Advanced SystemC Tracing and Analysis Framework for Extra-Functional Properties

Philipp A. Hartmann[1], Kim Grüttner[1(✉)], and Wolfgang Nebel[2]

[1] OFFIS – Institute for Information Technology, Oldenburg, Germany
pah@computer.org, gruettner@offis.de
[2] Department for Computer Science, University of Oldenburg, Oldenburg, Germany
nebel@informatik.uni-oldenburg.de

Abstract. System-level simulations are an important part in the design flow for today's complex systems-on-a-chip. Trade-off analysis during architectural exploration as well as run-time reconfiguration of applications and their mapping require detailed introspection of the dynamic effects on the target platform. Additionally, extra-functional properties like power consumption and performance characteristics are important metrics to assess the quality of a design. In this paper, we present an advanced framework for instrumentation, pre-processing and recording of functional and extra-functional properties in SystemC-based virtual prototyping simulations.The framework is based on a hierarchy of so-called *timed value streams*, allowing to address the requirements for highly configurable, dynamic architectures while allowing tailored introspection of the required system characteristics under analysis.

1 Introduction

One of the main challenges for extra-functional property monitoring in today's complex embedded systems is the correct attribution of platform activity (computation, communication and e.g. the resulting power dissipation) to the currently active (software) applications running on the platform. Especially in dynamic reconfigurable scenarios, where multiple applications share the same processing elements, interconnects, memories, peripherals and even energy sources over time, the implicit, parasitic interference along extra-functional dimensions can be hard to quantify and consequently to control.

In order to integrate such a monitoring infrastructure in a virtual platform simulation enhanced with extra-functional properties, several requirements have to be fulfilled, both for the functional models used in the simulation, as well as for the recording and pre-processing capabilities provided by the simulation environment. Last but not least, as some use cases require an online feedback (e.g. power management, power-aware scheduling, etc.), the modelling of extra-functional sensors and probes needs to be integrated in the same simulation environment as well.

Fig. 1 shows a typical virtual platform simulation model. Here, the extra-functional activity is ultimately consumed at the architecture level within processing elements, interconnects, memories and peripherals. For multi-application

K. Sano et al. (Eds.): ARC 2015, LNCS 9040, pp. 141–152, 2015.
DOI: 10.1007/978-3-319-16214-0_12

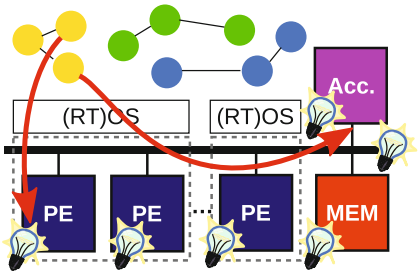

Fig. 1. Application-aware platform activity monitoring

scenarios, the correct attribution to the triggering application tasks needs to be ensured. Moreover, the hardware and software architecture consists of multiple service layers, including operating and run-time layers and potentially further platform support for dynamic reconfiguration. Starting from the activity recording at the SoC component level, the requirements for an application-aware simulation infrastructure are summarized in the following.

- **Simplicity:** Only use those (dynamic) parameters needed for the current use case (e.g. ignore area, when not looking for thermal behaviour).
- **Composability:** Derive combined values from physical relations between individual contributors (e.g. total power, temperature-dependent power, capacitance-based power).
- **Hierarchy:** Put parameters on "correct" geometric level, inheriting parameters from the context and/or the environment.
- **Adaptivity:** Allow changing parameters during run-time to support the modelling of dynamic (e.g. power management) subsystems.
- **Abstraction:** Allow flexible selection and structural/temporal abstraction according to the current analysis and monitoring requirements.

From the simulation infrastructure perspective, this requires a flexible and composable framework to describe customized, use case specific, extra-functional models. As the extra-functional simulation infrastructure has to deal with physical quantities (like capacitance, temperatures, etc.), a shortcoming of many C/C++-based environments is the lack of a proper checking for correctly combined quantities. To avoid such modelling errors, strongly-typed **physical units support** is needed, where composition errors can be caught by the compiler.

