# SystemC$^{SV}$: An Extension of SystemC for Mixed Multi-Level Communication Modeling and Interface-Based System Design

Robert Siegmund, Dietmar Müller

Chemnitz University of Technology
Professorship Circuit and Systems Design
09107 Chemnitz, Germany
E-mail: {rsie,mueller}@infotech.tu-chemnitz.de

## Abstract

*An extension of SystemC for mixed-multi level communication modeling and Interface-based system design is proposed in this paper. SystemC$^{SV}$ provides a new design unit, the* interface*, which enables specification, design and verification of system communication separately from system functionality, thus introducing a new quality of system design into SystemC. The concepts and computational model of SystemC$^{SV}$ interfaces are presented together with a design example, the digital part of a wireless SmartCard transponder-reader/writer system.*

## 1. Introduction

With growing complexity of integrated circuits and the possibility to realize whole electronic systems with all hard- and software components on a single chip, there is a need for new system design methodologies and accompanying languages which allow to capture executable system specifications at a high level of design abstraction early in the design cycle. Among other approaches to a system level design language such as SpecC [3], in the last year SystemC [5] has emerged as a C++ based language for specification of digital systems. With SystemC, all hard- and software components of a digital system can be described using one language which in addition most hard- and software architects are familiar with. However, currently SystemC does not extend the modeling capabilities of VHDL or Verilog HDL but rather resembles them. In particular, the level of design abstraction is not effectively raised in SystemC, compared to common HDL's.

In this paper, we propose an extension of SystemC, called SystemC$^{SV}$ (the SV superscript is a reference to **S**uper**V**ISE, ICL's design methodology that first used the concept of interfaces). With this extension, we want to address two major challenges designers have to face at the system level. First, as the complexity of system modules grows rapidly, they reveal very complex control-dominated interfaces which are connected to system buses with sophisticated data communication protocols. Therefore, beside the design and implementation of system functionality, the proper design of *system communication* is more and more in the focus of the designer. This involves the exploration of the design space for different communication protocols regarding their influence on the overall system performance and implementation cost for the corresponding controller logic. Second, in large and highly complex systems, different parts of the design my be at different levels of design abstraction. This may stem from the situation that the work is split onto a number of designer teams which advance with a different speed. Or, an IP core is used whose simulation model is at a low level of abstraction with a clock cycle accurate interface while the surrounding design has been modeled at a higher level of abstraction. And finally, concerning testbench reuse, testbenches have mostly to be rewritten as the design advances to lower abstraction levels because stimuli become more complex and detailed.

### 1.1. Overview

SystemC$^{SV}$ attempts to provide a solution for the problems previously mentioned by introducing a new design unit to SystemC, called the *interface*. The Interface describes communication between a number of behaviours at different levels of abstraction in terms of *interface items*. An interface item describes communication between behaviours at a specific level of abstraction. Taking the CAN bus protocol as an example, an item may represent a complete CAN bus transaction, a data or address frame, a field within this frame or just a single bit, represented by a signal state on the physical bus wire. Figure 1 visualizes the principle of SystemC$^{SV}$ interfaces. Modules A and B represent two communicating behaviours which are part of a system to be de-

