1. The models definition (lines 3-43)
2. The measure definition (lines 46-54)

```

1  <?xml version="1.0"?>
2  <monitor>
3    <models>
4
5      <model device='MICRO'>
6        <input>
7          <external function='osKernelSysTick' header='cmsis_os.h' ref='now' />
8        </input>
9        <output name='Time'>now()</output>
10     </model>
11
12
13     <model device='UART'>
14       <state>
15         <metric name='Data_rx' ref='rx' />
16         <metric name='Data_tx' ref='tx' />
17       </state>
18
19       <input>
20         <local name='Data_amount' ref='d' />
21       </input>
22
23       <output name='Data_rx'>rx</output>
24       <output name='Data_tx'>tx</output>
25
26       <fsm>
27         <transaction name='TX_START' from='Idle' to='Transmitting'>
28           <update metric='Data_tx'>tx + d</update>
29         </transaction>
30         <transaction name='TX_END' from='Transmitting' to='Idle'>
31         </transaction>
32         <transaction name='RX_START' from='Idle' to='Receiving'>
33           <update metric='Data_rx'>rx + d</update>
34         </transaction>
35         <transaction name='RX_END' from='Receiving' to='Idle'>
36         </transaction>
37         <transaction name='INIT' from='' to='Idle'>
38           <update metric='Data_rx'>0</update>
39           <update metric='Data_tx'>0</update>
40         </transaction>
41       </fsm>
42     </model>
43   </models>
44
45   <profile>
46     <measure name='My_measure' observations_number='5'>
47       <metric name='Data_tx' device='UART' ref=t/>
48       <metric name='Data_rx' device='UART' ref=r/>
49       <metric name='Time' device='MICRO' ref='T' />
50       <compute name='Tx_speed'>t / T</compute>
51       <compute name='Rx_speed'>r / T</compute>
52     </measure>
53   </profile>
54 </monitor>
55

```

The microcontroller model (lines 5-10) is pretty simple. It doesn't have any state variable or constant but it takes as input the function “osKernelSysTick” declared in the header *cmsis_os.h*; moreover line 7 states that every time in a formula of this model appears the term “now”, it refers to this input function. This model provides as output the extra-functional property named “Time” (line 9), computed using the formula “now()”. In order to interpret the formula, the

framework tokenizes the formula and associates the term “now” to the input function. It is important to stress the fact that the framework analyses only words that might be valid C identifiers, numbers or symbols are ignored by the analyse process. In this way it is possible to use also C functions that have parameters. The functions that are declared in the *math.h* header file, are automatically identified by the framework, thus it is possible to use standard mathematical functions, such as “abs(n)”.

The UART model is described in Section 3.4.3 and the lines 13-42 are its translation in XML. The state of the model (lines 14-17) is composed by the variable “Data_rx” with reference “rx” and the variable “Data_tx” with reference “tx”. The input of the model (lines 19-21) is defined as the local parameter “Data_amount” with reference “d”. Since this input is local, it may be used only on the transactions formula. The output of the model (lines 23,24) compute the “Data_tx” and “Data_rx” extra-functional properties that represent, respectively, the amount of transmitted and received data. In this model the output is directly represented by the state of the model, thus the output formulas refer directly to the state variable of interest. The FSM of the model (lines 26-41) is defined by stating all its transactions. The transactions “TX_START” and “RX_START” update the internal state of the model using the input parameter “d” that is local with respect to each transaction. On the other hand, the transaction “INIT” has an empty origin state (line 37), for the framework this states that this transaction is the initialization of the framework. Thus, all the formula of this transaction are executed during the framework initialization. For each FSM there must be at most only one initialization transaction.

In this example, the system designer is interested only in a single measure (lines 47-53) named “My_measure”. This measure is a list of the metric of interest for the designer and the list of extra-functional properties that are derived from the observed one. In this case the receiving speed (line 51) and the transmitting speed (line 52) are computed as the amount of data, received or transmitted, over the elapsed amount of time. It is important to stress the fact that the value of a measure refers only to a start point and a stop point on the lifespan of the system, instead the value provided by the models refers to the whole uptime of the system.

3.4.5 Framework C interface

This section describes in details the framework C interface towards the target application. The interface is partitioned in three services: the monitor service, the model service and the profiler service. The C interface is implemented using macros that will expands on a generated function, thus in the remainder of the document, every parameter of the macros must be used without quote.

3.4.5.1 Model service

The model service is the back-end of the framework and enables the application to notify the framework about the events that affects the system models. The framework exposes a single macro to define a single transaction of a FSM of a model. The syntax is the following:

```
EFP_MODEL_UPDATE(<device>, <event>, ... )
```

Where “<device>” is the name of the device (object of the event) and “<event>” is the name of the transaction. The variadic parameters of the macro are the local parameters that the transaction may require. The transaction actually updates the model internal state only if the transaction is generated when the FSM of the model is in the correct starting state. If the

transaction is generated when the FSM of the model is in a wrong starting state, the transaction doesn't produce any effect.

Internally the macro expands in the function call generated by the framework tool. If the transaction requires some parameters, they will be passed as arguments to the function. The macro and the related functions are declared in the header file *efp_model.h*, while their definition is in the file *efp_model.c*. Along with the macro definition, the tool adds a comment that states all the possible transaction with their parameters. For instance, using the example XML configuration described previously, the framework will generate the following transaction functions in the file *efp_model.c*:

```
void efp_model_UART_init( void )
{
    UART_Data_rx = 0;
    UART_Data_tx = 0;
}

void efp_update_UART_TX_START( int Data_amount )
{
    if ( UART_current_state == 2 )
    {
        UART_Data_tx = UART_Data_tx + Data_amount;
        UART_current_state = 1;
    }
}

void efp_update_UART_TX_END( void )
{
    if ( UART_current_state == 1 )
    {
        UART_current_state = 2;
    }
}

void efp_update_UART_RX_START( int Data_amount )
{
    if ( UART_current_state == 2 )
    {
        UART_Data_rx = UART_Data_rx + Data_amount;
        UART_current_state = 0;
    }
}

void efp_update_UART_RX_END( void )
{

```

```
if (UART_current_state == 0)
{
    UART_current_state = 2;
}
}
```

3.4.5.2 Monitor service

The monitor service provides to the application the values of the extra-functional properties of interest. This service exposes two macros: the first one retrieves the value of the extra-functional property and the second one initializes the framework.

These macros and the related functions are declared in the header file *efp_monitor.h*, while their definition is in the file *efp_monitor.c*.

Framework initialization

The syntax for the initialization macro is the following:

```
EFP_FRAMEWORK_INIT()
```

This macro initializes all the state variables of the models and set the FSMs to their initial state. This macro should be called before using any other feature of the framework.

For instance, using the example XML configuration described in before, the framework will generate the following initialization function in the file *efp_monitor.c*:

```
void efp_monitor_initialize( void )
{
    efp_model_UART_init();
}
```

Retrieving the extra-functional property value

This macro retrieves the value of one extra-functional property exposed by a model.

The syntax of the macro is the following:

```
EFP_MONITOR_GET(<device>, <metric>)
```

where “<metric>” is the name of the extra-functional property and “<device>” is the device object of the property.

Internally the macro expands in the function call generated by the tool. The value of the extra-functional property is obtained using the formula of the model that provides it. The tool that automatically generates the framework is able to find the correct model. Along with the macro definition, the tool adds a comment that lists for each device all the exposed extra-functional properties.

For instance, using the example XML configuration described before, the framework will generate the following functions to retrieve the extra-functional properties value in the file *efp_monitor.c*:

```
unsigned long intefp_get_metric_MICRO_Time( void )
{
    return KernelSysTick();
}
```

```
unsigned long intefp_get_metric_UART_Data_rx( void )
{
    return UART_Data_rx;
}
```

```
unsigned long intefp_get_metric_UART_Data_tx( void )
{
    return UART_Data_tx;
}
```

3.4.5.3 Profile service

This service is the front-end of the framework and enables the application to monitor a defined region of code surrounded by two macros that starts and stops the measure. If the monitor service provides to the application the current value of the extra-functional properties, the profiling service provides the application the difference between the starting value of the extra-functional property and the stopping value.

Moreover, the profile service is able to compute indirect measures that depend on the value of the monitored ones, for instance the power consumption if it is defined as the energy consumption over the elapsed time. Conversely, the monitor service provides to an application how much energy is consumed and how much time is elapsed in total. Dividing those values provides to an application the average power consumption over the up-time period.

Internally, the profile service uses a circular buffer to store the observed measures and to provide to the application the average of the observations. The size of the buffer is defined in the XML configuration file and might be different for each measure. Every measure may be composed by several extra-functional properties from different devices.

The profile service exposes to the application three macros: the macros that start and stop a measure; and the macro that retrieves the average values of the extra-functional properties. These macros and the related functions are declared in the header file *efp_profiler.h*, while their definition is in the file *efp_profiler.c*.

Start a measure

This macro begins a measure. For every extra-functional property that is directly observed, the framework stores its current value as the starting value. The syntax of the macro is the following:

```
EFP_PROFILER_BEGIN(<measure>)
```

where “<measure>” is the name of the measure, as defined in the XML configuration file.

Stop a measure

This macro ends a measure. For every extra-functional property that is directly observed, the framework computes the difference between the starting value and the current value and stores this information in the circular buffer. After these operations, for every indirect measure, it computes their value using the values of the observed extra-functional properties. The syntax of the macro is the following:

```
EFP_PROFILER_END(<measure>)
```

where “<measure>” is the name of the measure, as defined in the XML configuration file.

Get the average

This macro retrieves the average value of the extra-functional properties that belong to a measure. The average is computed over the circular buffer. The syntax of the macro is the following:

```
EFP_PROFILER_GET_AVARAGE(<measure>, ...)
```

where “<measure>” is the name of the measure, as defined in the XML configuration file. The variadic parameters of the macro are output parameters for the extra-functional properties average value. For every property of the measure, there must be a reference to a variable. After that the macro has been called, the variables will holds the average value of the extra-functional properties. Along with the macro definition, the tool adds a comment that lists for each measure, the observed extra-functional properties.

For instance, using the example XML configuration described previously, the framework will generates the following function declaration to retrieve the values of the “My_measure” measure in the file *efp_profiler.h*:

```
void efp_profiler_get_average_My_measure(  
    unsigned long int* UART_Data_tx,  
    unsigned long int* UART_Data_rx,  
    unsigned long int* MICRO_Time,  
    unsigned long int* Tx_speed,  
    unsigned long int* Rx_speed );
```

3.4.5.4 Metamodel run-time execution example

The infrastructure describe above has been integrated into the full application considered within the Use Case 2 (Automotive Use Case) and some examples of the results that have been obtained are discussed in the following.

The extra-functional properties monitoring infrastructure has been configured to collect all the information needed by the run-time manager BBQLite to take its decision concerning the functions to activate/deactivate and the operating modes of the microcontroller and the devices. At run-time, such information are measured asynchronously, i.e. when the corresponding event is generated, and collected every 10ms to be then used by BBQLite.

All these data is of course used internally, but using the communication protocol that has been defined between the sensor node and the main ECU, they can also be sent out for debugging, investigation and information. Given the limited bandwidth of thee serial interface (usually 19200 baud) the data can only be collected with slower frequency. The following trace shows an example of data collected with a period of 1s and exported through the serial channel thanks to the monitoring infrastructure developed in the project. It should be noted that some figures have been arbitrarily multiplied by a large numeric constant in order to appreciate the values, which, with real values, would have been very small.

...									
E_LIS	16080	BW_RX	9	BW_TX	87	T_CPU	200	E_BAT	983545
E_LIS	24205	BW_RX	36	BW_TX	186	T_CPU	300	E_BAT	974943
E_LIS	32325	BW_RX	105	BW_TX	377	T_CPU	400	E_BAT	965852
E_LIS	40410	BW_RX	141	BW_TX	482	T_CPU	500	E_BAT	957239
E_LIS	48465	BW_RX	201	BW_TX	667	T_CPU	600	E_BAT	948264
E_LIS	56605	BW_RX	270	BW_TX	858	T_CPU	700	E_BAT	939153
E_LIS	64740	BW_RX	339	BW_TX	1049	T_CPU	800	E_BAT	930047
E_LIS	72795	BW_RX	432	BW_TX	1320	T_CPU	900	E_BAT	920629
E_LIS	80950	BW_RX	501	BW_TX	1511	T_CPU	1000	E_BAT	911503
E_LIS	89055	BW_RX	601	BW_TX	1696	T_CPU	1100	E_BAT	902358
E_LIS	97140	BW_RX	661	BW_TX	1881	T_CPU	1200	E_BAT	893353
...									