Today's system-level simulations oftentimes use sophisticated mechanisms to improve the simulation speed. In virtual platform simulations, *temporal decoupling* is a widespread technique to achieve the required simulation performance to run complex software stacks on the simulated platform. Another approach includes *co-simulation* with models described in other simulators (e.g. RTL, custom instruction-set simulators, Matlab/Simulink) or the integration of emulators or hardware/prototypes in the loop. Even third-party extra-functional models might need to be integrated into a single system simulation. Both of these aspects require the separation of the **functional and the extra-functional simulation time progression**. This means that functional blocks need to be able

to inject their property updates independently from a global simulation time, preferably in a distributed manner. Hence, a separate update and synchronisation mechanism is required for the extra-functional model.

In this paper, we present a highly flexible instrumentation, tracing, and analysis infrastructure for SystemC-based virtual platform simulations. This framework can be used to record arbitrary (physical) values and quantities over time, based on so-called *timed value streams*. These streams can be combined to a run-time (pre-)processing hierarchy before recording the required data for offline analysis. The paper is organised as follows: In Section 2, we give an overview of the state-of-the-art and refine the goals behind this work. The details of the *timed value streams* semantics are given in Section 3 and the recording and instrumentation of the functional models is described in Section 4. The stream processing and analysis capabilities are then presented in Section 5, and Section 6 concludes the paper with an outline of future work based on the presented infrastructure.

2 Goals of this Framework beyond the State-of-the-Art

The main motivation for a dedicated tracing and analysis framework explicitly targeting extra-functional modelling in SystemC is the lack of physical quantity support in the existing, standardized SystemC tracing facilities. Additionally, based on the requirements given in Section 1, more flexibility for the instrumentation, tracing and analysis is needed.

- `sc_core::sc_trace` [5] is not flexible enough, e.g. by being tied to the global simulation time and lacking pre-processing and filtering capabilities.
- `sca_core::sca_trace` [9] is SystemC AMS-specific, not widely supported in commercial environments, and not compatible with temporal decoupling.
- SCV transaction recording [10] not appropriate for physical quantities.
- Some advanced instrumentation APIs are partly available in commercial tools [11], but usually not flexible enough for online preprocessing.

Explicit support for dimensional analysis based on the Boost.Units library [1] has been added to SystemC-AMS in [7]. This approach addresses the lack of proper composition of physical quantities in (SystemC AMS) models. It does not address their tracing, recording, and preprocessing explicitly, though.

Extended tracing frameworks for SystemC have been proposed in the research literature already [3,6]. These approaches focus on transaction-level modelling and do not address the need for the separation of the functional model and the run-time preprocessing of platform activity information. The DUST framework [6] is designed for transaction-level introspection and analysis with advanced storage and online debugging capabilities. The (transaction) recording itself is based on the SCV API [10] and not suitable for extra-functional properties. In [3], "CULT: A Unified Framework for Tracing and Logging C-based Designs" has been proposed. The main features include support for custom back-

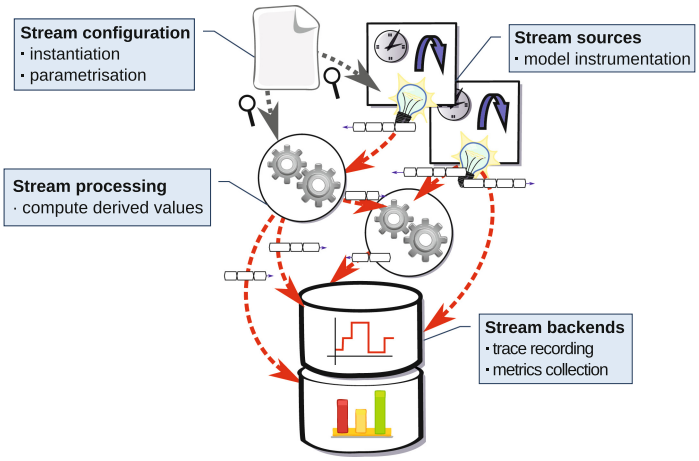

Fig. 2. Composable stream processing framework for activity extraction, preprocessing, monitoring and recording

ends and an minimal-invasive instrumentation of the functional models. Explicit support for physical quantities and their online processing is not addressed.

