Static Analysis of Transaction Level Communication Models

Giovanni Agosta, Francesco Bruschi, Donatella Sciuto

***Abstract*— We propose a methodology for the early estimation of communication implementation choices effects, starting from an abstract transaction level system model (TLM). The reference version of TLM considered is the OSCI library. The methodology is based on the computation of metrics that abstract useful information from the initial system model. The metrics are precisely defined upon a general, formal model of transaction level system descriptions. A set of design problems of relevant interest, such as shared communication resources assignment, pipelining partitioning, bandwidth and latency constraints estimation, is considered to show some potential applications of the metrics proposed.**

I. Introduction

The aim of this work is the definition of an analysis methodology of the communication properties in a high-level system model. This information provides a support to the designer in gaining a better insight on the system characteristics. The outcome of the analysis can be exploited by the designer to perform a better partitioning of the system and to obtain a more effective synthesis of the communication parts.

Simulation-based validation through executable models of a system is commonly employed in the embedded systems design community in the first phases of the design process to evaluate the different architectural alternatives. Typical advantages claimed in favor of the adoption of such models are that they can help in better understanding the specification, allowing to highlight unnoticed ambiguities, and that they can act as a formal reference for the system implementation.

The ability to model communication without specifying an implementation choice allows easier and faster writing of the system models that can be used as *functional* specification of the system.

Still, part of the current EDA research effort originates from the assumption that such models implicitly contain information that would allow the automatic implementation of the communication between the system components and the early estimation of its effects on performance.

The first assumption is the conceptual basis for the abstract *communication* synthesis approaches. The second assumption implies that it is possible, starting from a purely functional description, to compare different implementation choices without having to directly refine a model or to generate a prototype. The information extracted from the analysis of the abstract system model could be exploited to guide architectural choices to determine the number and type of resources to employ. Furthermore, such information can drive the automatic synthesis of parts of the system, to drive automatic algorithms in achieving optimal or satisfying results (such as, for instance, various types of resource sharing).

A. Design problems addressed

The implementation of a high-level communication model requires *mapping* onto physical components, that includes *resource sharing*, *performance* (bandwidth and latency) *estimation*, *dimensioning* (of buses width, for instance).

In this context, *mapping* means finding a correspondence between the communication channels of the model and the physical channels of the target platform. An important problem is to decide what model channels can be mapped onto the same architectural elements (*resource sharing*). To this aim it would be useful to know what model communication channels are most likely to be activated at the same time, risking to cause access conflicts. This problem, if addressed with brute force simulation approaches, is computationally very complex (the number of configurations is equal to all the possible partitions of the communication channels with a given number of classes, a number that diverges rapidly).

Performance estimation and *dimensioning* are strictly interrelated. In particular, it would be interesting to know in advance the effect of given dimensioning choices on the performance of other parts of the system. Exemplifying, a typical problem could be: let the communication channel A be implemented with a component that can guarantee at most the bandwidth b; how will this affect performance of the communication on channel C? Will

this choice impose an upper constraint on the bandwidth of *C*? Even for this problem, a brute force simulation approach would be impractical.

The intrinsic complexity of a simulation approach to these problems suggests the exploration of static analysis techniques, which would be valuable to support the designers in making effective choices by better understanding the specification features.

In our work we determine the information that can be obtained statically and that can be useful for the solution of the problems considered.

This information can be either structural, identifying the communication dependencies, or relate to communication performance. *Communication Dependencies Information* allows detection of such dependencies, that may be hidden, but allows also the estimation of the communication load and latency. *Communication Performance Information* can be either local or global. Local performance can be characterized by latency and throughput of specific communication channels. Global performance allows the identification of possible bottlenecks but also the detection of the available parallelism.

Dependencies Information can be valuable independently from *Performance Information*, or be exploited as a basis for evaluating the latter.