Figure 24. Example of trace of extra-functional properties

A longer trace has also been generated (again, with scaled figures) to generate the plots reported in the following. With scaled figures, the systems enters an intermediate power consumption stage at approximately 20s and the low-power mode at approximately 155s.

The first figure shows the bandwidth usage during realistic communication. It is worth noting that the bandwidth does not change when the system changes power consumption mode, since the communication with the main ECU is essential for proper operation.

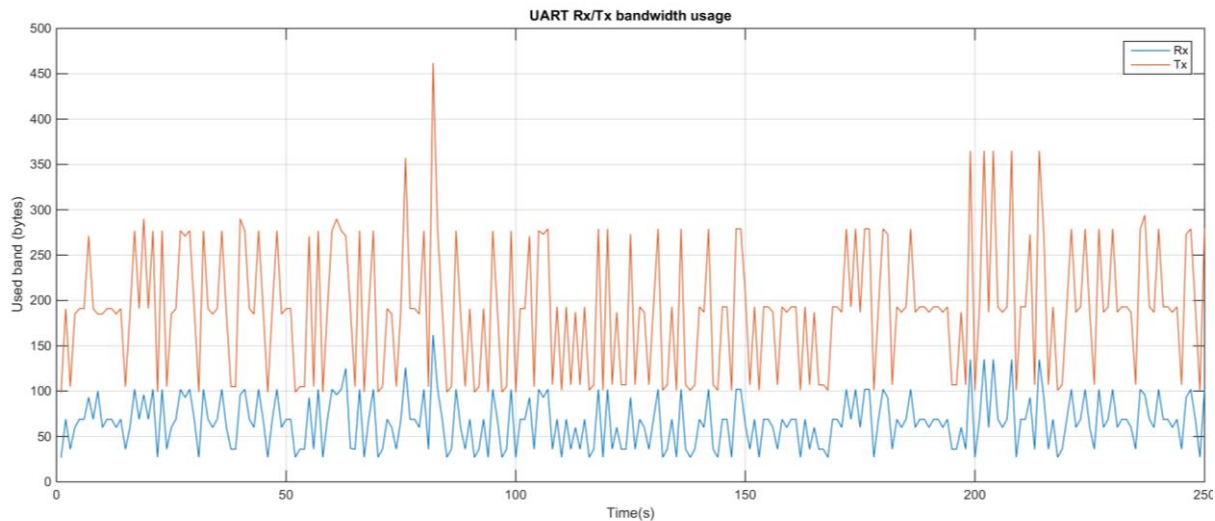


Figure 25. Extra-functional properties: UART bandwidth usage

The plot below shows the absorbed energy (as a fraction of the total battery energy) over time, when the run-time manager is disabled. This plot is representative of the original application, before the run-time manager BBQLite was introduced.

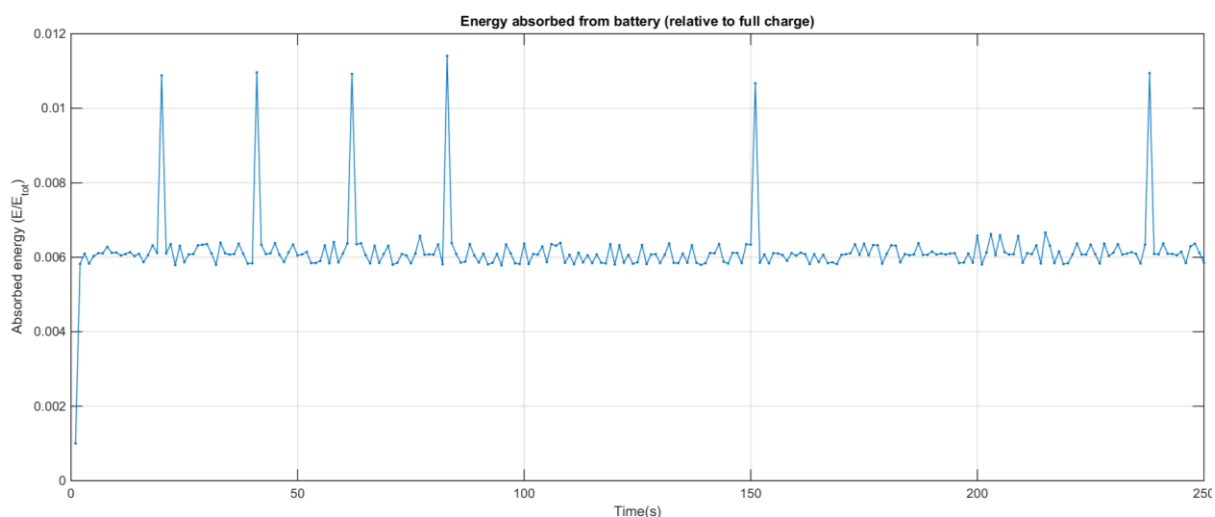


Figure 26. Extra-functional properties: relative energy absorption, no run-time management

Enabling BBQLite the power consumption is reduced depending on the charge status of the battery, as the figure below shows. In the example, the change of the functional operating mode is triggered by the charge status of the battery. By suitably scaling the power consumption figures, the discharge process can be forced to be very rapid. In particular, the switch between operating modes happens at approximately 20s and 155s.

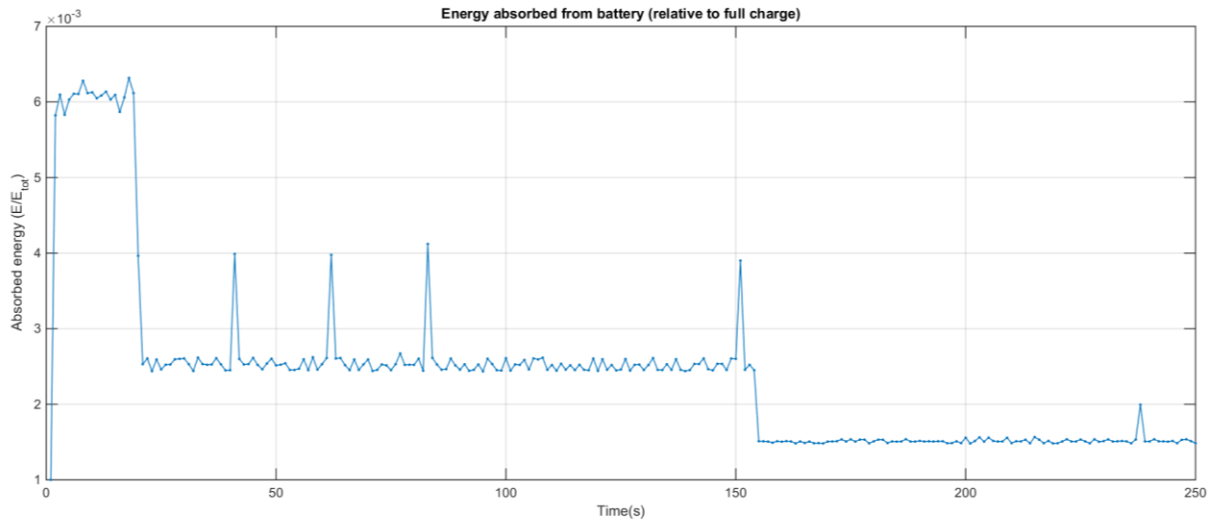


Figure 27. Extra-functional properties: relative energy absorption, with power management

Finally, the last figure shows the amount of remaining energy, with and without the intervention of BBQLite. It is worth noting that the very “clean” shape of these figures is due to the fact that several minor effects are not modelled by the extra-functional monitoring infrastructure, as much less relevant to the overall power consumption.

Again, the figure clearly shows how the rate of change of the remaining energy has different slopes, depending on the three operating modes.

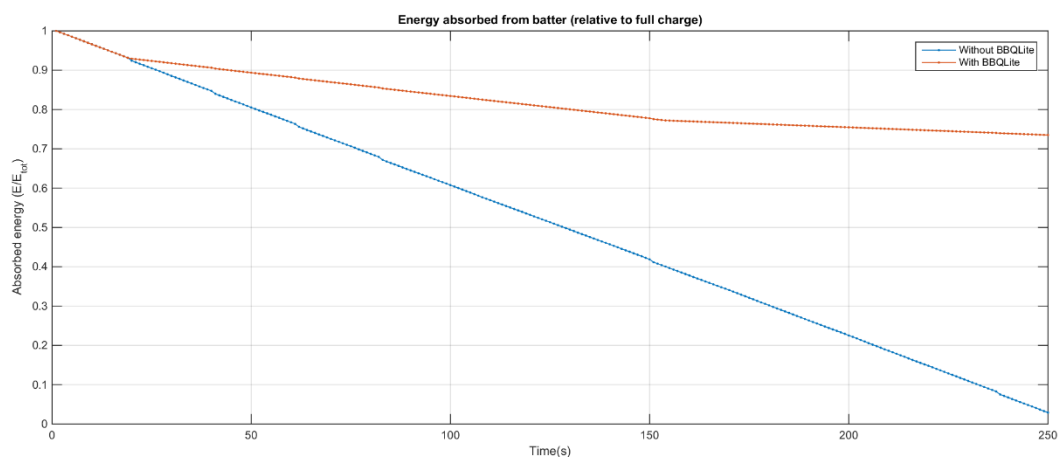


Figure 28. Extra-functional properties: battery remaining energy, with power management

Again, the trace example including the related figure shown before should be used as a test case for what can be extracted by using the monitoring system. As mentioned before, the monitored data coming from the EFP monitoring framework are used both to support the external monitoring through the serial interface but also to support the adaptivity decisions taken by the BBQLite power-manager developed in Task 3.2.

3.5 Sensor modelling and monitoring [INTEL]

The Intel Docea power and thermal modelling and validation tools implemented in CONTREX include modules specifically designed to simulate power and thermal sensing and help analyse power and thermal management behaviour. The tool power and thermal physical simulators indeed provide temperature and consumed current estimates for all parts included in the system that is modelled, but the power and thermal manager usually uses values coming from specific sensors. The values may represent the same physical quantity (temperature and current), but at different places in the designed system or sampled at different times (typically on management routine timer ticks) than the ones available in physical simulators. The values may alternatively represent other physical quantities, in the case of reliability and degradation sensors. To enable users to simulate and validate power and thermal management, Intel thus developed and integrated in the tools modules to define and use temperature and current sensors.

Only thermal sensor modelling is actually implemented in CONTREX using the Intel Docea tools (Intel® Docea™ Thermal Profiler and Intel® Docea™ Power Simulator). For the sake of conciseness the rest of this section thus only deals with this topic.

In Intel® Docea™ Thermal Profiler a temperature probe is a point anywhere in the modelled system for which the simulator reports temperature throughout the simulation. This mechanism is used especially to monitor and report temperature at the specific sensor locations.

