ABSTRACT

The introduction of design languages, such as SystemC 2.0, that allow the modelling of digital systems at the transaction level will impose some major changes to the design flows. Since these formalisms allow for a higher level of abstraction in the systems description, new methodological tools will be needed to support all design phases.

The goal of this paper is twofold: first we formalize in an abstract way a significant set of features of a Transaction Level Model, according to the SystemC 2.0 formalism. Then, upon this model we define numerical metrics that can provide useful information in the analysis of the system-level specifications. In particular these metrics are useful in the design exploration phase, to define the main characteristics of the hardware and software architectures.

Categories and Subject Descriptors
B.8.2 [Hardware]: Performance and reliability—performance analysis and design aids

General Terms
Design,Performance

Keywords
Transaction Level Modelling, static analysis, communication implementation

1. INTRODUCTION

In the field of digital design, the use of Transaction Level Models is becoming a key issue in the efficient development of complex systems. Therefore, the implementation of features that allow such an abstraction level in a language of ever-growing popularity such as SystemC, will make its use affordable by more and more design teams. The aid offered by EDA tools should then extend its domain to these higher levels of abstraction.

One of the tools that can be offered to the designer is the automatic extraction from the high level model of information that can help them in the early implementation choices.

These metrics can support the design exploration phase in order to identify target architectures and hardware/software partitions. Furthermore, the formal model and the associated figures of merit could represent the startup point for the definition of a communication synthesis methodology for TLM models.

In the subsequent sections we present an overview of the features of a TLM model. Thereafter, we briefly describe what are the SystemC 2.0 transaction level modelling capabilities. We then introduce a formal model of a TLM specification that mathematically represents some important features of the communication elements of the system. Based on this representation, we introduce the computation of metrics that quantify some characteristics of the system. We propose the use of such metrics as a methodological support to the design process.

1.1 Transaction Level Modelling

The expression Transaction Level Modelling refers to an abstraction level in the description of a system that provides modelling of the communication between the elements that describe the behavior of the system in a functionally, but not pin-accurate way. That is, in a TLM model the focus is on the data that is passed between two modules, rather than on the way the transfer is accomplished.

For instance, it is possible to specify the functional characteristics of the communication, such as the blocking or non-blocking semantics, without defining their implementation. To do so, the designer does not need to use the hardware signal semantics, as it happens in languages such as SystemC 1.0 and VHDL.
One of the many advantages of the introduction of such a modelling style in a specification language is the possibility of obtaining an executable model at a higher level of abstraction, not biased by architectural choices. Most of the implementation choices will be performed after this early modelling phase. Thus, a TLM model of a system can describe an abstract system that can be mapped onto different architectures.

Transaction level modelling was first introduced in hardware specification languages in SpecC [11], and later developed under the name of behavioral wrappers in [10] and as Functional Interface by the VSIA [4].

1.2 TLM and functional modelling in SystemC 2.0

SystemC [1] [6] [8] [7], starting from its version 2.0, introduced the following language elements to support transaction level modelling:

- interfaces: an interface is a set of pure virtual C++ methods;
- channels: a channel is a module that implements one or more interfaces;
- events: the event is the basic synchronization item.

Modules have ports that are typed upon an interface type. A module’s port can be connected to a channel that implements the interface type of the port. This formalism is called Interface Method Call (IMC).

Threads and methods can suspend their execution upon the notification of an event, and can in turn notify events.

The hardware-oriented communication signals that characterized SystemC 1.0 can be described through the use of these elements, plus a specialized family of channels, called primary channels, that implement the evaluate-update semantics. This shows that the introduction of such concepts generalizes the classical HDLs communication elements.

Another effect of the introduction of these concepts is the separation of the behavior from the communication modelling within the elements of the system. In this way, in addition to the TLM capabilities, a mechanism is offered that allows the encapsulation of the description of the communication between “functional” items (the modules) into ad hoc elements (the channels).

2. MOTIVATION OF THE TLM ANALYSIS

The adoption of a TLM level of abstraction in the modelling of a digital system poses new problems that should be addressed by EDA tools [2]. In particular, the higher abstraction level implies that the models described at the TLM level can represent functional descriptions of the system prior the hw/sw partition and even before the choice of the target architecture.