To cover all of the requirements and to provide the required flexibility and composability, we present an instrumentation framework based on *timed value streams*. The basic idea is shown in Fig. 2. The source streams contents are produced by (functional) components, recording the relevant activity information according to their distributed time model (to support temporal decoupling). These primary traces are then processed by a set of stream processors to derive the physical quantities based on the extra-functional property model.

3 Timed Value Streams

The underlying core technique of the extra-functional monitoring framework presented to the user is based on so-called *timed value streams*, consisting of a sequence of `(value,duration)` tuples.

The basic timed value stream infrastructure is shown in Fig. 3: The leaf annotations in the functional model (1) are `push`ed as tuples according to the current local simulation time of the producing process. These incoming tuples are buffered within the stream (2) without advancing the local time of the stream itself. Once the stream writer explicitly `commit`s its updates, the local simulation time of the stream is advanced and the pending tuples are sent to the listening readers (3). Each reader receives its own copy of the tuples, in order to allow independent consumption of the tuples in the following *stream processors* (Section 5).

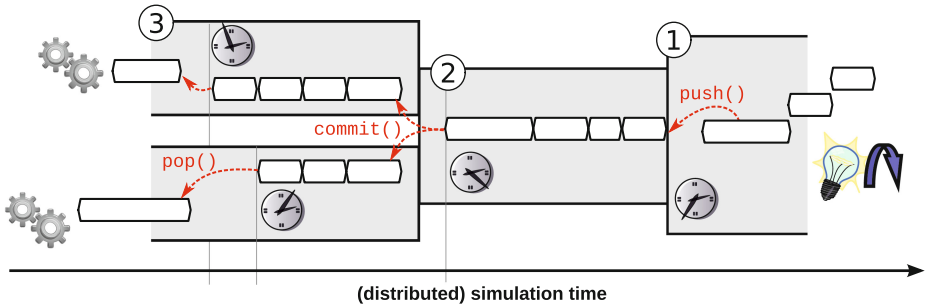

Fig. 3. Basic *timed value stream* infrastructure

Definition 1 (timed value stream).
A *timed value stream* $\mathcal{S} = \langle (v_0, \tau_0), \ldots, \rangle$ is a sequence of tuples (v_i, τ_i) with $v_i \in V(\mathcal{S})$ and $\tau_i \in \mathcal{T} \subseteq \mathbb{R}_{>0}$, where

- $V(\mathcal{S})$ is the value domain of the stream, optionally with a *physical dimension*,
- τ_i denote the *durations* of each tuple, taken from the *time domain* $\mathcal{T}$.

A *stream window* $\mathcal{S}_t^n$ is a finite subsequence of a stream $\mathcal{S}$ of length n, where

- t denotes the starting time, with $\exists j : t = \sum_{i=0}^{j} \tau_i$,
- and n tuples in the stream $\langle (v_j, \tau_j), \ldots, (v_{j+n}, \tau_{j+n}) \rangle$.[1] ♣

During simulation, the (infinite) stream stays a theoretical concept and the following discussion uses finite stream windows instead. Together with the distributed time model, this leads to the phased approach shown in Fig. 3, where individual streams can advance separately and independently from the SystemC simulation time.

It is important to note, that according to above's definition each stream has a strictly monotonic time advance and consists of gapless windows. On the other hand, the activity recording can lead to empty intervals. Furthermore, during the stream processing, streams with unaligned tuples need to be normalized (i.e. split into aligned tuples) to define the time resolution (tuple durations) of the output streams. Both aspects need to be addressed properly in the context of physical quantities.