As illustrated on the figure below, the definition of a probe consists of:

- A name: unique in the system and used to identify the probe
- A description: user description of the probe (positioning, role, ...)
- A system part (optional): the system part, ie the volume, the probe is attached to 3D coordinates: coordinates on the x, y and z axes defining the probe position probe in the system

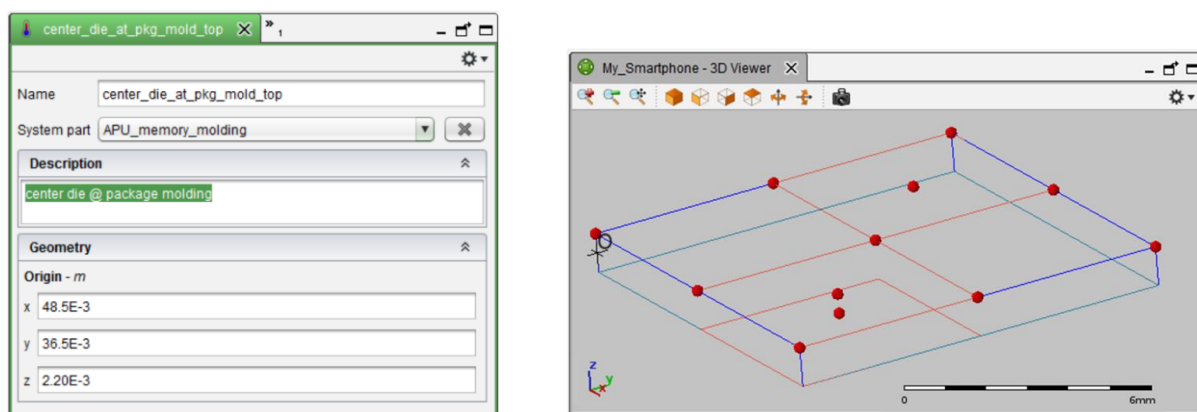


Figure 29. Thermal probe editor in Intel® Docea™ Thermal Profiler.

A temperature that is probed, ie that is measured at the probe location, can be exploited during the simulation phase and/or the post-processing phase.

Exploiting thermal probes during the simulation phase:

Exploiting a probe during the simulation phase means you have the thermal simulator of Intel® Docea™ Thermal Profiler being connected to another simulation environment (usually a behavioural or functional virtual platform), so both simulators run together, and you want to convey the probed temperatures to the other environment. Establishing the connection between the two simulators is carried out using the API (Application Programming Interface) dedicated

to this purpose in Thermal Profiler. Using this API, the simulation environment will receive a *StepResult* object from Thermal Profiler each time it calls the *getTemperatures* function. This object contains the thermal results computed between two consecutive calls to *getTemperatures*, including the temperature coming from the probes defined in the simulated system. The figure below shows the related API commands (in C++ programming language).

```
#include <StepResults.hpp>
```

Collaboration diagram for idtp::profiling::StepResults:

[List of all members.](#) [\[legend\]](#)

Public Member Functions

```
std::vector< float > getAvailableTimings () const
float getSourceAverageTemperatureAt (float timing, const std::string &systemPartName, const std::string &sourceName) const
float getProbeTemperatureAt (float timing, const std::string &probeName) const
```

Detailed Description

A **StepResults** object holds the temperature results of all the system observables (sources + probes) at the timesteps that are the bounds to the last simulated time interval.

```
float idtp::profiling::StepResults::getProbeTemperatureAt ( float timing,
                                                         const std::string & probeName
                                                         ) const
```

Returns the temperature corresponding to the provided probe at the given timing.

Parameters:
timing,: the timing for which the source region temperature is queried.
probeName,: the name of the probe for which the temperature is queried.

Returns:
: the requested probe temperature in deg. C

Figure 30. Samples of the API documentation describing how to exploit thermal probing in an Intel® Docea™ Thermal Profiler simulation.

Exploiting thermal probes during the post-processing phase:

In the case of offline analyses, ie when the simulation has ended and one is re-opening the simulation results for analysis and further post-processing, the probed temperatures can be exploited in the thermal simulation results viewer through:

- Either statistics presented in the key indicators table
- Or in the chart view section displaying the variations of the probes' temperatures over the time

The viewer is shown in the figure below.

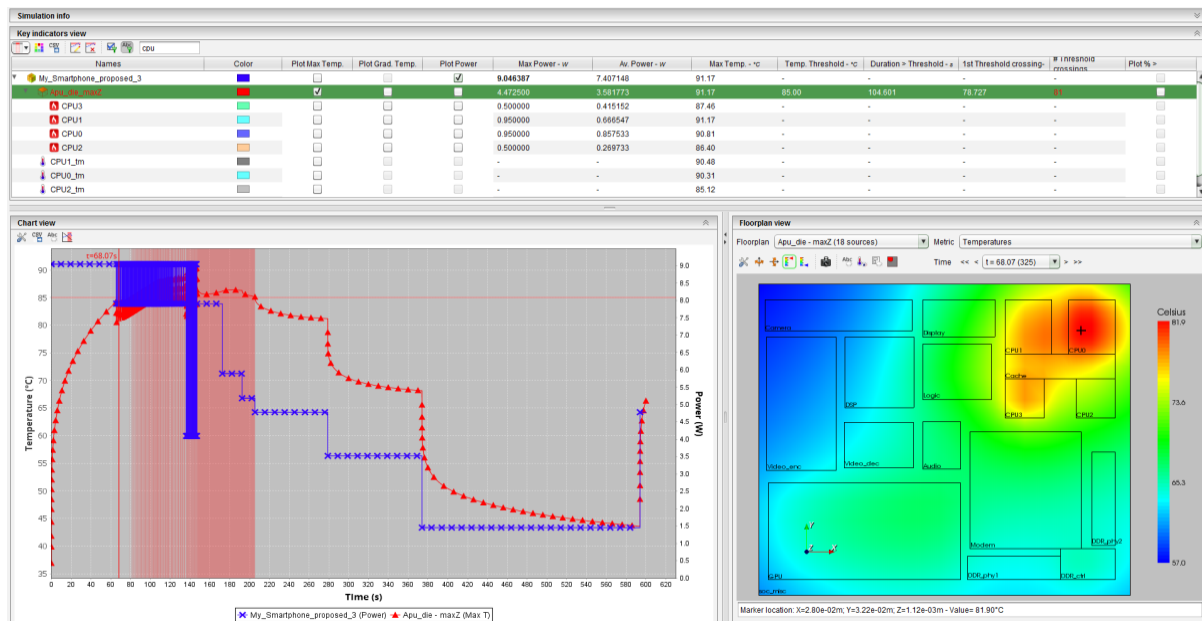


Figure 31. Intel® Docea™ Thermal Profiler thermal simulation viewer for offline analyses.

These results can be exported in csv (comma-separated value) format or can be accessed through a Python API for post-processing, transfer to an external tool or reporting in any appropriate shape. The viewer provides besides a threshold-checking feature. Based on a user specified temperature threshold applied to probes (also applicable to the so-called thermal domains locating where power is dissipated in the simulated system), one can get statistical business metrics like the time spent above the threshold or the number of times the threshold is crossed. These metrics are for instance meaningful when evaluating the efficiency of a dynamic thermal management algorithm or when estimating the reliability of a system.

In conclusion, the Intel Docea toolscan model systems equipped with sensors. This feature provides the tool users in CONTREX with the ability to observe temperature variations in simulation as the sensors would actually see them in reality. Coupled to the OVP-based virtual platform, it moreover provides a means to simulate how power and thermal management software react to temperature as the actual software running in the system would do.

3.6 Battery modelling and monitoring [PoliTo]

For devices like batteries (or similar energy storage devices - ESDs), whose main operation consists of storing energy and delivering it according to requests from generic “loads”, the simulation of the energy flow represent conceptually a “functional” aspect. In other words, the flow of energy in storage devices is in fact a *functional property*, i.e., it describes its behaviour. For this reason, accurate battery modelling and monitoring requires the construction of a *native power simulation*, i.e., of a framework where power is explicitly simulated as a signal.

Moreover, the complex interaction between power and functionality with other extra-functional properties should be enforced. It is well known that temperature is affected by power dissipation and vice versa, but it is less known that temperature also impacts the capability of batteries of delivering their charge [1][2][11]. Similarly, batteries also exhibit reliability effects in the form of aging, which modify battery capacity and thus its capability of delivering energy over multiple discharge cycles [12].

These two requirements (need of a “native” power simulation and complex inter-relation among different metrics) lead us to envision a simulation framework covering multiple domains. Indeed, *runtime simulation* of multiple functional and extra-functional properties allows to highlight the mutual dependencies and to reproduce and validate power behaviour in the context of overall simulation. To achieve this result, the simulated system is structured according to different views, called *layers*, each focusing on one specific property. This allows to handle information related to each property independently and to simulate at the same time different aspects of the system.

D3.4.1 presented the overall framework, and D3.4.2 focused on the definition of the native simulation of power. The current deliverable extends the framework with temperature simulation, to increase the accuracy of battery simulation.

3.6.1.1 Proposed framework for native power simulation with extra-functional properties

The simulated system is structured according to different views, called *layers*, each focusing on one specific property. This approach allows to handle information related to each property independently and to simulate at the same time different aspects of the system, yet keeping the overhead and complexity low thanks to a unified approach for each layer. Figure 32 exemplifies the framework by proposing the simultaneous simulation of power with functionality, temperature and aging.

It is important to note that layers evolve independently, even if they share information to model mutual influence. As a result, it is possible to restrict simulation to one single layer, i.e., the power layer, by breaking inter-layer dependencies through the adoption of traces.

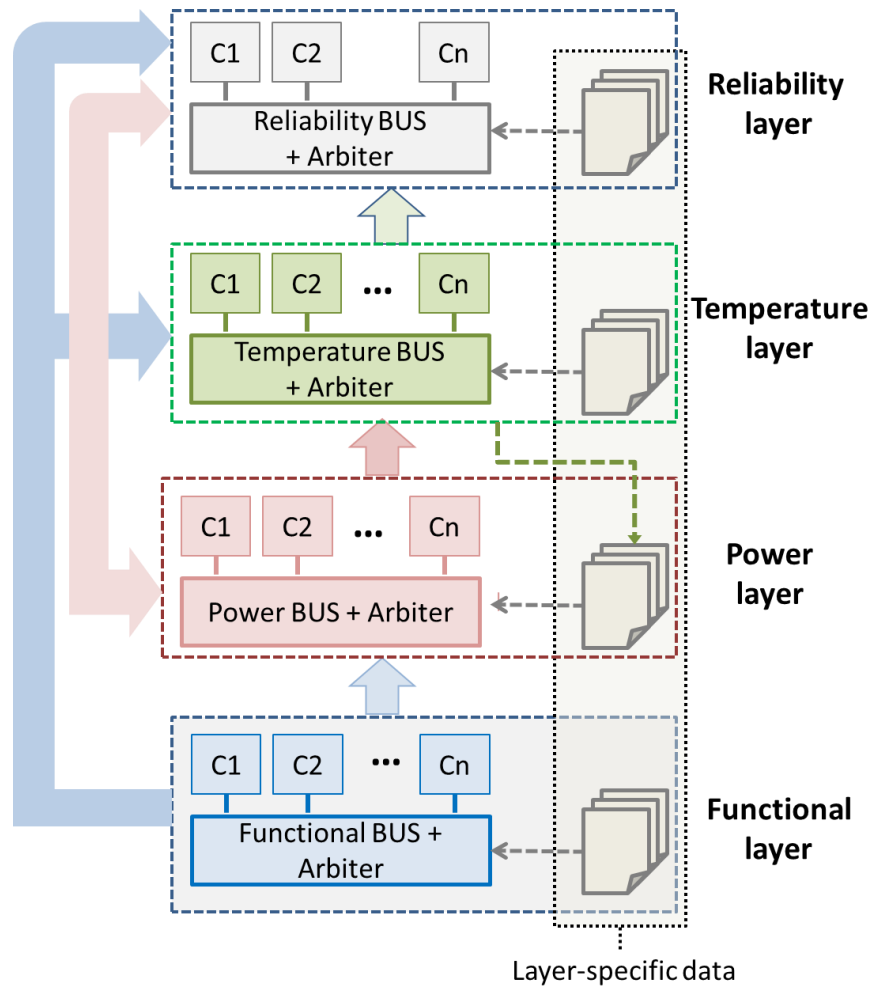


Figure 32. Layered framework for power simulation with the support of mutual effects from other extra-functional properties.

All layers have a similar structure, which reflects the physical architectures of digital systems:

- *Layer-specific signals:* main property-specific signals that are tracked by the simulation engine in each layer.
- *Inter-layer signals:* signal used to exchange information between layers and to represent the mutual influence between properties.
- *Role of bus and bus-arbiter:* the bus is in charge of conveying information and of reproducing energy flows, while the arbiter is in charge of aggregating information local to each component to determine information flow and overall property-specific information.
- *Layer-specific data/information:* information related to additional data that are essential to run the simulation but not involved with the simulation semantics.