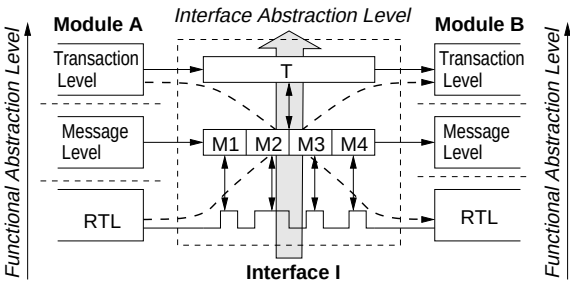


**Figure 1. Mixed multi-level communication modeling in SystemC$^{SV}$**

signed and which themselves may be at different levels of design abstraction. In an early stage of the design these modules are usually represented as asynchronously communicating processes and no communicative details such as data exchange protocols have been fixed. Therefore, communication is modeled with abstract transactions $T$. As the design of the system advances, the module behaviours will be iteratively refined through decomposition into a set of lower level behaviours. In the SystemC$^{SV}$ interface, abstract communication items can be refined accordingly by decomposing them into a set of lower level items ($M1..M4$ in Fig. 1). This enables the design of system communication similar to the design of system functionality. At the lowest level of interface abstraction, items finally represent signal states on physical wires. It is an inherent feature of a SystemC$^{SV}$ interface to represent an interface item *concurrently* at *all* levels of interface abstraction that are described by means of item compositions. This feature enables mixed multi-level communication modeling as depicted in Fig. 1. There is no need for explicit module interface wrappers that perform the necessary translation of communication between abstraction levels, e.g. if a module with a signal level interface wants to communicate with another more abstract module that only understands abstract transactions. The interface itself will perform the necessary translation between interface abstraction levels in all directions.

### 1.2. Related work and motivation

The concept of separating system communication from system functionality is not new. In [4] a design methodology referred to as Interface-based design was proposed that orthogonalizes functionality and communication of a digital system, aiming at the separate, independent exploration of the corresponding design spaces. Among other system level specification languages, SpecC [3] implements this methodology with its capability to model systems at the system level as a set of behaviours which communicate by means of abstract messages bundled in virtual channels. In the course of specification refinement, the designer can describe communication in terms of channel functionality, represented by two distinct behaviours that decompose message parameters into a sequence of physical bus signals and vice versa. Because the design of these control-dominated behaviours may be time-consuming, the designer usually assigns standardized communication protocols to the channels for which the corresponding behaviours are retrieved from a protocol library. VHDL+, an extension of VHDL for system level specification [6], also enables communication modeling in terms of high level message passing between behaviours. In addition the *interface* design unit is provided which lets the designer specify declarative, hierarchically structured protocols for the abstract messages instead of requiring the explicit specification of the corresponding composing and decomposing channel behaviours that map the message to physical bus signal states and vice versa. One of the main advantages of this concept is that communication between system modules which are at different levels of abstraction is established without the need of interface wrapper modules that translate between the levels of interface abstraction.

In the last years, a number of C++ based system design languages have been proposed. They are based on C++ class libraries for modeling of hardware-inherent properties such as timing and concurrency. SystemC is to our knowledge the only language that is purely implemented in C++ and, in contrast to e.g. the Cynapps Hardware Specification Environment [1] does not require a compiler to translate the models into C++ code, meaning that the language can be easily extended by the user by simply adding new C++ classes. In SystemC, high level communication is currently implemented in form of the Remote Procedure Call (RPC) mechanism introduced in version 1.1. The RPC mechanism uses fixed, predefined signaling protocols and does not allow to actually design system communication. Also, for communication across levels of design abstraction explicit wrapper modules have to be implemented that translate between RPC level and signal level.

Currently, there is no feasible language and design environment making the benefits of interface-based system design fully available to the designer. Therefore, SystemC$^{SV}$ was created to combine the advantages of the interface modeling concepts of VHDL+, the high simulation speed and HW/SW-domain unifying language approach of C++ based system modeling and the user extensibility of SystemC.

## 2. Interface modeling

### 2.1. Interface specification

An example for a SystemC$^{SV}$ interface specification is given in Lst 1. An interface is declared with the