3.1 Support for Extra-Functional/Physical Quantities

Especially in light of extra-functional properties, the values held by a *timed value stream* need to be classified according to their semantics. While functional tracing usually observes the *state* of a system over time, extra-functional properties emerge both as *state quantities* and *process quantities*.[2]

[1] We sometimes omit j for improved readability.

[2] This classification is loosely adopted from the theory of thermodynamics. Sometimes, these classes are called state and process *functions*.

State Quantities describe the state of a system at a given time. Without external influences, this state is kept. In the context of system-level modeling, this can include extra-functional properties like the cache hit/miss rates, power consumption, (ambient) temperature and of course the functional state. For timed streams, tuples 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 loosing information.

Process Quantities describe a *state change* with an associated duration. Examples include the amount of cache hits/misses,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.

Timed Stream Traits are used in the implementation to address the need for the above distinction. A special template parameter defining the different stream tuple policies can be given. The different policies provide an implementation for the different stream (tuple) operations, like splitting, joining and merging tuples.

An example for the default traits of the more interesting *process quantity traits* is given in Listing 1.1. For state quantities corresponding traits classes are available as well. The `merge_policy` is needed for conflicting `push`es to a stream (see Section 4). If the default traits are not sufficient, users can define their own traits (and policies) by inheriting from the default classes and overriding the nested `typedef`s.

```
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 values 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;
}; // timed_process_traits<ValueType>

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

Listing 1.1. Timed stream traits for process quantities

3.2 Distributed Time Model and Synchronisation

In order to support the integration of the stream-processing with a temporally decoupled simulation, the different components integrate into a distributed time model, enabling hierarchical synchronisation between different components.

To finalise the values written to a particular temporarily decoupled timed stream, the pushes need to be explicitly committed. This finalizes the current stream window with all tuples written to the stream until the current local time offset. Subsequent pushes with explicit timestamps earlier than the committed time lead to a run-time error. In addition, attached tracing observers (processors or backends, see Section 5) are notified that new data is available for processing.

In case of multiple independent streams in a single component, the local time offsets need to be consistent as well. Furthermore, the driving SystemC process may need to consume the SystemC simulation time at some point. Both operations are tightly coupled, therefore a second overload of `sync` functions is provided by the timed streams. Having the same semantics for their arguments, these explicit `sync` functions return the offset to the current absolute SystemC simulation and perform a `commit` on *all streams in the current component*. With this, a common idiom to finalise a local computation is the `wait(sync)` call using one of the streams explicitly as shown in Listing 1.2. Alternatively, a convenience macro to mark such a synchronisation point explicitly is available as well.[3]

```
void commit();                                    // commit all pending pushes
void commit( duration_type const & offset );      // ... explicit window
void commit( time_type const & offset );          // ... until offset

time_type sync();                                 // commit & synchronise
time_type sync( duration_type const & offset ); // ... explicit window
time_type sync( time_type const & offset );       // ... until offset

#define SYSX_SYNCHRONISATION_POINT() \
  sc_core::wait( [stream_scope].sync() )
```

Listing 1.2. Stream synchronisation API

4 Activity Recording – Stream Sources

Writing to a *timed stream* can be done explicitly by using a `timed_writer` object attached to a stream (`push` interface), or based on implicit extraction of stream updates from variables within the functional model (annotation interface).

4.1 Explicit Writing to a Timed Stream

Explicitly pushing to a *timed value stream* can be performed via a *stream writer*, attached to the stream. Writers can be attached and removed from a stream dynamically during runtime, either based on the streams name or in terms of a direct C++ reference.

[3] The current "scope" is determined by the tracing framework automatically.