The proposed framework allows thus to reproduce and validate power behaviour in the context of overall simulation. The framework can then be restricted to the sole power layer, to perform further simulations and to be inserted in the overall modelling and exploration flow. On the other hand, simulating the power layer simultaneously with the other layers (i.e., functionality, temperature and reliability) allows to get a more accurate estimation of the power behaviour, and to capture the mutual influences of different aspects of the same system.

Table 1 highlights the main characteristics of all layers.

Table 1. Layer-specific mapping of concept onto extra-functional properties.

Property	Layer-Specific Signals	Inter-Layer Signals	Role of bus and arbiter	Layer-specific data/information
Power	Voltage, Current	Workload (functional layer), Temperature (temperature layer)	The bus models the energy paths among the components (guaranteeing energy conservation) and it provides a reference voltage for the overall system. The arbiter monitors the energy flow and it can be augmented with policies for managing system components (e.g., to attach a battery to a load whenever the power sources do not provide the loads with enough power).	Environmental information that influence the behaviour of power components, e.g., irradiance for a photovoltaic cell. These data can be provided either as parameters or as traces.
Temperature	Temperature	Power (power layer), Workload (functional layer)	The bus is in charge of conveying all technological and power information to the bus manager. The manager determines the temperature of each component by solving the electrical circuit equivalent of the overall system thermal network.	(1) <i>Geometrical data</i> : i.e., floorplan; width, height and thickness of each component. (2) <i>Technology data</i> : physical properties like thermal conductivity (k), heat transfer coefficient (h), and thermal capacitance per volume (c).
Reliability	Failure Rate	Temperature (temperature layer), Power (power layer), Workload (functional layer)	The reliability bus conveys the failure rates relative to individual components to the reliability bus arbiter. The latter determines overall system failure rate as a function of the local components failure rates.	(1) <i>Technology data</i> : Reliability models depend on material-related quantities that must be provided by the designer. (2) <i>Topology information</i> : overall reliability depends on the relationship between components. The specified topology allows building the aggregation function that determines overall system failure.

It is important to note that extra-functional properties monitoring is implicit in the framework, both thanks to the power models and the reconstruction of mutual influences. Furthermore,

property-specific buses provide a first control infrastructure, as they allow to apply property-specific policies, still taking into account overall system evolution. Therefore, although the overall paradigm emerges from the specific need of tracking power in the presence of devices that store and deliver energy and not just devices that consume it, the approach can be extended to (1) other non-functional properties, and (2) to other type of components.

In the context of the CONTREX design space, it is worth emphasizing that this methodology is in some sense orthogonal to the abstraction levels. It can be applied at the SoC, node, or system level. What differs in each level is simply the model of computation (MoC) of the underlying simulator.

3.6.1.2 Characteristics of the temperature layer

Thermal simulation requires to trace how temperature evolves over time, as an effect of the changing operating conditions (e.g., power consumption). To this extent, the temperature layer traces temperature as layer specific signal (Table 1). This guarantees a good view of the thermal flow, and it allows to track thermal behaviours of all system components.

Over time, a variety of tools have been proposed for estimating temperature (both transient and steady-state), with the goal of enhancing the design flow with increased reliability and knowledge of the underlying physical mechanisms. All these tools solve the same problem (i.e., a 2D or 3D heat diffusion equation) and basically differ in terms of their granularity and of the type of solver employed, with the latter aspect affecting their accuracy [1][4][5]. The de-facto standard for thermal simulation is *HotSpot* [5], a tool based on the circuit-equivalent of a thermal network, which achieves a good trade-off between granularity and accuracy. *HotSpot* uses the chip floorplan and thermal package information to build an equivalent circuit description, that is solved over time on given traces of power dissipation.

The adoption of an equivalent circuit description enforces the construction of a centralized temperature model for the whole system. This modifies the information flows and the role of bus and bus manager w.r.t. the power layer.

The *temperature-specific bus* is in charge of collecting all necessary information for the thermal model, including power dissipation of components and thermal-specific data. The *temperature arbiter* encapsulates the electrical circuit equivalent model, thus determining the temperature of each component over time, together with the heat flows in the system. This solution allows to accommodate temperature in a bus-based template.

Finally, *temperature-specific data* involves all information not strictly related to temperature that affects the evolution of the thermal model. Information ranges from geometrical data (e.g., position of components in the floorplan; width, height and thickness of each component) to technology data (i.e., physical properties like thermal conductivity and thermal capacitance per volume).

Template of the temperature layer

At temperature layer, all components have the same role w.r.t. the thermal information flow. Components forward necessary power dissipation information to the temperature bus, and they receive their updated temperature value. As a result, there is no distinction in terms of interface (Figure 33). All components will have two ports:

- One output port for carrying the component *power consumption*;

- One input port for receiving the estimated *temperature* of the component over time.

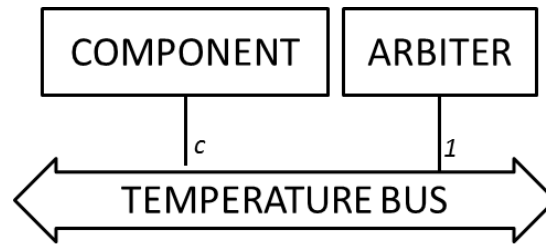


Figure 33. Template of the temperature layer.

3.6.1.3 Languages, Formats and Technologies

The realization of the proposed layered framework implies the adoption of languages suitable for multiple domains and abstraction levels. A winning choice would be the adoption of standard open languages, as they would guarantee interoperability and extensibility.

The central role of buses in each layer makes functional simulators and languages a natural fit for the modelling of the proposed approach. The construction of the proposed framework will thus require (i) to identify the interface of system components on each bus and (ii) to build a model of the property-specific evolution for each component.

Interfaces of components and buses can be modelled by adopting the IP-XACT standard, a standard XML format for describing interfaces of digital IPs and systems [13]. IP-XACT supports a set of different description schemas: a component description describes the interface of an IP, detailed as a list of ports, while a design description outlines the components instantiated in a system and their connections (defined by means of port binding constructs). The IP-XACT extension for analog mixed signal modelling allows to support also physical and non-digital interfaces [14].

The choice of a language and simulator for *models implementation* is more crucial and complex. The choice fell on SystemC-AMS for its support to a wide range of domains and abstraction levels, and for the definition of a range of Models of Computation (MoCs), that allow to cover a wide range of domains and descriptions with a single language [15]. Timed Data-Flow (TDF) models discrete time statically scheduled processes. Linear Signal Flow (LSF) supports the modelling of continuous time non conservative behaviours. Finally, Electrical Linear Network (ELN) models electrical networks through the instantiation of predefined primitives, e.g., resistors or capacitors.

Interface modelling with IP-XACT

Power and temperature ports are real values modelling a continuous physical evolution. They are thus represented by using the IP-XACT extension for analog mixed signal modelling, that supports also physical and non-digital interfaces [14].

Ports are associated with a default value, annotated with the measure unit (e.g., Celsius) and a prefix (e.g., kilo and milli). Note that the measure unit tag is especially useful with temperature ports, as it allows to remove ambiguous quantities, by specifying whether a port expresses temperature in Celsius, Kelvin or Fahrenheit. This allows to build semantic checks on top of the IP-XACT files, to guarantee that system connections are sound in terms of adopted measure

unit. 23 compares the IP-XACT file modelling a digital input port (a) and for modelling a temperature port (b).

<pre> <port> <name>data</name> <wire> <direction>in</direction> <vector> <left>3</left> <right>0</right> </vector> <typeName>bit_vector </typeName> </wire> </port> </pre>	<pre> <port> <name>temp</name> <wire> <direction>in</direction> </wire> <vendorExtensions> <value unit="celsius" prefix=""> 36.0 </value> <signalType>continuous-conservative </signalType> </vendorExtensions> </port> </pre>
a.	b.

Figure 34. Excerpt of IP-XACT file modelling a digital port (a) and a temperature port (b).

IP-XACT component descriptions are provided for all components of the temperature system, excluding the temperature bus. Each IP-XACT component description adopts the previously defined temperature and power dissipation ports. exemplifies these concepts on the component description of a core.

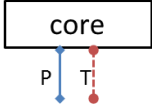
<pre> <component> <name>core</name> <model> <ports> <port> <name>P</name> <value unit="watt" prefix="milli">3.69</value> </port> <port> <name>T</name> <value unit="celsius" prefix=""></value> </port> </ports> </model> </component> </pre>	
---	---

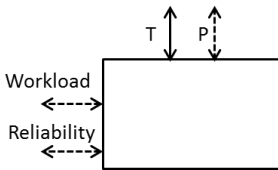
Figure 35. Excerpt of IP-XACT component description of a core at temperature layer.

IP-XACT design descriptions model connections between system components. No relevant change distinguishes this kind of descriptions from the power ones. Note that the IP-XACT design description must be modelled by the user. However, the topology is always as in Figure 33: all components are connected to the temperature bus. This does not reflect the physical contiguity between components (i.e., the floorplan).

Implementation of the temperature layer and available models

reports the main characteristics of the implementation of the temperature layer. Starting from interface definition, column *Component Interface* depicts the generic interface that applies to all components. As previously mentioned, the only layer specific signal at temperature layer is temperature, but power is also a necessary signal for the evaluation of the temperature electrical circuit equivalent model. Thus, temperature is solid (i.e., layer specific), while all other signals (including the power signal) are depicted as dashed arrows and they may be either part of the interface (i.e., \mathbb{P}) or additional information originated by traces or by the other layers (e.g., workload).

Table 2. Layer-specific mapping of concept onto power.

Property	Interface definition		Model definition	
	Component interface	Design description	Model types	Simulation semantics
Power		All components are connected to the temperature bus, as described by the IP-XACT design description. This does not reflect the physical contiguity between components (i.e., the floorplan).	Models use a RC-circuit equivalent. Temperature is estimated through a centralized model implemented in the bus manager. Components simply forward power information to the bus.	Temperature simulation mixes the ELN (for the electric circuit equivalent model) and TDF MoCs (for components).

The characteristics of the corresponding IP-XACT *design descriptions* are outlined in Column Design Description of . At temperature layer, all components are connected to the temperature bus. This does not reflect the physical contiguity between components (i.e., the floorplan, that is provided as a separate file and used for the construction of the electrical circuit equivalent model).

Moving towards framework implementation, column *Model Types* of outlines the main types of models available at state of the art. The temperature bus manager computes the centralized model, based on the operating conditions provided by the components. This forces the adoption of two different MoCs for the bus and the components: the RC model is implemented by adopting electrical linear circuit primitives (ELN), while the components can adopt the more efficient synchronous dataflow semantics (TDF).

Construction of the RC network

The electric circuit equivalent model used for evaluation of temperature can be built in SystemC-AMS by reproducing the method followed by HotSpot [5]. Figure 36 exemplifies the application to a simple case study consisting of a core, a memory, a RF transceiver and a UART device.

The construction of the RC network heavily depends on the layer-specific data: chip floorplan, necessary to determine which components are adjacent, and technology information (e.g., number of layers, materials, thermal characteristics).

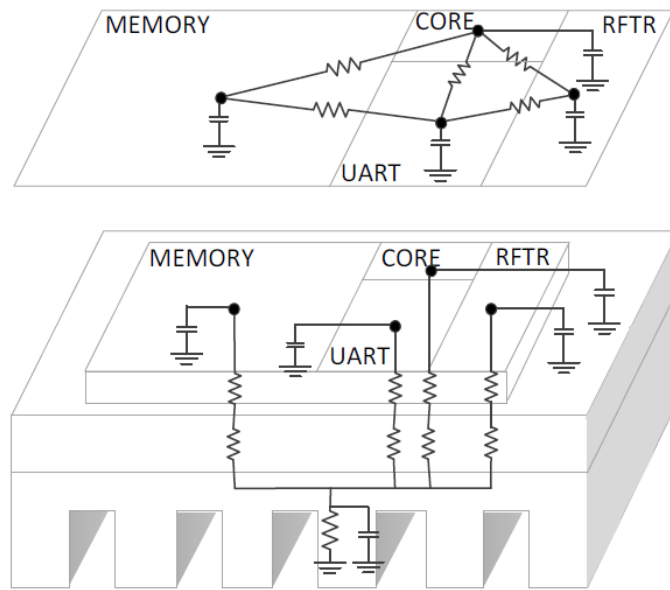


Figure 36. Example of electrical circuit equivalent model: the RC network is decomposed in a lateral model (top), representing heat transfers that occur horizontally between adjacent components, and a vertical model (bottom), that sketches the heat spread across the package layers.