SC_INTERFACE keyword and populated with a set of interface item references. The referenced interface items, in our case the message *Frame*, are declared and implemented subsequently. Finally, in the interface constructor the logical ends of the interface between which items are transmitted (*Master,Slave*) are specified and all item references are linked with an allocated item of the corresponding type.

```
SC_INTERFACE(IEC14443) {
  sc_SVMessageRef Frame_M;
};

SC_MESSAGE(Frame) {
  sc_SVParamArray <sc_uint<8>, 8> data;
  sc_SVParamArray <sc_uint<8>, 2> crc;
  sc_SVParam <sc_uint<8> > datalen;

  SC_MESSAGE_CTOR(Frame) {
      FROM << "Master"; TO << "Slave";
      TAKES(16,200);
      PARAMETER(data);PARAMETER(crc);
      PARAMETER(datalen); }
};

SC_INTERFACE_CTOR(IEC14443) {
  BETWEEN << "Master" << "Slave";
  ALLOCATE_ITEM(Frame, Frame_M);
}
```

**Listing 1. SystemC$^{SV}$ interface specification**

## 2.2. Interface Item

The basic element of a SystemC$^{SV}$ interface is the *interface item*. Interface items represent resources used for communication between behaviours at a specific level of abstraction. Listing 2 shows the C++ class model of a SystemC$^{SV}$ interface item. In order to have interface items executed concurrently to other SystemC modules during simulation, they are simply derived from the sc_module class.

```
class sc_SVInterfaceItem : sc_module {

   sc_SVInterfaceEnd FROM, TO, BETWEEN;
   sc_SVTimer TAKES;
   sc_SVParam_base_list PARAMETERS;
   sc_SVCompositionScheme COMPOSITION;

   void generate();   void consume();
   void decompose();  void compose();
   void notify(); };
```

**Listing 2. SystemC$^{SV}$ interface item model**

This ensures that they are scheduled and executed by the SystemC simulation kernel like any other module. A further advantage of this method is that SystemC$^{SV}$ does not make any modification of the code of the SystemC class library but hooks onto the well-defined user interface, which should ensure compatibility with future versions of SystemC.

A SystemC$^{SV}$ interface item has a number of attributes which reflect typical properties of communication. So express the **FROM**, **TO** and **BETWEEN** attributes the fact that communication is directed and specify that direction in terms of interface terminals between which a particular item is transmitted. In Lst. 1, message *Frame* would be transmitted from end *Master* to end *Slave*. Furthermore does the time-based **TAKES** attribute reflect the property that the transmission of an item consumes an amount of time. This time attribute can actually be set to a concrete value in terms of interface clock cycles or to a time span, which introduces controlled nondeterminism into interface modeling. This is especially useful in early stages of design for constraining the design space without actually making a concrete implementation decision. In the example it would take between 16 and 200 cycles to transfer message *Frame*. Finally, the **PARAMETER** attribute describes the information content or payload of an interface item. Message *Frame* has three parameters *data*, *crc* and *datalen* which contain the user data to be transmitted between *Master* and *Slave*. Currently, three different types of interface items, describing different levels of communication abstraction, are defined in SystemC$^{SV}$:

**Transaction item.** The most abstract SystemC$^{SV}$ interface item is the transaction item and is declared with the `SC_TRANSACTION` keyword. It is used for specification of a *multi-directional* information transfer between system modules, e.g. a data transfer which involves a turnaround in direction such as handshaking. A transaction item specification basically looks the same as a message item specification, except that the direction attribute is set separately for each transaction parameter which reflects the multi-directional information flow.

**Message item.** In contrast to the transaction item, the message item represents a *unidirectional* information transfer between system modules and usually describes a specific part of a transaction. The specification of a message item in SystemC$^{SV}$ is shown in Lst. 1.

**PHYMAP item.** This item provides a means for clock-synchronous mapping of ”virtual” transactions and messages with their parameters to sequences of physical signal states, which eventually implement the communication between system modules. Listing 3 shows a code example for a PHYMAP item specification.

```
SC_PHYMAP(MapBit) {
    sc_SVSignalRef<bool> XD;
    sc_SVParam<bool> XD_val;
```

```
    SC_PHYMAP_CTOR(MapBit) {
      FROM << "Master"; TO << "Slave";
      PARAMETER(XD_val);
      ASSOCIATE(XD - XD_val);
    }
};
```