The SystemC formalism however requires to functionally decompose even the very initial models; so, the design flow starts from a model that does not imply a particular implementation, yet it has been conceptually structured into communicating elements. In such a scenario, the designer has to face a set of decisions that prelude to the implementation of the system on a particular target architecture. Starting from a model described at a TLM level of abstraction, where communication is characterized through the use of channels, one possibility would be to extract a set of figures of merit that may support the designer in the analysis of those aspects of the design useful for selecting the target architecture, which in turn implies:

- the decision on the type and number of computation elements (processors, ASICS, DSP, etc.);
- the choice of the communication means (buses, fifos, shared memories);
- the hw/sw partitions.

The set of figures of merit we propose in this paper addresses the analysis of the communication elements present in a SystemC 2.0 description and can provide some useful information on a model described at the TLM level. Once extracted, this information could be used in different ways. For instance, patterns of these figures could be recognized, suggesting one architectural choice over another. In particular, the figures related to the channel analysis could suggest the use of a communication mean rather than another. The metrics proposed can be computed with a static analysis of the model, and belong to four categories:

- **width metrics**: these figures represent the width of the data token passed through a connection;
- **direction metrics**: they represent the read/write behavior of a connection;
- **synchronization metrics**: they describe the presence of *synchronous* behaviors in communication channels, that is, of service requests where the initiator suspends itself until its request is served (as in a rendez-vous); on the other hand, in *asynchronous* connections the initiator continues its operation regardless the immediate effects of its request;
- **memory occupation**: they provide an estimation of the memory size of the elements of the system.

3. RELATED WORK

Metrics computation has been widely used in the fields of hardware/software codesign and power estimation.

Static metrics are used to estimate affinity between functional elements, to allow some form of clusterization, aimed at the partitioning of the system. A wide set of behavioral metrics have been developed by Vahid [9] for system-level partitioning.

Static analysis of SystemC models is also used in [5], for the timing analysis of descriptions based on parallel communicating processes. The analysis, in this case, is aimed at determining some temporal properties of the system such as the worst-case response time.

The problem of the implementation of transaction level models has been addressed by Grötker et al. in [3]. Here the authors show how to use the SystemC 2.0 to refine high level models into descriptions closer to the implementation; the analysis focuses on the modelling capabilities of the language, that allow the refinement of the description towards its implementation, rather than proposing a methodology for the synthesis of such models. Moreover, the problem of describing abstract models at the transaction level is extensively considered.

No specific metrics have been defined to evaluate Transaction Level and Functional models for architecture exploration.

4. FORMAL MODEL

In this section, we present the formal model defined to represent TLM systems. It formalizes some aspects of a TLM model necessary for the metrics computation. This model abstracts from a SystemC 2.0 model some of the main TLM characteristics.
4.1 SystemC components and features

Before defining our abstract transaction level model, let us introduce the following definitions, describing components of the SystemC 2.0 models.

\( Mod \) is the set of all module instances \( M_i \) in the system

\( Mod_{st.} \subseteq Mod \) is the subset of modules instantiated in module \( M_i \)

\( Ch \subseteq Mod \) is the set of all channels present in the system

\( P(M_i) \) is the set of ports instantiated in a module \( M_i \)

\( P \) is the set of all ports in the system, \( P = \bigcup_{M \in Mod} P(M) \)

\( bound(p,c) \) is a function \( P \times Ch \rightarrow \{ \text{true, false} \} \), that is true if port \( p \) is bound to channel \( c \), and false otherwise

\( DTy pes \) is the set of all data types used in the system

\( sizeof(t) \) is a function \( DTy pes \rightarrow \mathbb{N} \cup \{ \infty \} \) that indicates the size in bit of the memory space needed to allocate an instance of type \( t \), where \( sizeof(\text{void}) = 0 \) and \( sizeof(t) = \infty \) if \( t \) has unbounded potential memory occupation, as in a dynamic array

\( B \) is the binding set of the system, defined as \( B = \{(M_i, M_j, p) \mid p \in P(M_i) \land bound(p, M_j)\} \)

4.2 TLM components and features

In our model, the system is represented as a labelled directed graph \( G = (N, E, L), E : N \times N \times L \) such that there is an isomorphism \( F = (f, g, h) : (N, E, L) \rightarrow (Mod, B, P) \). We call the edges of our graph connections.