Each chip component is mapped to a *RC network node*, whose current represents power consumption and whose voltage represents temperature. Additional nodes are used to represent the underlying package layers (i.e., heat spreader and heat sink). Heat transfer flow between adjacent nodes is represented by a *resistor*, whose thermal resistance is proportional to the thickness of the material and inversely proportional to the cross-sectional area and to the thermal conductivity. If the RC network is used for transient simulation, *capacitors* are connected to each node, to capture the delay before a change in power determines a change in temperature. Thermal capacitance is proportional to both thickness and area, and it depends on the thermal capacitance per unit volume.

Modelling and simulation in SystemC-AMS

Of the models of computation provided by SystemC-AMS, ELN is especially useful for the temperature layer, as it allows to build the RC circuit by using native SystemC-AMS constructs, and to delegate the computation of the RC network over time to the underlying SystemC-AMS solver. Figure 37 exemplifies the proposed approach by showing how the lateral model depicted on top of Figure 36 is implemented in SystemC-AMS.

Interface modelling.

System components are implemented as SystemC modules, featuring a TDF interface. TDF determines a fixed timestep at which ports are updated, and thus at which the RC network is evaluated. This reflects the behavior of HotSpot, that assumes that the power traces contain samples collected at fixed time steps. The interface of each component module is made of two ports: one input port to gather from the bus the evolution of temperature over time (`sca_tdf::sca_in`), and one output port to convey power consumption to the bus (`sca_tdf::sca_out`).

The temperature model is encapsulated by the temperature bus manager, that is instantiated as a single SystemC module (`SC_MODULE`, line 1), encapsulating the entire RC network. The

interface of the module is made of two ports for each chip component: one input port to gather the evolution of power consumption over time, and one output port to convey the corresponding value of temperature. The abstraction level adopted for ports is still TDF (i.e., ports are thus declared as `sca_tdf::sca_in` and `sca_tdf::sca_out` ports of type double, lines 2–3).

Body implementation

The body of the system components is implemented in TDF, and it consists of gathering the power values from the power layer (or from traces). This value is then written to the power port of each component (`P`). The temperature value received on port `T` is then used for further internal calculation, or forwarded to the other layers, e.g., to affect the evolution of power consumption and reliability.

The body of the bus manager `SC_MODULE` includes the implementation of the RC network. The flexibility of SystemC-AMS and its support for multiple levels of abstraction allows to decouple the semantics of the interface from the semantics of the actual behaviour implementation. This allows to implement the RC network by adopting the ELN level of abstraction, that natively supports electrical network elements. The construction of the SystemC-AMS thermal model is thus a one-to-one mapping of circuit elements into SystemC-AMS primitives.

Each circuit node is implemented as a SystemC-AMS ELN node (`sca_node`, lines 6–7), despite of ground, that is represented with an ad-hoc primitive (`sca_node_ref`, line 5).

Resistors are mapped to instances of the `sca_r` primitive, that represents SystemC-AMS resistors (line 12): e.g., the snapshot of code in Figure 3 shows that the resistance modeling the heat flow between the core and the memory is 215.48Ω (lines 27–30).

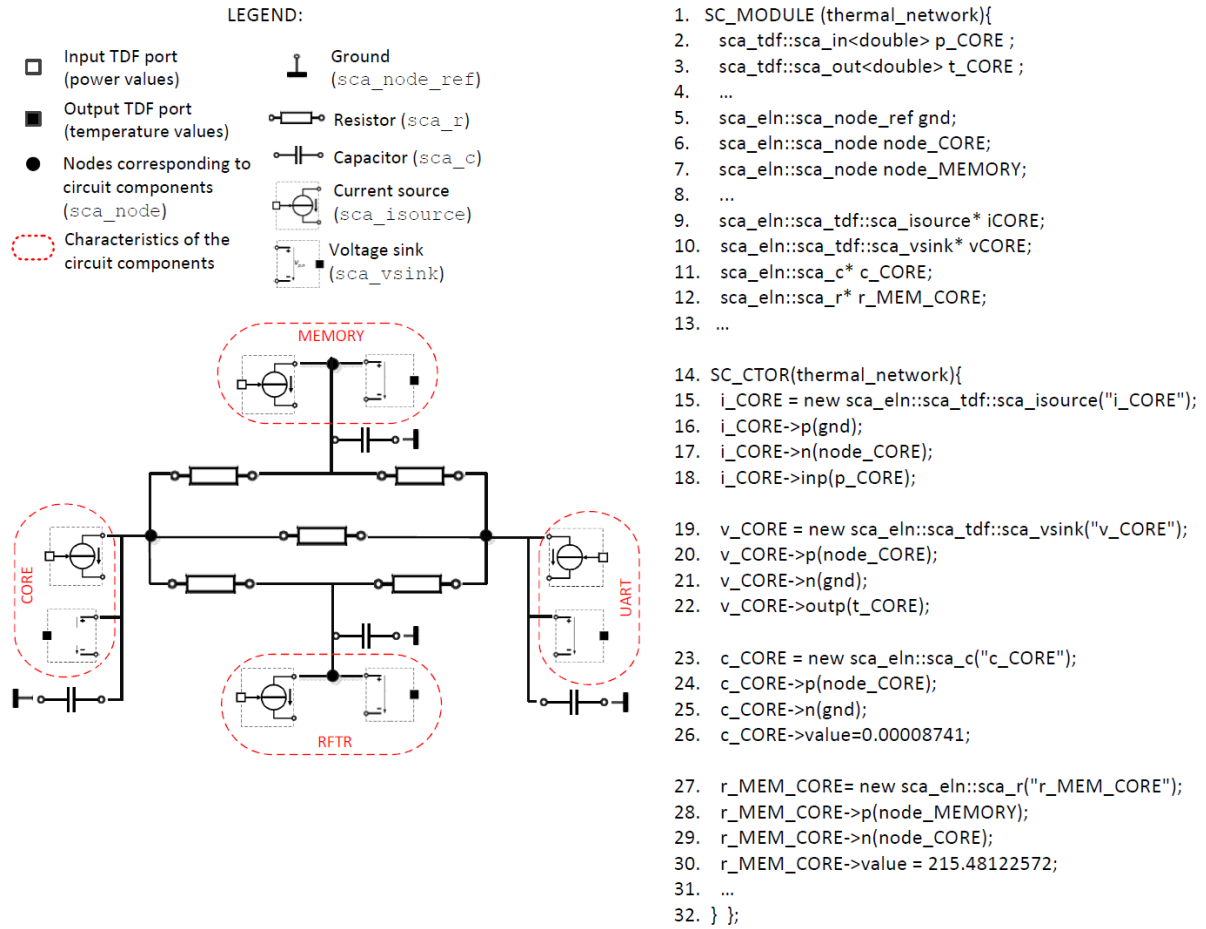


Figure 37. Implementation of the lateral model of Figure 2 in SystemC-AMS: implementation of the RC network with ELN primitives (left) and excerpt of SystemC-AMS code (right).

Capacitors are mapped to an instance of the *sca_c* primitive, that represents SystemC-AMS capacitors (line 11): e.g., the core capacitance is $874 \cdot 10^{-5}\text{F}$ (lines 23–26).

The input power ports are connected to the ELN circuit via current source primitives, that transform a numerical value into a current (*sca_isource* primitive, lines 9 and 15–18). In the pictorial representation in Figure 3, TDF ports are represented by the white square terminals of the primitive blocks.

Conversely, the temperature of each chip component is extracted through a voltage sink, that extrapolates a voltage value and makes it available on the output ports. Voltage sinks are represented with instances of the *sca_vsink* primitive, whose output terminal is connected to the corresponding output temperature port (the black square terminals, lines 10 and 19–22).

To ease the adoption of the proposed framework also at temperature layer, the construction of the RC network has been automated in a C++ tool, that takes in input two files, modelling floorplan and technology information. The tool builds the RC network and saves the generated code to a SystemC-AMS file [6].

3.6.1.4 Validation of the temperature layer

To compare our framework to HotSpot, we used five benchmarks of different sizes and complexity: the example in Figure 36 (benchmark 1), two built-in examples of HotSpot

benchmark 2 and 3), and two synthetic benchmarks (4 and 5). The latter two do not correspond to any functionality, and are only used to prove effectiveness and scalability of the proposed simulation approach. The benchmarks have been generated by replicating a number of cores, caches and memories, with typical power consumption traces.

The corresponding floorplans have been derived by using the HotSpot thermal-aware floor-planning tool. To ensure a fair comparison, we fed both HotSpot and our tool with the same power traces, to break the dependency from the power layer, and with the same floorplan and technology files.

Correctness of the RC

To test the correctness of our approach, we compared the RC network built by our approach for each benchmark with the one built by HotSpot. Table 3 reports the main characteristics of the RC networks. Since our RC network construction algorithm is based on HotSpot, it is no surprise that the *RC networks are identical*, in terms of both number of nodes, resistors and capacitors. This guarantees that both SystemC-AMS and HotSpot solve the same equations, and that any inaccuracy lies in the solver, rather than in the constructed thermal model.

Table 3. Characteristics of the computed RC networks.

Benchmark	Components (#)	Nodes (#)	Resistors (#)	Capacitors (#)
1	4	28	64	28
2	18	84	288	84
3	30	132	442	132
4	40	172	586	172
5	86	356	1,206	356

Simulation time

Table 4. Comparison of simulation time.

Benchmark	RC Network Generation		Simulation time		
	SystemC-AMS (s)	Hotspot (s)	SystemC-AMS (s)	Hotspot (s)	Speedup (x)
1	0.002	0.001	0.101	1.472	14.55
2	0.004	0.001	0.536	17.446	32.55
3	0.011	0.002	0.983	40.609	41.31
4	0.016	0.003	1.425	72.591	50.94
5	0.120	0.120	3.343	321.397	96.14

To evaluate the temporal performance of SystemC-AMS simulations, we compare separately the time to generate the RC thermal model and the time to simulate the network.

Table 4 reports generation times, and shows that the time necessary to build the thermal model is similar for HotSpot and SystemC-AMS, since both the approaches rely on the same algorithm for RC network construction. The time is slightly higher for SystemC, since it requires a further step to map the resistor and capacitor matrices to SystemC-AMS primitives, and to print the generated code onto a file. However, generation times are truly negligible.

Table 4 completes the analysis with the most relevant information, i.e., the comparison of simulation times. Data refers to a 1000ms trace with timestep 1ms. *SystemC-AMS proves to be*

much faster than *HotSpot*, with a speedup of up to 96X. The interesting aspect is that speedup scales linearly with the size of the netlist. This proves that the improved performance is due to the SystemC-AMS simulation kernel, that handles more efficiently the RC network equations. Note that the achieved simulation speedup compensates for the slightly longer generation time.

Accuracy

To prove the accuracy of our simulator, we determine the relative error w.r.t. *HotSpot*, by analysing the traces of each component. We first compared SystemC-AMS and *HotSpot* on *steady-state scenarios*. The input power traces consist of a single power value per component, corresponding to the average power dissipation of the component over time. SystemC-AMS proved to be extremely accurate, as we got a 0% error on all benchmarks. This can be explained by the similar underlying solvers, and by the fact that steady-state evaluation does not require iterations, thus avoiding the accumulation of approximations.

Table 5. Accuracy of transient temperature estimation.