**Listing 3. PHYMAP item specification**

## 2.3. Interface item composition

The item model has a further element, the COMPOSITION. It describes the implementation of an interface item at a lower level of abstraction in a declarative style. An item composition is formed by a set of lower level interface items whose execution is scheduled relatively to each other in a particular scheme. Currently, SystemC$^{SV}$ provides four different composition schemes to the designer (Fig. 2), which can be arbitrarily combined and should allow for modeling of any communication protocol used in practice. Items can be composed serially or in parallel using the SERIAL or PARALLEL composition scheme, respectively, and, depending on the value of an item parameter, items can be repeatedly scheduled (REPEAT scheme) or one item out of a list of alternatives can be selected (SELECT scheme). An example for a SystemC$^{SV}$ item composition is shown in Lst. 4:

```
SC_MESSAGE(Frame) { ...
  SC_MESSAGE_CTOR(Frame) { ...
    COMPOSITION(
          SERIAL(SoF_M(),
                 DATA_M(data,datalen),
                 CRC_M(crc),
                 EoF_M()))
};
```

**Listing 4. interface item composition**

The property of an interface to translate interface item representations between all abstraction levels that have been defined for them implies that item composition schemes must be *reversible* and actually reveal two distinct behaviours. The *decomposing behaviour* describes how the corresponding item is decomposed in time into a number of lower level items. The *composing behaviour* describes the reversed decomposing behaviour, e.g. how a sequence of lower level items are assembled to form a more abstract item. These behaviours are implemented in every item's compose() and decompose() methods, respectively. Due to limited space, the behaviours of the composition schemes are not detailed. It is just pointed out that the demand for

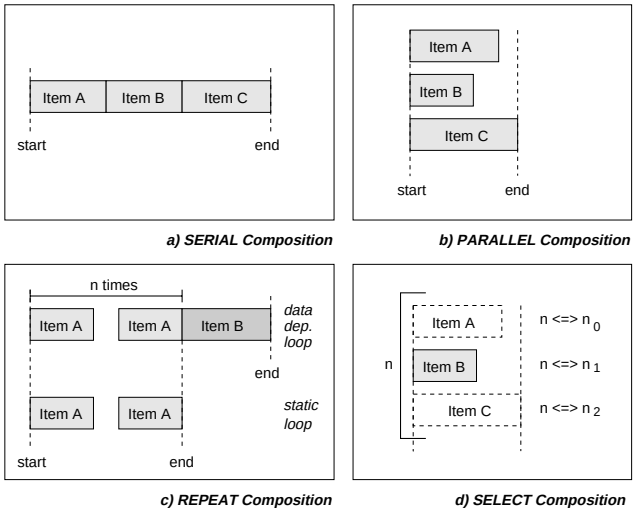


**Figure 2. Item composition schemes in SystemC$^{SV}$**

reversibility has a consequence for parameter driven composition schemes such as the REPEAT and the SELECT scheme: In the decompositional behaviour, parameter values determine the number of repeats or select the item to be executed. However, in the compositional behaviour these parameter values are *established* by the composition, e.g. in the SELECT scheme *n* is set to the value which corresponds to the consumed item. The REPEAT composition scheme has two forms concerning its compositional behaviour: The first form realizes a statically bounded loop, e.g. the number of items *A* to be consumed is fixed by parameter *n*. The second form implements a data dependent loop which is terminated by an item *B* that must be different from item *A*. The parameter *n* will then be set to the number of consumed items *A*.