```
void push( value_type const & value, duration_type const & duration ); // (1)
void push( time_type  const & offset
         , value_type const & value, duration_type const & duration ); // (2)
void push( value_type const & value );                                  // (3)
```

Listing 1.3. Stream push interface

Overload (1) adds a given `value` for a given `duration` to the current end of the stream. Since each stream maintains a local time offset, this time offset is automatically advanced by the given duration, when this interface call is used. This interface is particularly well suited for annotating local computations without requiring external synchronisation in-between.

The `push` (2) function can be used to write (future) values to a stream, delayed relatively to the stream's local time as given by the `offset` parameter. When using this overload, the local time of the stream is not advanced. This variant can be used to support out-of-order temporal decoupling and potentially leads to overlapping tuples that need to be merged according to the stream's `merge_policy` (creating an error by default).

The final overload (3) can be used in case of an unknown duration (only suited for *state quantities*). In this case, the value is assumed to be held indefinitely until overwritten again. Again, the stream's local time is not advanced until the next `commit`.

4.2 Block-Based Annotations

A frequent use case of the annotation framework is the augmentation of application source code with extra-functional properties. If only a single stream is driven from within a process, no inter-stream synchronisation is needed and the local time of the component is equal to the local time of the (only) stream. If multiple streams are maintained, each of which has different update characteristics, keeping the local time offsets consistent quickly becomes inconvenient. Therefore, a higher level abstraction for block-based annotations of execution time durations is provided, separating the time annotation from the actual value updates again and handling the synchronisation transparently for the user.

```
timed_stream<process_state>                  state_str;    // IDLE
timed_stream<unsigned, timed_process_traits> mem_load_str; // 0

void faculty(int in0, int& out0) {
   timed_var<process_state> state( state_str );       //traceable var
   timed_var<unsigned>      mem_load( mem_load_str ); //traceable var
   timed_ref<int>           in      ( in0 );          // alias parameter

   SYSX_TIMED_BLOCK( sc_time(500, SC_NS) ) {   // functionality
      state    = BUSY;
      mem_load = 10;
      // ...
   } // automatic push to all streams in scope

   while(tmp0) SYSX_TIMED_BLOCK( sc_time(200, SC_NS) ) {
      state    = BUSY;
```

```
    mem_load = 2;
  } // automatic push to all streams in scope
  SYSX_SYNCHRONISATION_POINT}(); // commit and sync with SystemC
  return;
}
```

Listing 1.4. Simple example of tracing multiple local values

The different basic blocks for annotations are wrapped within `SYSX_TIMED_BLOCK` C++ scopes providing the duration of the block. Instead of explicit pushes to different streams, `timed_var` objects can be used and updated via plain assignments. These timed variables (or their `timed_ref` counterpart, aliasing an existing variable) are used to implicitly push the corresponding tuples upon exit of the annotated block.

5 Run-Time Extra-Functional Property Monitoring – Stream Processing

As sketched in Fig. 2, the actual processing of the leaf instrumentation towards derived extra-functional properties is performed by a hierarchical set of so-called *stream processors*. These processors are triggered based on the object-oriented *Observer pattern*, without relying on the SystemC simulation itself (no SystemC processes, events, channels) to facilitate a separation of the models and to allow dynamic reconfiguration of the analysis environment during runtime. Stream processors can attach to a (set of) `timed_reader`s and subscribe to `commit`s, either for each extension of a pending window or just for the start of new windows.

```
void notify( timed_reader_base & src );
```

Listing 1.5. Stream observer interface

Subsequently reading pending values via a `timed_reader` can be done in several ways. For most pre-processing operations, the `(const_)tuple_iterator` based interface is sufficient. The streams provide the output interface for consuming the committed data sketched in the following listing:

```
value_type const & get() const;                              // read a
value value_type const & get( duration_type const & o ) const;
// ... at given offset

tuple_type const & front() const;                            // read
first tuple tuple_type const & front( duration_type const & d );
// ... for a duration

void pop_front();                                            // drop
first tuple void pop_until( time_type const & until );
// ... for a duration void pop_all();
// ... all tuples