Benchmark	Components (#)
1	4
2	18
3	30
4	40
5	86

The *transient scenario* is more interesting, as the RC network includes also capacitors and multiple iterations are necessary to compute the thermal trace. Table 5 shows that the average error committed on all benchmarks is far below 1%. This proves that the SystemC-AMS solver is extremely accurate w.r.t. the *HotSpot* simulation kernel. Figure 38 proves the high level of accuracy by showing that the temperature curves computed by SystemC-AMS and *HotSpot* for benchmark 1 are overlapped (top), and that the error becomes visible only by zooming in on very small time windows (bottom).

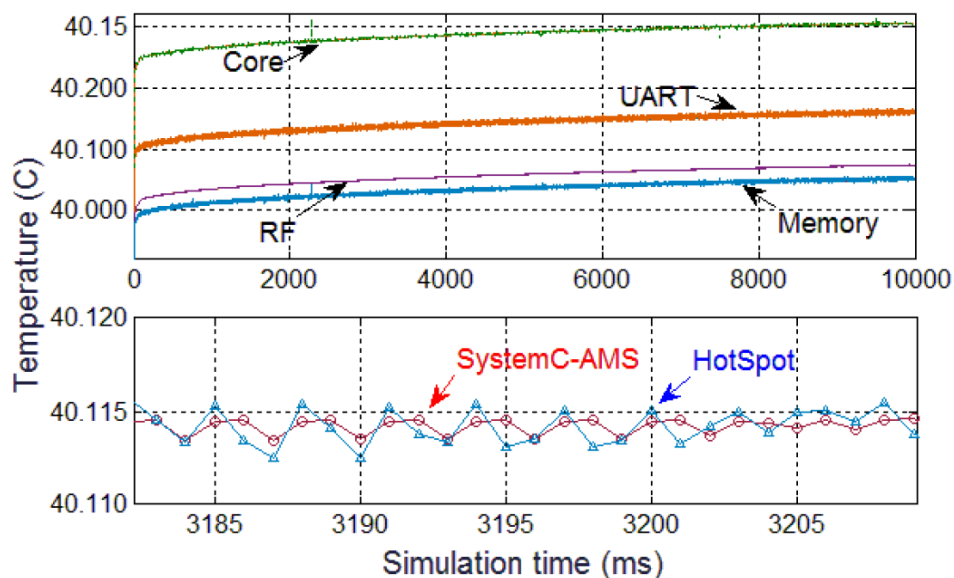


Figure 38. Transient temperature profile from SystemC-AMS and *HotSpot* applied to benchmark 1 (top) and zoomed view of the memory transient temperature profiles (bottom).

By zooming in the curves, we observed that the maximum error always happens at inflection points of the temperature profiles. This is caused by the electrical characteristics of SystemC-AMS capacitors and resistors, that result in slightly different time constants. The inspection of the traces suggests however that the less abrupt transitions resulting by SystemC-AMS simulation are more realistic than the discontinuous saw-tooth profile computed by the HotSpot solver.

Framework simulation with functionality, power and temperature layers

As a final experiment, we implemented the proposed framework for the benchmark in Figure 36, by modelling also the functional and power layers. Our goal was to dynamically derive power information for the digital core from its functional evolution [7], to determine the influence of the executed application on the thermal profile.

The power model of the core is a power state machine, that is ruled by the `pause` functional signal: as soon as the functional model hibernates, the power state machine changes state and it dynamically updates the power consumption values written in output to the thermal model.

Figure 39 traces the evolution of the resulting simulation. As soon as the `pause` signal is set to 0, the power model reacts and performs the transition from ACTIVE to SLEEP, thus lowering power dissipation from 231.4mW to 50mW. This is detected by the temperature bus manager, that computes a gradually decreasing temperature for the core, moving from 41.42°C to 40.33°C. Overall simulation lasted 6.302s and was performed in a single run with the sole support of the SystemC simulation kernel. Building the same scenario with HotSpot, or with any other custom thermal simulator, would have required the construction of a co-simulation infrastructure with an architectural or a circuit simulator.

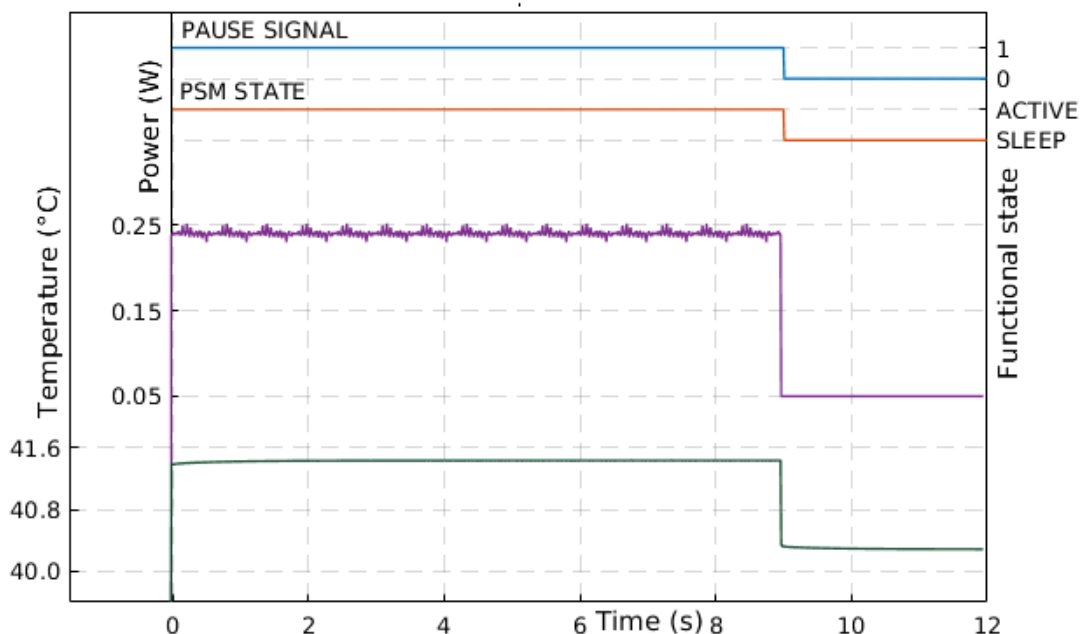


Figure 39. Effect of simultaneous simulation of functional layer, the power layer and the temperature for the benchmark in Figure 36.

3.7

3.7 xtra-functional monitoring at the cloud level [EUTH]

The cloud-based monitoring of functional and extra-functional properties is a two-steps bidirectional process relying on Kura framework and on the cloud platform. Kura is responsible for the data acquisition from the field and for the consequent publication of the acquired data on the cloud. The cloud platform is responsible for the storage and management of the collected data, for remote management of the devices deployed on the field and to provide value added business-to-business services. The cloud-based monitoring has been adopted in the automotive scenario (UC2), but this approach is information agnostic and it can be potentially used to monitor any property of the remote system.

3.7.1 Kura pervasive monitoring

Kura framework provides a set of services that, abstracting the remote device's hardware/software details and the communication related aspects, allows to collect data from the field and publish it to the cloud. This is the first part of the cloud-based pervasive system for monitoring functional and extra functional properties.

Kura framework has been developed on the OSGi framework and currently runs on two different OSGi implementations: Equinox, an open source implementation sustained by the Eclipse Foundation, and a second implementation known as Hitachi's SuperJ Engine Framework.

The OSGi applications framework provides a programming model for developing, assembling, and deploying modular applications that use Java™ and OSGi technologies. OSGi application development tools provide a way to build enterprise applications that benefit from the modularity, dynamism, versioning, and third-party library integration provided by the OSGi applications framework. For more information on OSGi and these implementations, see <http://www.osgi.org>.

OSGi specifications are defined and maintained by the OSGi Alliance, an open standards organization. The specification outlines open standards for the management of voice, data, and multimedia wireless and wired networks. The OSGi Service Platform Specification defines an open common architecture for service delivery and management using bundles.

The core part of the OSGi Service Platform defines a secure and managed Java-based service platform that supports the deployment of extensible and downloadable applications known as bundles. The specification defines a security model, an application life cycle management model, a service registry, an execution environment and modules.

An OSGi bundle is a Java archive file that contains Java bytecode, resources and a manifest that describes the bundle and its dependencies. The bundle is the basic building block when developing and deploying an application. An OSGi application emerges from a set of bundles, of different types, to provide a coherent business logic.

Kura components are designed as configurable OSGi Declarative Service exposing service APIs and raising events. While several Kura components are in pure Java, others are invoked through JNI and have a dependency on the Linux operating system.

Following the OSGi development model the services offered by Kura (including the “monitoring” related services) are provided on a per bundle/component basis. There isn’t a one to one mapping between services and bundles: it can happen that more than one service is implemented by a bundle, but also there is the possibility to have two bundles that define a service in which a bundle specifies the service definition, while another manages the service implementation. The reason for this split is to provide the service definition to a third party without giving it the full implementation that, following the OSGi philosophy, is not required to develop an application. The binary version of bundle(s) and the service API are the only elements required for application development. Furthermore, in some cases, some services require multiple implementations. For example, the Kura Modem Service, the service that provides abstraction to the hardware of modems, is implemented by a set of bundles, one for each supported modem.

Kura Framework provides the following services:

- I/O Services
 - Serial port access through javax.comm 2.0 API or OSGi I/O connection.
 - USB access and events through javax.usb, HID API, custom extensions.
 - Bluetooth access through javax.bluetooth or OSGi I/O connection.
 - Position Service for GPS information from a NMEA stream.
 - Clock Service for the synchronization of the system clock.
 - Kura API for GPIO/PWM/I2C/SPI access.
- Data Services
 - Store and forward functionality for the telemetry data collected by the gateway and published to remote servers.
 - Policy-driven publishing system, which abstracts the application developer from the complexity of the network layer and the publishing protocol used. Eclipse Paho and its MQTT client provides the default messaging library used.
- Cloud Services
 - Easy to use API layer for IoT application to communicate with a remote server. In addition to simple publish/subscribe, the Cloud Service API simplifies the implementation of more complex interaction flows like request/response or remote resource management. Allow for a single connection to a remote server to be shared across more than one application in the gateway providing the necessary topic partitioning.
- Configuration Service
 - Leverage the OSGi specifications ConfigurationAdmin and MetaType to provide a snapshot service to import/export the configuration of all registered services in the container.

- Remote Management
 - Allow for remote management of the IoT applications installed in Kura including their deployment, upgrade and configuration management. The Remote Management service relies on the Configuration Service and the Cloud Service.
- Networking
 - Provide API for introspects and configure the network interfaces available in the gateway like Ethernet, Wifi, and Cellular modems.
- Watchdog Service
 - Register critical components to the Watchdog Service, which will force a system reset through the hardware watchdog when a problem is detected.
- Web administration interface
 - Offer a web-based management console running within the Kura container to manage the gateway.

The next figure illustrates the full structure of the last version of Kura (ver. 2.0.1).

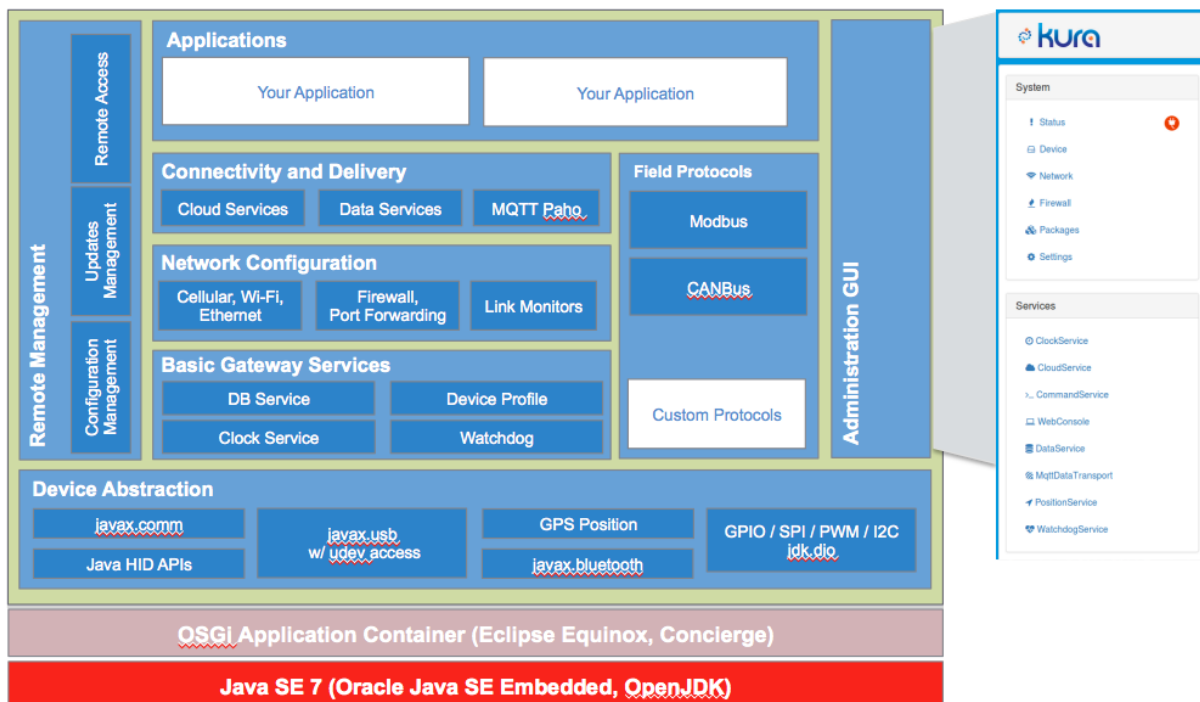


Figure 40: architecture of the Kura framework.