## 2.4. Interface computational model

As stated before, the interface represents communication concurrently at all levels of abstraction, thus enabling mixed multi-level communication. The computational model for performing the translation between the abstraction levels is shown in Fig. 3. If e.g a message item is sent by a behaviour the following actions are taken by the interface simulation kernel: The item generate() method is invoked, which starts the item life timer that has been initialized before with the value of the TAKES attribute. Concurrently, if the item has a composition, the item decompose() method is invoked with the consequence that a number of lower level items forming the item composition will be generated and which will in turn invoke their decompose() methods. If the item composition has PHYMAP items, the PHYMAP item

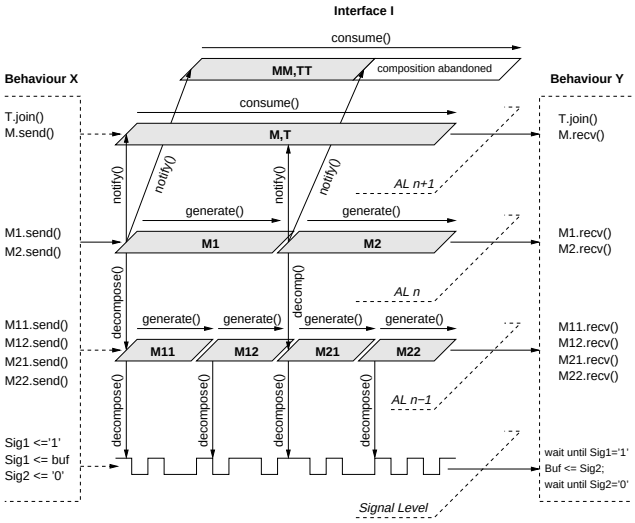


**Figure 3. Computational model for translation between levels of interface abstraction**

decompose() method is executed, mapping the item with its parameters to physical signal states. Through this recursive item decomposition, all lower level representations of the item are generated in the interface. If the item had no composition, the item life timer would determine the duration of the item. In case the item has a composition, the time required to execute it is automatically checked against the expiration of the item alive timer, and an error would be signaled if the composition execution took longer than the time specified in the TAKES attribute.

In order to translate an item into higher level representations, it has to be considered that the item could be referenced in the composition of different higher level items. In order to account for this, when generated each item executes notify() callbacks to these higher levels items. These items will then invoke their consume() method, trying to assemble themselves according to their composition, by composing the lower level items appearing in the interface over time. Because an item composition is unambiguous, in the course of item consumption, the composition of only one particular item will be satisfied and this item will be received by a behaviour listening to the interface. All other item compositions are abandoned. With this mechanism, a SystemC$^{SV}$ interface can translate an abstract transaction or message into a sequence of signal states or, vice versa, a signal sequence into a more abstract interface item, while inherently verifying the conformity of all lower level interface implementations to the timing constraints specified in higher levels of abstraction.

## 3. Behavioural modeling extensions

While conventional SystemC modules are connected via ports by means of signals, in SystemC$^{SV}$ they can also be connected via *interface ports* and *interface signals*, (latter are instances of a particular interface), which enables interface item-based messaging. Interface ports implement the logical ends of an interface and provide methods *send* and *receive* for sending and receiving message items in behaviours, as shown in the code example in Lst. 5. Multi-directional transactions can be initiated or participated in behaviours by means of the *join* method. All these methods implement a *blocking* communication, e.g. their execution will block the enclosing threads.

```
SC_MODULE(IEC14443_Master) {
   sc_SVInterfacePort<IEC14443> IP;
   IP.IMPLEMENTS("Master");
   ...
   IP.send(IP->Frame_M(data,crc,8));
};
```

**Listing 5. Use of interface items in behaviours**

## 4. Design example: Wireless SmartCard transponder-reader/writer system

For evaluation purposes, SystemC$^{SV}$ has been applied to the design of the digital part of a wireless SmartCard transponder-reader/writer system [2] (Fig. 5). The mobile SmartCard contains the transponder whose core part