Method Signature a pair \( m = (D, C) \), where \( D \) and \( C \) are multisets, \( \forall t \in D, C, t \in DTy pes \), and \( D \) and \( C \) are the domain and co-domain of the corresponding method in the SystemC description, where we take into account non const arguments passed by reference

\( M \) is the set of all method signatures

Method is an implementation of a method signature \( m \)

Interface is a set of method signatures present in the system

\( I \) is the set of all interface instances \( i \) in the system

\( I(n) \) is the set of interfaces implemented in a module \( f(n), n \in N \)

Channel is a node \( n \) such that \( I(n) \neq \emptyset \)

Type(e) is a function \( E \rightarrow I \) that returns the interface type of a connection

The graph \( G \) provides information on the topological structure of the system. The direction of edges, e.g., determines whether the communication between two modules follows the master-slave paradigm or each module can take an active role. The dominators tree allows the detection of the nodes that control the execution of part of the system.

5. METRICS

So far, our model provides topological information on the presence of communication between computation nodes. We want to enrich this information by specifying both qualities of the connections, such as the size of the data tokens passed through them and the direction of the information flow, and qualities of the nodes, such as their memory occupation.

We also want to add information about the dependencies induced by synchronization statements.

5.1 Communication Width

These metrics provide information on the width of the tokens involved in the data transactions.

Let us first define the width \( W \) of a data type \( t \in DTy pes \) as \( W(t) = sizeof(t) \), and the width of a multiset \( D \) of types as \( W(D) = \sum_{t \in D} W(t) \).

For a single method \( m = (D, C) \), we can define the width \( W \) as

\[
W(m) = W(C) \oplus W(D)
\]

The \( \oplus \) operator can be any operator such that \((N, \oplus)\) is a commutative monoid.

For an interface \( i \in I \),

\[
W(i) = \bigoplus_{m \in I} W(m)
\]

and for an edge \( e \in E \), \( W(p) = W(type(g(e))) \).

Finally, for a node \( n \in N \),

\[
W(n) = \bigoplus_{e=(n,\ n')} W(g(e))
\]

and for a pair of nodes \((n_1, n_2)\), Communication Width can be defined as

\[
W(n_1, n_2) = \bigoplus_{e=(n_1, n_2)} W(g(e))
\]

Let us now define the Communication Width \( \hat{W} \) between two nodes connected through a set of channels. First we define the width of the communication between two modules through a single channel:

\[
\hat{W}(n_1, n_2, c) = W(n_1, c) \oplus W(n_2, c)
\]

Now we can define the communication width between nodes connected through an arbitrary number of channels: \( C(n_1, n_2) = \{c \mid \exists e_1 = (n_1, c, l_1), e_1 \in E \land \exists e_2 = (n_2, c, l_2), e_2 \in E \} \)

\[
\hat{W}(n_1, n_2) = \bigoplus_{e \in C(n_1, n_2)} \hat{W}(n_1, n_2, c)
\]

These definitions identify a family of metrics, parameterized by the operator \( \oplus \). Some significant operators would be, for example, the addition and the maximum. The former would define a metric that computes the bit size of all tokens that can be exchanged between two modules, while the latter would compute the largest data token exchanged. Both metrics would be useful, though for different purposes.

These metrics could be used to estimate the size of the communication medium needed to implement the connection. For instance, if the medium were a parallel bus, these metrics could be used to estimate the number of lines required.

Note that these metrics can be applied to different topological structures: modules connected to channels, channels themselves, modules connected through channels.

5.2 Directionality

The directionality metric \( D \) accounts for the nature of communication between two modules, and attempts to detect whether the connection is read- or write-only, or the communication includes control information only.

For method signatures, \( D(m) = d \in \{r, w, +, c\} \), where the possible values of \( D(m) \) represent a read-only, write-only, read/write or control-only communication. A method \( m = (D, C) \) is read-only when \( W(C) \gg W(D) \), it is write-only when \( W(C) \ll W(D) \), control-only when all the elements in \( D \) and \( C \) are event
5.4 Synchronic Behavior

5.4.1 Implicit Synchronization

When a port in module $M_1$ is used to call a method implemented in module $M_2$, that method may in turn call a method $M_3$ from a port of $M_2$. Now, there is a synchronization between $M_1$ and $M_3$ through channel $M_2$. We detect this type of synchronization by constructing the transitive closure of the call graph of our system model.

Given the graph $G_C = (\text{Method}, E_{C_v})$, where $E_{C_v} \subseteq (\text{Method}, \text{Method})$ and for $m_1, m_2 \in \text{Method}$, $(m_1, m_2) \in E_{C_v}$ iff $m_1$ may call $m_2$, we build its transitive closure $G^*_C$.