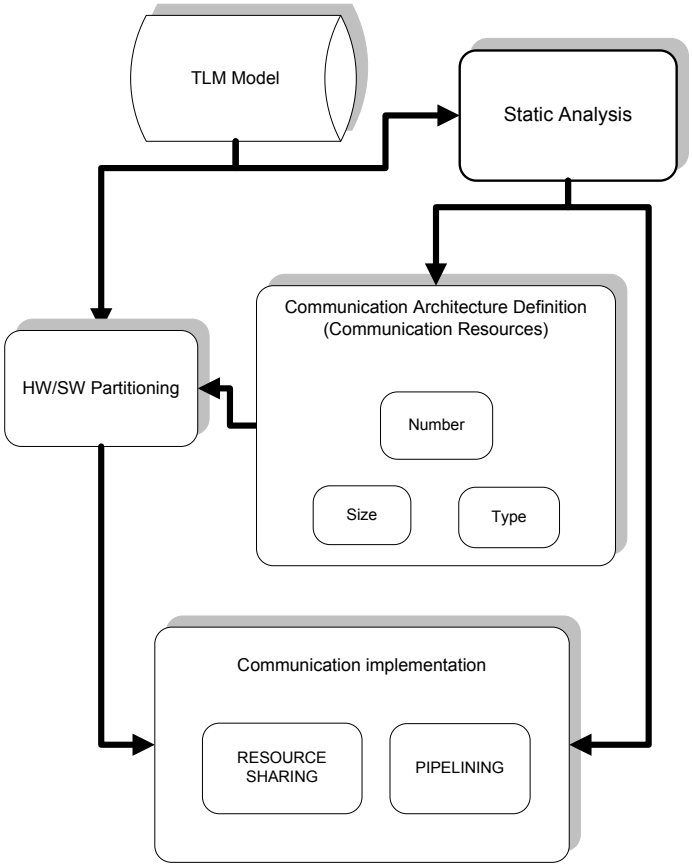

Fig. 1. Overview of the implementation of system communication by exploiting static analysis

B. Proposed methodology

The analysis methodology proposed in this paper aims at the extraction of the above information by means of a static analysis of the early, transaction level based executable model.

This is obtained by defining a set of *metrics* that can be statically computed. The role of the analysis in an implementation flow (here emphasis is on communication implementation) is shown in Figure 1.

It is worth noting that the metrics proposed are not in a one to one correspondence with the information categories mentioned. The link between their estimation and the achievement of the desired information is obtained with a further processing step. Nevertheless, metrics are formulated with these goals in mind, so it is possible to apply each of them in the estimation of a particular information category among those specified.

Among the system level design formalisms available, we choose to analyze SystemC Transaction Level Models as defined by the Open SystemC Initiative consortium (OSCI). The OSCI TLM library was designed to obviate to the lack of standardization in SystemC Interface Method Call (IMC) formalism used to describe communication at a high level of abstraction.

C. Paper organization

The rest of this paper is structured as follows. Section II provides an overview of the related works. In Section III the input SystemC subset to which the analysis can be applied is defined. In Section IV we introduce a mathematical representation of the system, used in section V to define the metrics. Some computation examples are provided in section VI. Section VII shows the application of the defined metrics to some relevant design problems such as shared communication resources assignment, pipelining partitioning, bandwidth and latency constraints estimation. Section VIII provides a larger application scenario based on an industrial case. Finally, Section IX draws some conclusions.

II. Related Works

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 [11] for system-level partitioning.

Static analysis of SystemC models is also used in [9], 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.

Transaction level modeling was first introduced in hardware specification languages in SpecC [13], and later developed under the name of behavioral wrappers in [12] and as Functional Interface by the VSIA [4].

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 SystemC 2.0 to refine high level models into descriptions closer to the implementation; the analysis focuses on the modeling 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.

In [5] the authors present a set of metrics that extract, from C specifications, information that characterizes system behavior with respect to three different aspects: potential parallelism available, amount of memory access operations, amount of control component. The metrics are defined and computed relying on a hierarchical graph representation of the initial C specification. Metrics values provide information that can be exploited for architectural choices such as component pipelining, memory access optimization effectiveness, amount of architecture parallelism. The approach presented in [5] is somehow similar to that of the present work, in that it aims at providing designers with statically computable information, useful for certain architectural choices. The results of [5] are complementary to what here presented, in the sense that the concerns addressed are fairly different: we focus on the communication architecture rather than on the choice and design of the processing elements.