The description of the API is available at the following address:

<http://eclipse.github.io/kura/ref/api-ref.html>

The monitoring architecture of the Kura framework is based on these services and, more specifically, on six functional components that are currently mapped to six different Java packages. Each package contains a set of bundles that implement the functionalities offered by

that architectural component, providing the functionalities as a set of services. The core components in order of abstraction and hierarchical level are (see Figure 41):

- **org.eclipse.kura.net:** this package contains networking related bundles. The APIs include mechanism for setting routes, setting up firewalls, DHCP, and DNS proxy servers. The OS, in general, provides these functionalities natively, thus the services provided by this package are wrappers around those OS services.
- **org.eclipse.kura.system:** the package org.eclipse.kura.system contains services to get system information and perform basic system tasks. It includes services to provide basic system information like operating system information, JVM information and file system information.
- **org.eclipse.kura.db:** this package contains a bundle that provides APIs to acquire and use a JDBC Connection to the embedded SQL database running in the framework. This database represents the local storage for the Kura framework and runs on the remote device.
- **org.eclipse.kura.data:** this package contains a set of bundles that provide services for connecting and communicating with a MQTT broker. The DataTransportService bundles are available in different implementations and provide the ability to connect to a remote broker, publish messages, subscribe to topics, receive messages on the subscribed topics, and disconnect from the remote message broker. The DataService bundle offers methods and configuration options to manage the connection to the remote server. For example, it can be configured to auto-connect to the remote server on start-up or it offers methods for applications to directly manage the connection. It, also, adds the capability to store published messages in a persistent store and send them over the wire at a later time. The purpose is to relieve service users from implementing their own persistent store. In order to overcome the potential latencies introduced by buffering messages, the DataService allows to assign a priority level to each published message. Dependently on the store configuration there are certain guarantees that stored messages are not lost due to sudden crashes or power outages.
- **org.eclipse.kura.cloud:** this package provides services for managing communications between M2M applications and the remote cloud. The CloudService bundle provides an easy to use API layer for M2M applications that need to communicate with the cloud. It operates as a decorator for the DataService providing add-on features over the management of the transport layer. In addition to simple publish/subscribe, the Cloud Service API simplifies the implementation of more complex interaction flows like request/response or remote resource management. Cloud Service abstracts the developers from the complexity of the transport protocol and of the payload format used in the communication. The CloudService allows a single connection to a remote cloud server to be shared across more than one application running on the gateway and providing the necessary topic partitioning. This bundle is responsible to:
 - add application topic prefixes to give to applications running on Kura the capability to share a single remote server connection;
 - define a payload data model and provide default encoding/decoding serializers;
 - publish life-cycle messages when device and applications start and stop.

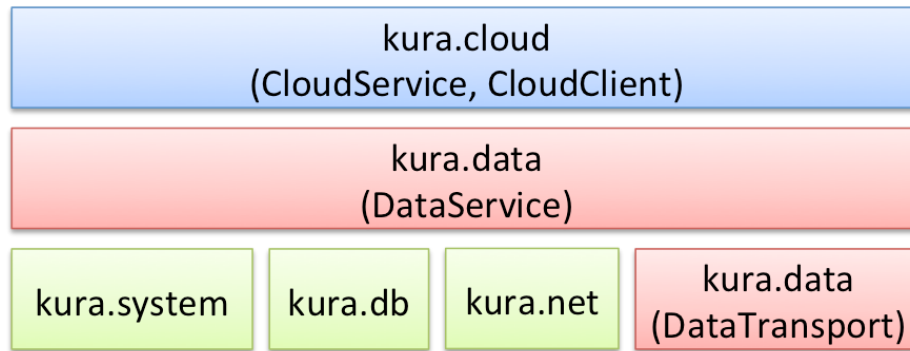


Figure 41: monitoring architecture of the Kura framework.

The CloudService can be used through the CloudClient API.

The CloudClient bundle is designed to be used by single application bundles. CloudClient instances are acquired from the CloudService and they are released when the work is completed. CloudClient leverages the DataService for all the interactions with the transport layer and the communication with the remote server. CloudClient establishes a set of default subscriptions that allow remote cloud servers or other devices to direct messages to the application instance. There can be more than one instance of CloudClient in the system, ideally one per ApplicationId but this is not mandatory.

The CloudClient publishes and receives messages using a topic namespace following a structure as: [CTRL_PREFIX/]accountName/deviceId/appId/appTopic:

- CTRL_PREFIX: is an optional prefix to denote topic used for control messages as opposed to data messages. The framework makes use of control topics to separate management messages like replies from those used for application data.
- accountName: a unique identifier that represents a group of devices and users.
- deviceId: a unique identifier within an account that represents a single gateway device. By default, the MAC address of its primary network interface is generally used as the deviceId of the gateway. In the case of a MQTT transport, for example, deviceId maps to the Client Identifier (Client ID).
- appId: an identifier to denote an application running on the gateway device. We suggest to version the application identifier in order to allow multiple versions of the application to coexist, e.g. CONF-V1, CONF-V2, etc.
- appTopic topic defined and managed by the application.

3.7.2 The telemetry protocol

The monitoring of functional and extra functional properties from the cloud requires an efficient, reliable, light and M2M oriented transport protocol. Considering the application context, the attention has been focused on telemetry-oriented protocols, which must be specifically conceived for embedded system, bandwidth management and communication costs optimization. The automotive use case (UC2) is characterized by these requirements.

A protocol that satisfies the requirements of this application area and that is widely accepted in the industrial world is the MQ Telemetry Transport (MQTT) protocol. The cloud platform and the Kura framework adopt MQTT V 3.1 (<http://mqtt.org/>) as the default transport protocol for device connectivity. MQTT is a lightweight broker-based publish/subscribe messaging protocol designed to be open, simple, lightweight and easy to implement. MQTT is a protocol specifically designed for M2M applications and is currently in the process of standardization by the OASIS standards body under the supervision of Eurotech and IBM. Its main benefits include:

- optimization for M2M applications: only 2 bytes of overhead per packet and integrated management of "quality of service". Through a session-oriented connection to the broker, the communication latency is only limited by the available bandwidth.
- Firewall friendly: the installation of devices within corporate intranets doesn't require opening additional incoming network ports since the connection is initiated by the device.
- Publish/Subscribe messaging: the message pattern to provide one-to-many message distribution and decoupling of message producers (Devices) from consumers (Applications).
- Session awareness: the system automatically generates events when a device disconnects abnormally and provides the ability to fully re-establish the session upon reconnection.
- Security: the connection is protected by SSL and authenticated with username and password.
- Data Agnostic: the independence from data enables transmission of any type and any content of data, in any user-defined form.
- Open standard protocol, based on TCP/IP.
- Quality of Service (QoS) levels determines if and how messages will be delivered.
- Built-in mechanism for connection monitoring.

MQTT is based upon the concepts of "topic" and "payload":

- the "topic" is the destination to which the data is published. It has a hierarchical nature. The topic namespace can be designed to capture, in a tree-like structure, the target destination of the data being published or subscribed. The MQTT topics are mapped on cloud topics (see D3.2.1 "Initial definition of execution platform and cloud services models").
- The "payload" consists of the value/s of the variable/s that has to be transmitted.

As a simple example of an automotive monitoring application, a "topic" can be represented as follows:

acme/123456/AM456KV/acceleration/xaxis

acme/123456/AM456KV/acceleration/yaxis

acme/123456/AM456KV/acceleration/zaxis

This representation applies to Kura and MQTT. On the cloud side, in the topic naming convention, the first two nodes of the topic namespace represent the cloud account name (acme), the identifier (123456) of the device that published the data and the car (AM456KV) in which the device is installed. The nodes of the topic following the namespace are referred to as the “semantic namespace”. This is user-defined and depends on the application: in this case the acceleration data of the car (acceleration/zaxis). The semantic topic is a user-defined hierarchy of sub-topics, and should be carefully designed with the goal of allowing appropriate sets of data to be published or subscribed according to the needs of the application.

In MQTT, the payload is open and can be simply represented by the numeric value of the related variable. The cloud platform uses an additional open payload structure that is further more optimized for telemetry oriented applications. This structure allows third party devices that adopt MQTT to be connected to the platform and to use the Metrics for rules-based statistical analysis. The cloud platform data model defines a message payload composed by:

- timestamp,
- metrics,
- position,
- body.

The “timestamp” is the time when the data was collected and sent to the cloud platform. A “metric” is a data structure composed of the name of the variable, the value of the variable, and the type of the variable (each payload can have zero or more metrics):

(Variable_name, Variable_value, Variable_type)

The following example represents a temperature (floating value) called temp1 with a value of 43.31:

temp1, 43.31, float

or a digital input called din with a value of zero:

din, 0, bool

The allowed variable types for “metrics” are:

- *double* – Double precision (64-bit) IEEE 754,
- *float* – Floating point (32-bit) IEEE 754,
- *int64* – 64-bit integer value,
- *int32* – 32-bit integer value,
- *bool* – “true” or “false”,

- *string* – ASCII string of text,
- *byte array* – Set of hexadecimal byte values.

The “position” data structure is a predefined data model for geo location applications, optimizing the amount of data sent. The “position” part of the payload consists of the following fields:

Table 6 Fields of the “position” part of the payload.

Field	Description	Type	Required?
latitude	latitude of the asset in degrees	double	mandatory
longitude	longitude of the asset in degrees	double	mandatory
altitude	altitude of the asset in meters	double	mandatory
precision	dilution of the precision (DOP) of the current GPS fix	double	optional
heading	heading (direction) of the asset in degrees	double	optional
speed	speed of the asset in meter/sec	double	optional
timestamp	time of day extracted from the GPS system	long	optional
satellites	number of satellites seen by the system	integer	optional
status	status of GPS system: 1 = no GPS response 2 = error in response 4 = valid	integer	optional

Finally, “body” is a non-predefined part of the payload that allows additional information to be transmitted in any format determined by the user.

MQTT is a command-based protocol and the commands mainly used by the Kura framework and by the cloud platform are (for the full list visit <http://mqtt.org/>):

- CONNECT to the data broker – the CONNECT message contains a header identifying the client ID, and other properties of the connection, such as whether to use a Keep Alive timer, and whether to use a Last Will and Testament.
- DISCONNECT from the data broker – normally a client should issue a DISCONNECT message to the broker to close the session
- PUBLISH messages – data is published to the broker using the topic namespace.
- SUBSCRIBE – a device or system can subscribe to one or more topics, including wildcards, in order to receive messages published on those topics.
- QOS – all messages are published with a Quality of Service value:
 - QOS=0: "At most once", where messages are delivered according to the best efforts of the underlying TCP/IP network. Message loss or duplication can

occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost, as the next one will be published soon after.