The graph induces an Implicit Synchronization metric between two methods,

$$\text{ISyn}(m_1, m_2) = 1, \text{ if } f(m_1, m_2) \in E_{C^*_v}.$$ 

$$\text{ISyn}(m_1, m_2) = 0 \text{ otherwise}$$

For two nodes $n_1$ and $n_2$ and a label $l$ such that $p = h^{-1}(l) \in P(f(n_1))$ we define

$$\text{ISyn}(n_1, n_2, l) = \bigoplus_{\forall m_2 \in h^{-1}(l) \forall m_j \in I(n_2)} \text{ISyn}(m_1, m_j).$$

And, generalizing,

$$\text{ISyn}(n_1, n_2) = \bigoplus_{\forall p \in P(f(n_1))} \text{ISyn}(n_1, n_2, p).$$

Here, the operator may be the arithmetic sum, if we want to stress the degree of synchronization between two modules, the logical sum if we want to map synchronization constraints in the whole system, without taking into account the number of constraints, and the logical product if we want to highlight asynchronous behaviors rather than synchronic ones. This last choice is not to be used in the last formula, since it would introduce spurious contributes from unrelated ports in the first module.

5.4.2 Explicit Synchronization

Once more, we first define the metric for methods. We define two functions $wset$ and $sset (E, M) \rightarrow \text{Events}$, where $\text{Events}$ is the set of events in the SystemC 2.0 model. These are respectively the set of events which a method $m \in M$ available to a connection $e \in E$ can be set in waiting for, and the set of events it can raise.

Therefore, the rate of Explicit Synchronization $E\text{Syn}(x, y)$, $x, y \in (E, M)$ between two connection method pairs is defined as

$$E\text{Syn}(x, y) = |wset(x) \cap sset(y)|$$

Note that $E\text{Syn}(x, y) \neq E\text{Syn}(y, x)$.

We define

$$\text{E} \text{Syn}(e_1, e_2) = \bigoplus_{\forall m_1 \in \text{E}(e_1), \forall m_2 \in \text{E}(e_2)} E \text{Syn}(m_1, p_{1,c}, m_2, p_{2,c})$$

the Explicit Synchronization of a couple of connections associated with the same channel, where $e_1 = (n_{1,1}, c, l_1)$ and $e_2 = (n_{2,1}, c, l_2)$.

We can extend this metric to nodes, defining $E\text{Syn}(n_i, n_j)$ as the sum of the Explicit Synchronization metric for all couples of connections associated with the same channel.

We first define the set

$$C(n_i, n_j) = \{ c \in M | \exists(n_i, c, l_i) \in E, \exists(n_j, c, l_j) \in E \}$$

of channels accessed by both modules $n_i$ and $n_j$.  

Table 1: Operators over the $A$ set

<table>
<thead>
<tr>
<th>$\oplus$</th>
<th>$r$</th>
<th>$w$</th>
<th>$c$</th>
<th>$-$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r$</td>
<td>$r$</td>
<td>$r$</td>
<td>$r$</td>
<td>$r$</td>
</tr>
<tr>
<td>$w$</td>
<td>$w$</td>
<td>$w$</td>
<td>$w$</td>
<td>$w$</td>
</tr>
<tr>
<td>$c$</td>
<td>$c$</td>
<td>$c$</td>
<td>$c$</td>
<td>$c$</td>
</tr>
</tbody>
</table>

We can now define

$$D(i) = \bigoplus_{m \in E} D(m)$$

for an interface $i \in I$, and, for a single node $n \in N$, $D(n) = \bigoplus_{(n, x) \in E_{\text{var}}} D(\text{type}(g(e))) - \bigoplus_{j \in f(n)} D(i)$. 

For an ordered pair of nodes $(n_i, n_j)$, we define

$$D(n_i, n_j) = \bigoplus_{x \in (n_i, n_j) \in E} D(\text{type}(e)).$$

This metric can highlight unidirectional communications between modules, therefore suggesting implementation choices such as pipelines, FIFOs for hardware-hardware solutions; for software-software systems, this could affect the implementation of interprocess modules, therefore suggesting implementation choices such as pipelines, FIFOs for hardware-hardware solutions; for software-software systems, this could affect the implementation of interprocess communication, for instance revealing the need for locking policies.

5.3 Memory Size

Memory Size metrics estimate the size of the state space of the elements of the system.