const_tuple_iterator begin() const;       // get iterators to the
pending tuples const_tuple_iterator end()   const;
```

Listing 1.6. Stream reader interface

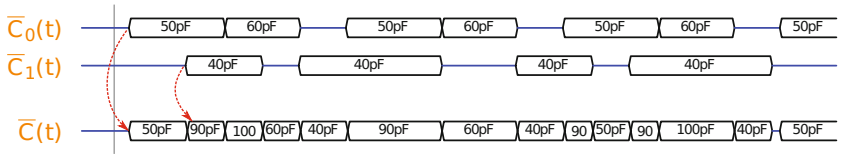

Fig. 4. Structural abstraction: Accumulating switching activity by stream processing

Additionally, various querying functions about the different time offsets and other utility functions are available in the stream base class.

5.1 Time Normalisation of Streams

When combining streams with different time resolutions, a *time normalisation* needs to be performed. This means that tuple boundaries need to be aligned before combining their values according to the stream processor's output function.

For a stream processor listening to multiple streams $\mathcal{S}_0, \ldots, \mathcal{S}_k$, the pending windows[4] $\mathcal{S}_{i,t_0}^{n_i}$ are transformed to *normalised windows* $\mathcal{S}_{i,t_0}^{n_i'}$ by applying the stream's `split_policy` according to the following rules:

$$
\begin{aligned}
\mathcal{S}_{i,t_0}^{n_i'} &= \left\langle \left(v'_{i,0}, \tau'_0\right), \ldots, \left(v'_{i,n'-1}, \tau_{n'-1}\right) \right\rangle \\
&= \left\langle \left(v'_{i,0}, \left(t'_1 - t_0\right)\right), \ldots, \left(v'_{i,n'-1}, \left(t'_{n'-1} - t'_{n'-2}\right)\right) \right\rangle, \text{with} \\
t_{i,j} &= t_0 + \sum_{l<j} \tau_{i,l} \\
t'_j &= \inf_{0 \le i \le k} \left\{ t_{i,l} : t_{i,l} > t'_{j-1} \right\}, t'_0 = t_0 \\
v'_i &= \text{SplitPolicy}(\mathcal{S}_{i,t_0}^{n_i}, \tau'_i)
\end{aligned}
$$

This normalisation can be performed by the stream processor base class to simplify the stream processor implementation itself. An example for this operation is shown in Fig. 4, which depicts a *structural abstraction* by combining two (average) switching activity streams from two functional components into a joint switching activity stream.

Another dimension for stream abstraction is the reduction of the time resolution, called *temporal abstraction*. A stream processor can for instance reduce the number of tuples in a stream by averaging the values over a fixed time window. This is useful to reduce the amount of tuples that need to be processed and helps improving the simulation performance.

5.2 Offline Analysis – Stream Backends

Especially for offline or post-mortem analysis, a special set of stream processors can be used, so called *stream backends*. Compared to a generic stream processor,

[4] The starting time is assumed to be aligned already.

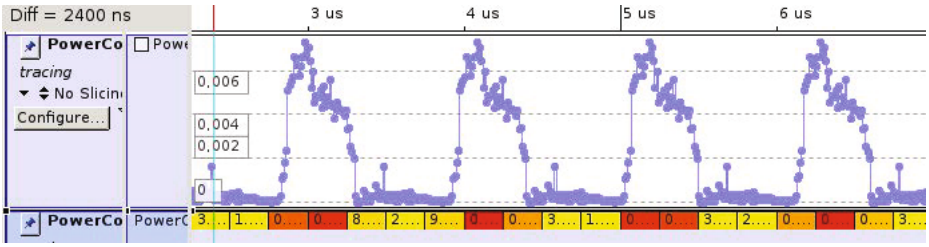

Fig. 5. Example trace integration into Synopsys Virtualizer

a stream backend has no outgoing timed streams on its own and merely processes (a set of) incoming streams for specific purposes. Different use cases for stream backends are to be considered, as described in the following. The detailed external interface of these backends are specific to these use cases and therefore out of the scope of this paper.

Trace File Generation. One of the most obvious backends for a tracing and analysis infrastructure is the generation of value-over-time trace files, e.g. based on the Value Change Dump (VCD) format [4]. This widespread text-based format is supported by most graphical analysis tools and can be generated by the standard SystemC `sc_trace` API as well. Consequently, a VCD backend has been implemented for timed streams.

Secondly, some commercial simulation environments provide additional analysis capabilities beyond a mere visualisation of traces. One example for such an environment is the Synopsys Virtualizer tool suite [11], which includes an generic data analysis API suitable for the integration with the timed value streams extension. An example excerpt of a dynamic power trace stream visualized by the Virtualizer user interface is shown in Fig. 5.