Another approach supporting design choices by means of statically computed information can be found in [8]. In this work, the information extracted from C based specifications is exploited to estimate whether given code is most suitable for general purpose processors, digital signal processors, or on application specific integrated circuits. Entry point of the computation is a C specification, that is translated into a control data flow graph based representation. A set of *affinity metrics* between the code and the three implementation technologies are then computed on this representation. Metrics are based on the taxonomy of instructions of the intermediate representation, with respect to their supposed affinity with different architectures. Affinities are then computed as the fraction of instructions of the class affine to one implementation technology. The work is extended in [7], where three metrics are added: *load indexes communication indexes*, and *physical cost index*. These metrics are meant to be computed to support system partitioning, and give information on aspects such as the balance between computation and communication, processor loads, and implementation cost.

In [2], we defined a set of metrics potentially useful for the estimation of the effectiveness of several communication implementation choices. In this work, the set of metrics presented in [2] has been significantly extended with the introduction of *execution dependent* metrics, that allow to take into account statistical analysis on the estimated information. More details are provided in section V. Moreover, in this work, specific applications of the proposed metrics are formally defined and tested on a design example. Another significant improvement over [2] is the definition of a precise mathematical abstraction of a SystemC transaction level model.

TABLE I
TLM INTERFACES SUMMARY

Interface	Data flow	Synch type	Methods
tlm_blocking_get_if	←	blocking	get()
tlm_blocking_peek_if	←	blocking	peek()
tlm_blocking_put_if	→	blocking	put()
tlm_nonblocking_get_if	←	non-blocking	get(), nb_can_get()
tlm_nonblocking_peek_if	←	non-blocking	peek(), nb_can_peek()
tlm_nonblocking_put_if	→	non-blocking	put(), nb_can_put()
tlm_transport_if	↔	blocking	transport()

III. SYSTEMC TRANSACTION LEVEL MODELING DEFINITION

This section summarizes the definition of the core Transaction Level Modeling library, proposed by OSCI as a standard for this level of abstraction. The library is freely available for download from the OSCI site [1]. As explained and motivated in the previous section, this TLM formalization will be considered as definition of the language input for the metrics computation.

The TLM formalization proposed by OSCI (simply "TLM" in the rest of this paper) is, in its essence, composed of a set of predefined, parametric base interfaces. Whereas a SystemC interface is composed of an arbitrary set of method signatures, TLM provides a fixed base set of defined interfaces, each with a given number of methods, whose semantics is determined. TLM interfaces are characterized by three main choices: whether their methods have a blocking or non-blocking behavior, whether the data transmission is bidirectional or unidirectional, and what kind of data is passed when the interface methods are invoked.

Given these basic characterization criteria, TLM defines a set of seven interfaces that represent homogeneous communication transactions, as shown in Table I. *A SystemC model is TLM compliant if all its channels implement only interfaces among those listed in Table I.*

All the interfaces are parametric with respect to one (unidirectional) or two (bidirectional) data types.

In the rest of this section, we will deal with the two main features of those interfaces: data flow direction and synchronization, since these features of TLM models will be exploited in the analysis presented in the next sections. A more in-depth coverage of SystemC TLM modeling can be found in [6].

A. Synchronization

SystemC allows the description of modules behavior with two types of processes: SC_METHOD and SC_THREAD. The main difference between the two is that an SC_METHOD cannot suspend its execution waiting the occurrence of an external event, while an SC_THREAD can do so. Suspension, in SystemC, is achieved through invocation of the *wait* method. SC_METHODs cannot invoke a wait() statement, nor they can invoke any function or service that invokes it. Adopting a widespread terminology, OSCI calls *non–blocking* any function that *is guaranteed* not to call any wait, directly or indirectly, and *blocking* all the others. This concept is formalized in TLM: all the interface methods are characterized as *non-blocking* or *blocking*, depending whether they guarantee or not that calling them will not lead to a wait() suspension.

Non-blocking interfaces provide methods to check whether the non-blocking request is likely to succeed (e.g., *nb_can_get*) and methods that return an *sc_event* that is notified when the non-blocking action, if called, would likely succeed. These primitives allow the designer to adopt synchronization models for the non-blocking control scheme that are closely modeled on the *interrupt* and *program control* peripheral interaction modes.