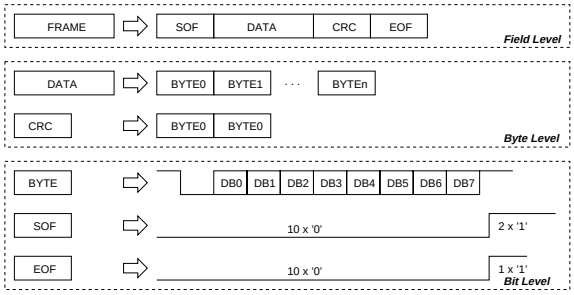


**Figure 4. IEC14443 frame and character transmission format**

is a non-volatile FRAM used for storing data such as electronic signatures. The reader/writer is a stationary terminal whose task is to read and modify the transponder FRAM. Transponder and reader/writer communicate using a serial protocol which conforms to the IEC14443 character and frame transmission format as shown in Fig. 4. In terms of SystemC$^{SV}$ interfaces, communication can be represented

at the abstraction levels *Field*,*Byte* and *Bit*. The most abstract interface item is the IEC14443 Frame consisting of Start/End-of-Frame and checksum fields and a field containing the actual data to be transmitted. This data block contains a SmartCard command for reading and writing to the transponder FRAM. The design task was to create an abstract model of the transponder/reader-writer system with the reader/writer sending commands to the SmartCard as well as receiving the SmartCard replies. For the reader/writer, also a software component for processing of the SmartCard reply data was to be implemented. Finally should the abstract model allow an iterative implementation of the reader/writer and transponder down to RTL level while enabling the verification of them in the functionally proven abstract model.

### 4.1. Design flow

In the first step, both transponder and reader/writer were described as abstract behavioural models (ABM, Fig. 5a). The reader/writer sends commands to the SmartCard transponder which executes the desired FRAM access and sends the result back to the reader/writer. Commands and replies are exchanged using IEC14443 frames. The interface specification contained just one transaction specifying this frame. With this abstract system model, a functional simulation of the system was performed, and the application software could be implemented. In the next step, the interface specification was refined down to signal level according to the protocol specification in Fig. 4 and simulated in order to verify their correctness (Fig. 5b). Afterwards, the reader/writer was implemented at RT-Level (Fig. 5c). The

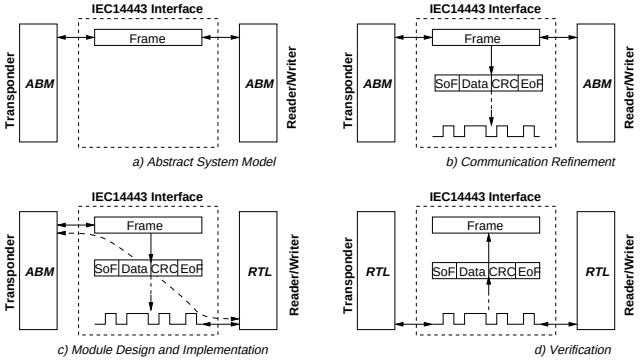


**Figure 5. Interface-based design of a SmartCard transponder-reader/writer system**

major part of this module is the IEC14443 protocol controller, modeled as explicit FSM. With the transponder still being at the very abstract behavioural level, the implementation of the reader/writer was verified by simulation using the same testbench. It was up to the interface to perform the translation of IEC14443 frame transactions generated by the reader/writer from their signal level representation up to their transaction level counterpart understood by the transponder, and vice versa to translate abstract frames sent by the transponder into the signal protocol understood by the reader/writer. Finally, also the transponder was implemented as RT model and simulated against the reader/writer (Fig. 5d). Now with the interface still translating the signal level representation of the IEC14443 frames up to the transaction level, the data stream was automatically checked for protocol errors which would be detected and asserted by the interface.