Metrics/Statistics Collection. Another frequent requirement of system-level simulations is the gathering of compact performance or quality metrics for example to drive an automatic design space exploration. To enable the collection of such metrics or statistical information, user-defined stream backends can be used as well. These specific metric backends then report their value(s) at the end of the simulation (or explicitly upon request). As there is no feedback to the simulation model itself, the computation of the design metrics can run independently of the SystemC simulation time.

6 Conclusion

In this paper, we have presented a novel approach for instrumentation, preprocessing and analysis of extra-functional properties in system-level simulations. The SystemC-based implementation uses *timed value stream* to transport such

extra-functional properties through a hierarchy of stream processors towards dedicated backends for offline analysis (trace files or report generators). The approach is currently included in our simulation environment for power-aware virtual prototypes [2].

As a special extension, we will now start to define a dedicated set of stream backends to allow simulation-based validation of extra-functional contracts [8] (assumption/guarantee pairs covering extra-functional properties). The underlying linear-temporal logic properties (LTL) will be used to run-time monitors based on stream processors. With the capability to store a continuous window of tracing history (see Section 5), the stream-based infrastructure is perfectly suited for this kind of temporal monitoring.

Acknowledgments. This work has been partially supported by the EU integrated projects COMPLEX (FP7-247999) and CONTREX (FP7-611146).

References

1. Boost. Units library 1.1.0. http://www.boost.org/doc/html/boost_units.html
2. Grüttner, K., Hartmann, P.A., Hylla, K., Rosinger, S., Nebel, W., Herrera, F., Villar, E., Brandolese, C., Fornaciari, W., Palermo, G., Ykman-Couvreur, C., Quaglia, D., Ferrero, F., Valencia, R.: The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration. Microprocessors and Microsystems 37(8,C), 966–980 (2013), Special Issue on European Projects in Embedded System Design (EPESD 2012)
3. Hong, W., Joshi, J., Vieh, A., Bannow, N., Kramer, A., Post, H., Bringmann, O., Rosenstiel, W.: Advanced features for industry-level logging and tracing of C-based designs. In: Forum on Specification and Design Languages (FDL 2013). IEEE (September 2013)
4. IEEE Standard Verilog Hardware Description Language. IEEE Std. 1364–2005, IEEE Computer Society (April 2006) ISBN 0-7381-4851-2
5. IEEE Standard SystemC Language Reference Manual. IEEE Std. 1666–2011. IEEE Computer Society (January 2012). http://standards.ieee.org/getieee/1666/ ISBN 978-0-7381-6801-2
6. Klingauf, W., Geffken, M.: Design structure analysis and transaction recording in SystemC designs: A minimal-intrusive approach. In: Forum on Specification and Design Languages (FDL 2006). IEEE (September 2006)
7. Maehne, T., Vachoux, A.: Supporting dimensional analysis in SystemC-AMS. In: IEEE Behavioral Modeling and Simulation Workshop (BMAS 2009), pp. 108–113 (September 2009)
8. Nitsche, G., Grüttner, K., Nebel, W.: Power contracts: A formal way towards power-closure?! In: Proc. of the 23rd Intl. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS), pp. 59–66 (September 2013)
9. Standard SystemC AMS extensions 2.0 Language Reference Manual. Accellera Systems Initiative (March 2013). http://accellera.org/downloads/standards/systemc
10. SystemC Verification Library 2.0. Accellera Systems Initiative (July 2014). http://accellera.org/downloads/standards/systemc
11. Synopsys: Virtualizer. http://www.synopsys.com/systems/virtualprototyping