Let $\text{Var}_n$ be the set of all attributes of module $f(n)$ that are not modules themselves, then $\text{size}(n)$ can be defined as:

$$\text{size}(n) = \sum_{v_j \in \text{Var}_n} \text{size}_o(f(\text{type}(v_j))) +$$

$$+ \sum_{M_i \in \text{Mod}_n} \text{size}(f^{-1}(M_i))$$

In the case of a non-structured module, $\text{Mod}_n = \emptyset$, thus the Memory Size is just the memory occupation of that module.

These metrics can discriminate different implementation options, depending on the size of the state space. It is possible to choose between combinatorial (as a bus) and sequential (as a shared memory) communication solutions.

5.4 Synchronic Behavior

This metric detects synchronization dependencies between elements of the system, such as modules.

We detect synchronization constraints at two levels. First, we are interested in extracting synchronization information from the interprocedural control flow graph. This allows us to detect which modules are implicitly synchronized by their connections. Then, we add the synchronization constraints explicitly imposed by waiting on events.

We call the two metrics respectively Implicit and Explicit Synchronization.
Now, we can define $ESyn(n_i, n_j)$ as

$$ESyn(n_i, n_j) = \bigoplus_{e_k \in E^\prime} ESyn(e_k, e_k)$$

The two synchronization metrics can highlight the need for synchronization protocols in the implementation. For instance, if the synchronization between two modules is greater than zero, the synchronous behavior could be implemented by some handshaking protocol.

### 6. EXPERIMENTAL FRAMEWORK

To assess the effectiveness of the metrics so far proposed, we set up an experimental framework. The goal of the experiments is twofold: we want to apply the methodology to a significantly complex, realistic model, and, at the same time, to be able to manually verify the correctness of the conclusions drawn from the computed metrics.

As target for this validation example we chose a decoder of a differentially encoded data flow, similar to the one employed for the transmission of video streams such as the mpeg.

The system we modelled is at the functional untimed level; structurally, it is composed of three sets of functional modules:

- a decoder module, that implements the interface towards the external world; it receives different types of data packets and accordingly invokes the appropriate decoding service;
- a set of ALUs; these are modules that implement different operations needed by the decoding task; in our system, these perform duties such as storing the data flow state and computing the difference between the state and the incoming data encapsulated in the packets;
- a channel that acts as a dispatcher between the decoder and the ALUs modules; it essentially receives service requests from the decoder, dispatches them to the appropriate ALU and communicates the results to the decoder.

We considered the ALUs as library IPs, with given interfaces. The channel was then built to adapt the interface required by the decoder to the ones provided by the ALUs. We considered two scenarios, characterized by different functional interfaces required by the decoder, which induced different implementations of the dispatcher.

We then computed the metrics for both cases; their results, shown in part in tables 2 and 3, prompted for specific implementation solutions of the communication channel. These solutions turned out to fit the results of the manual analysis performed by a designer.

Let us see how the metrics obtained can be exploited in order to make some architectural choices. In the first case, we can see that the value of $ISyn$ between the decoder and the ALUs shows the need for synchronization between these components. Since their communication is accomplished through the dispatcher, the implementation of this module needs to provide some synchronization mechanism. The $size$ metrics, computed for the dispatcher has a value equal to zero in the first case. This means that the communication can be implemented with a stateless resource. Moreover, the decoder has master relationships with all the ALUs. This information strongly biases the architectural choice towards a shared communication resource such as a bus, with a protocol that implements synchronization.

In the second case, the asynchronous relationship highlighted by the $ISyn$ metric with $\oplus = \lambda$, together with the memory requirements for the dispatcher, $size(disp) = 16348$, show that a stateless solution like a simple bus is not feasible. A possible alternative could include a bus between the ALUs and an adapter that is connected with a point to point connection to the decoder, and implements the asynchronous behavior with some buffering functionality.

### 7. CONCLUDING REMARKS

In this paper we defined an abstract model that formalizes most of the information of the communication aspect of a TLM model, as those described in SystemC 2.0.

Upon this model we defined a set of metrics that provide information useful for the some of the implementation tasks, such as architectural mapping and hardware/software codesign.

We proposed some sample applications of these metrics, although we think they can be applied in other design scenarios as well.

Further work will include an extensive application of the metrics to different design examples and a definition of a communication synthesis methodology based on the metrics result. We are also working to extend the formal model in order to cover behavioral elements of the design specifications, and to consequently define metrics for these aspects.

### 8. ADDITIONAL AUTHORS

### 9. REFERENCES