- QOS=1: "At least once", where messages are assured to arrive, but duplicates may occur.
- QOS=2: "Exactly once", where the message is assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
- Keep Alive – the client can request a certain Keep Alive interval. The broker issues a ping request (PINGREQ), and the client responds (PINGRESP).
- Last Will and Testament – the CONNECT message can include an optional LWT. This is a message that is published to the other subscribers when a client unexpectedly disconnects without sending a DISCONNECT message.
- Retain – all messages may be published with an optional Retain flag, indicating that the server should retain the last published message (one retained message per topic). This allows another client to connect and subscribe later and still receive last known data on certain topics.

Finally, the message life cycle for a session of the cloud client is organized as follows (on Kura side):

1. Connect to broker with MQTT (CONNECT).
2. Send Birth Certificate (PUBLISH with "BirthPayload"; if omitted, device will not be shown as active in the cloud platform console).
3. Issue subscription for topics (SUBSCRIBE; optional).
4. Publish data (PUBLISH; optional).
5. Send Disconnect Certificate (PUBLISH with "DisconnectPayload"; if omitted, device will not be shown as cleanly disconnected from the cloud platform).
6. Disconnect from broker (DISCONNECT; if omitted, MQTT Keep-Alive will be used to detect the missing client, and the Last Will and Testament will be sent to notify other subscribers of the broken connection).

Kura framework and the cloud platform support the last version of MQTT (ver. 3.1.1). This version is a maintenance release with the goal to clarify ambiguities and be as much backwards compatible as possible.

In particular, the following improvements have been added:

- session present flag: an additional flag that indicate that the broker already has prior client's session information like subscriptions, queued messages and other information;
- additional error code on failed subscriptions: prior to MQTT 3.1.1, it was impossible for clients to find out if a subscription wasn't approved by the MQTT broker;
- anonymous MQTT clients: it's now possible to set the MQTT client identifier to zero-byte length. The MQTT broker will assign a random client identifier to the client;

- immediate publishes or bursts of MQTT messages without waiting for responses: is the ability to send MQTT PUBLISH messages before waiting for a response from the MQTT broker;
- longer Client IDs: MQTT 3.1 had a limit of 23 bytes per Client ID. This limit is now set by MQTT 3.1.1 to 65535 bytes.

In the context of the automotive use case (UC2), MQTT is used for the communication between the cloud and the devices installed in the vehicles. The Minigateway manages the communication between the iNemo, the SecSoc and the cloud: Kura runs on the Minigateway and specific bundles are responsible to communicate with the iNemo and with the SecSoc, in order to receive information and publish it to the cloud. The information collected from the iNemo/SecSoc and the commands used to remotely control them are coded using MQTT and packaged into Vodafone Automotive field protocol for the last step of communication between the Minigateway and the iNemo/SecSoc themselves.

3.7.3 Connecting to the cloud

The second part of the cloud-based monitoring infrastructure is the cloud platform itself. Using the CloudService and CloudClient API of Kura framework it is possible to connect a Kura-enabled device to an MQTT broker-url, which handles receiving published messages and sending them to clients who have subscribed. The first step requires the creation of an account on the cloud platform. The account is the reference starting point for all the activities performed in the cloud and for all the data stored at cloud level.

The screenshot shows the Kura MQTT Data Transport configuration page. The sidebar on the left has a 'System' section with links for Status, Device, Packages, and Settings. Below it is a 'Services' section with links for ClockService, CloudService, CommandService, WebConsole, DataService, MqttDataTransport (which is highlighted), and WatchdogService. The main content area is titled 'MqttDataTransport' and contains several configuration fields:

- broker-url***: A text field with a placeholder URL 'mqtt://broker-sandbox.everyware-cl...' and a value 'mqtt://iot.eclipse.org:1883/'.
- topic.context.account-name**: A text field with a placeholder 'account-name' and a description: 'The value of this attribute will replace the '#account-name' token found in publishing topic and must match the name of the EDC account.'
- username**: A text field with a placeholder 'username' and a description: 'Username to be used when connecting to the MQTT broker.'
- password**: A text field with a placeholder 'password' and a description: 'Password to be used when connecting to the MQTT broker.'
- client-id**: A text field with a placeholder 'client-id'.

Figure 42: Kura MQTT Data Transport configuration page.

Before connecting a Kura CloudClient it is necessary to configure the MQTT broker that will answer to the remote publish and subscribe requests. The Kura framework will provide a web interface that allows the user to define the broker configuration. The MQTTTransportService of the cloud platform is responsible to manage the MQTT connection on the cloud side and provides the following parameters: broker url, account name, account credentials, keep-alive period, interaction timeout and MQTT LWT information.

The second bundle that needs to be configured is the DataService. The parameters required for the configuration include: auto connection, connection retry interval, store information (dimensions, data purge interval, etc.) and in-flight parameters required to manage traffic congestions.

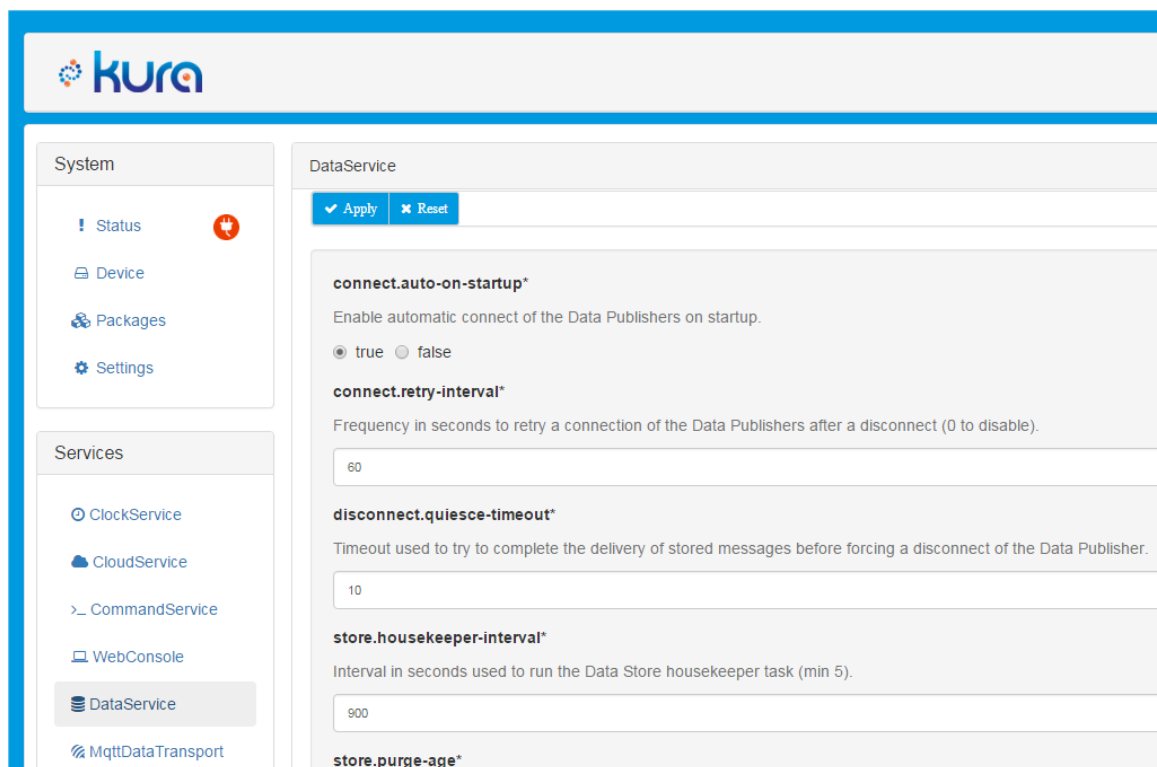


Figure 43: Kura Data Service configuration page.

After this steps a Kura enabled device can connect to the cloud, but some further prerequisites are needed: from the hardware/software point of view, the developer must implement the device-side application logic that, using Kura framework, collects the data from the field and publishes it to the cloud. This requires the development of one or more bundles that implement the application logic, using the Kura hardware abstraction on one side and the Kura Cloud Client on the other.

Finally, for every functional and extra functional property it is necessary to define the related MQTT topics and the corresponding cloud topics.

In the specific context of the automotive scenario, depending on the architecture that will be finally selected, the application will run on the Vodafone Automotive ECU, collecting data from the sensors and publishing directly to the cloud, or on the EUTH Minigateway, that will

collect data from the Cobra ECU and publish them in the cloud. An example of required topics, could be the car acceleration.

3.7.4 The monitoring methodology

Summarizing, the cloud-based monitoring methodology is organized in the following steps:

1. definition of all the required topics, both on the Kura framework and on the cloud platform;
2. development of a set of bundle that implement the use case specific application logic;
3. preliminary configuration of the MQTT Transport Service and of the Data Service, in order to connect to the MQTT broker on the cloud;
4. execution of the Kura framework.
5. The use case specific application:
 - a. collects the data from the field using the hardware abstraction provided by the Kura framework;
 - b. connects to the cloud using the MQTTTransport service and the CloudClient service;
 - c. publishes the information collected from the field.

On the cloud side, the MQTT Broker receives the publishing request, activate the DataPublishing service (see D3.2.3) related to the correct topics and the topics are published.

4 Conclusions

This deliverable described the activities of Task 3.4 concerning the **implementation** of the of extra-functional properties monitoring architectures described in D3.4.1. They have been mapped to the specific property estimation flows described in D3.1.1 and were carried out along the preliminary guidelines described in D3.4.2.

References

- [1] O. Erdinc, B. Vural and M. Uzunoglu, "A dynamic lithium-ion battery model considering the effects of temperature and capacity fading," Clean Electrical Power, 2009 International Conference on, Capri, 2009, pp. 383-386.
- [2] J. Leuchter and P. Bauer, "Capacity of power-batteries versus temperature," Power Electronics and Applications (EPE'15 ECCE-Europe), 2015 17th European Conference on, Geneva, 2015, pp. 1-8.
- [3] W. Liu, A. C. A., A. Macii, et al. Layout-driven post-placement techniques for temperature reduction and thermal gradient minimization. IEEE TCAD, 32(3):406–418, 2013.
- [4] K. Skadron, M. Stan, W. Huang, et al. Temperature-aware computersystems: Opportunitiesandchallenges. IEEE Micro, 23(6):52–61, 2003.
- [5] K. Skadron, M. R. Stan, B. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture: Extended discussionandresults. Technical Report CS-2003-08, Univ. of Virginia Dept. of CS, 2003.
- [6] Yukai Chen, Sara Vinco, Enrico Macii, and Massimo Poncino. 2016. Fast Thermal Simulation usingSystemC-AMS. In Proceedings of the 26th edition on Great Lakes Symposium on VLSI (GLSVLSI '16). ACM, New York, NY, USA, 427-432.
- [7] S. Rhoads. Plasma CPU Core, 2001. opencores.org.
- [8] R. Goergen, "Extra-functional property models" Deliverable D3.1.2.30/09/2015.
- [9] V. Zaccaria, G. Palermo and C. Silvano. "Specification of the XML interface between design tools and use cases. R1.4". Available in Multicube website. <http://www.multicube.eu/>.
- [10] COMPLEX project. <http://complex.offis.de>. Last visited, Sept 27, 2016.
- [11] Y.K. Tan and J.C. Mao and K.J. Tseng, Modelling of battery temperature effect on electrical characteristics of Li-ion battery in hybrid electric vehicle, In Proc. Of IEEE PEDS, pp.637,642, 2011.
- [12] L. Serrao and Z. Chehab and Y. Guezennet and G. Rizzoni, An aging model of Ni-MH batteries for hybrid electric vehicles, Proc. Of IEEE VPP, pp.8, 2005.
- [13] IEEE Standard 1685-2009 for IP-XACT, Standard Structure for Packaging, Integrating, and Reusing IP within Tools Flows, Accellera, 2010.
- [14] Recommended Vendor Extensions to IEEE 1685-2009 (IP-XACT), Accellera, 2013, <http://www.accellera.org>.
- [15] Accellera Systems Initiative, SystemC-AMS and Design of Embedded Mixed-Signal Systems, www.systemc-ams.org.