In conventional system design, both transponder and reader/writer, which are control-dominated designs, would have been specified at RT level with both behaviours being able to generate or consume IEC14443 transactions at signal level in order to perform a first system simulation. For

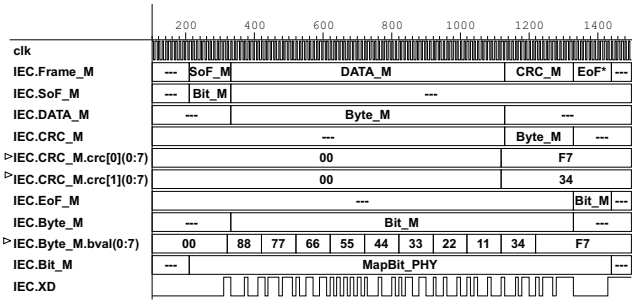


**Figure 6. IEC14443 interface trace in transponder-reader/writer simulation**

protocol checking, a third component implementing a complex automaton that listens to the transmitted signals and checks it for legal states would have been required. With SystemC$^{SV}$, system simulation was possible much earlier in the design cycle, and the proven abstract executable system specification could serve as a simulation environment through all stages of the design.

Figure 6 gives an impression of the interface tracing capabilities of SystemC$^{SV}$. It shows a trace of the IEC14443 interface at all levels of abstraction. For each interface item, the composition state and parameters are automatically recorded by the SystemC$^{SV}$ interface trace engine.

### 4.2. Experimental results

Figure 5a suggests that, due to the high abstraction level of transponder and reader/writer and abstracted communication, the simulation performance of the abstract system specification should be essentially higher compared to the RTL implementation. In order to investigate into this, a total of 5 simulations were conducted with transponder,

reader/writer and IEC14443 interface being at different levels of abstraction. In these simulations, 1000 frames of average length of 8 data bytes were exchanged between transponder and reader/writer. The time required per frame for each simulation is listed in Tab. 1. All examples were compiled with GCC 2.95.2 under Solaris 2.5.1 and run on a SUN UltraSparc with 200 MHz and 512MB RAM. Table entries marked ABM and RTL correspond to reader/writer and transponder at abstract behavioural or RT-level, respectively. For the interface, FL corresponds to Frame level and SL to Signal level. For comparison purposes, the figures for a conventional SystemC RT simulation without interface are shown. The figures show that with the abstract models of

| | Abstraction Level | | | | |
|---|---|---|---|---|---|
| Transponder | ABM | ABM | ABM | RTL | RTL |
| Reader/Writer | ABM | ABM | RTL | RTL | RTL |
| Interface | FL | SL | SL | SL | - |
| Time/Frame | 1.3ms | 0.10s | 0.15s | 0.3s | 0.21s |

**Table 1. Simulation time per frame for mixed abstraction level simulation**

transponder, reader/writer and interface, simulation speed is approximately 160 times higher compared to a conventional RT simulation. However, if the interface is refined to signal level, speed figures are in the magnitude of RT simulation, because the number of events generated in the interface is approximately the number generated in the protocol state machines of transponder and reader/writer.

## 5. Conclusion and future work

We presented an extension to SystemC, which lets system designers take full advantage of the interface-based design methodology in terms of raising the abstraction level in the system specification phase, and designing and verifying system communication separately from functionality. Future work will address the problem of concurrent use of the same interface item by different behaviours, which requires the development of concepts for specification of item arbitration and item queues in SystemC$^{SV}$.

## References


[1] CynApps, Santa Clara, CA. *Hardware Modeling with Cyn++*, 1999. Release 1.1.

[2] Fujitsu FME GmbH, Dreicheich/Buchschlag. *BabyFace2 Chip Specification*, 1999.

[3] D. Gajski, J. Zhu, R. Doemer, A. Gerstlauer, and S. Zhao. *SpecC: Specification Language and Methodology*. Kluwer Academic Publishers, Boston/Dordrecht/London, 2000.

[4] J. Rowson and A. Sangiovanni-Vincentelli. Interface-based design. In *Proceedings of the 34th Design Automation Conference*, Anaheim, CA, 1997.

[5] Synopsys Inc., CoWare Inc., Frontier Design, Inc. *SystemC User's Guide*, 2000.

[6] D. Wilkes and M. Hashmi. Application of High Level Interface-based Design to Telecommunication Systems Hardware. In *Proceedings of the 36th Design Automation Conference (DAC)*, New Orleans, 1999.