B. Data flow direction

In addition to managing different types of synchronization methods, TLM provides both unidirectional (*put* or *get*) and bidirectional (*transport*) transactions. The rationale behind this choice is that any arbitrarily complex communication protocol can be broken down into a sequence of unidirectional (or bidirectional) transactions. On the other hand, the possibility of defining both unidirectional and bidirectional communications allows to easily model different types of interactions, such as a read across a bus or a network packet transmission, without breaking them down in smaller components or requiring more resources than what is needed.

C. TLM 2.0 Extensions

The need for features not present in the initial TLM specification has led to the development of a second TLM standard proposal (*TLM 2.0*). TLM 2.0 presents several novel or variant abstractions, mainly aimed at improving performance, allowing finer timing specification and augmenting the expressive capabilities of the formalism. While TLM 2.0 is still in its second draft form, it is worthwhile to consider how the proposed extensions will affect the methodology proposed in this paper.

First, a new transport interface has been defined, which, instead of performing bidirectional data flow, concentrates information into a *transaction* argument. This allows faster simulation, but preserves the transport semantics.

For timing expressiveness, a new level of accuracy, called *loosely-timed*, is introduced. The possibility of expressing timing information is present in TLM 1.0 as well, though not in a standardised form. Our work is focused on the analysis of functional specifications, without timing information. Thus, the new level of timing accuracy does not affect the system models considered in the rest of the paper.

Beyond some syntactic adjustments, the proposed methodology applies indifferently to TLM 1.0 and TLM 2.0 models at the untimed functional level of abstraction.

IV. System Representation

In this section, a mathematical abstraction of the TLM SystemC models is presented. This representation serves as basis for the definition and computation of the communication estimation metrics in Section V. In the remainder, the structure of this information will be referred to as *MOIR*, that is *Metric Oriented Intermediate Representation*.

A MOIR of a given model is basically an *annotated graph* of connected components:

$$D =< M, E >$$

where M is the set of components and E is the set of edges.

The components are defined according to the OSCI TLM interpretation of SystemC [6].

A. Components

A *module class* m is defined as a tuple $m =< id_m, I_m, P_m, T_m >$ where id_m is a unique identifier; M_C is the set of module classes; $I_m \subseteq I_{TLM}$ is the set of *interfaces* implemented by the module class m; P_m is the set of *ports* of m; and T_m is the set of *processes* belonging to m. Processes, in a SystemC module, describe the reactive behavior of a component.

A *port* p is defined as $p =< id_p, i_p >$ where id_p is a unique identifier and $i_p \in I_{TLM}$. It can be interpreted as an interface *required* by a given module m.

A module class m such that $I_m \neq \emptyset$ is defined a *channel class*. A channel, in SystemC, is in fact any module that offers some TLM service. The set of channel classes C_C is therefore defined as $C_C = \{m | I_c \neq \emptyset\}$.

Components c are instances of module classes, defined as $c =< id_c, m_c >$ where id_c is a unique identifier and m_c is a module class. The sets of module components M and of channel components $C \subseteq M$ are also defined. M is the set of all the instances of module classes in the system, and $C = \{c | I_{m_c} \neq \emptyset\}$.

In the remainder of the paper, we will refer to ports P_c, interfaces I_c and processes T_c of a component $c =< id_c, m_c >$, meaning the ports P_{m_c}, the interfaces I_{m_c} and the processes T_{m_c} of the module class m_c.

B. Processes and Events

For our purposes, the most interesting distinction of processes in Transaction-Level Models is between *methods* and *threads*. The notable difference between the two is that the former are guaranteed to be *non-blocking* (they cannot suspend themselves and cannot invoke *blocking services*), while the latter can be *blocking*. A formal specification of the synchronization properties will be given in Section V.

Processes belonging to different modules can communicate via *service invocations*, which provide a point-to-point form of communication between a process (or a service) and a service. To allow intermodule process to process synchronization, *events* are employed. Events implement the rendez–vous synchronization semantics.

Each event is represented by a unique identifier (or tag). An event can be *notified*, according to SystemC execution semantics, and it is possible for a thread to wait for it to be notified.

In MOIR, processes are characterized by their *Communication Control Flow Graph*, that is a graph that is obtained from the classical *Control Flow Graph* by collapsing all nodes that do not represent a service invocation, a wait suspension or an event triggering.

The Control Flow Graph of a process is a directed graph in which each node represents a *basic block*, that is a sequence of statements that have a single entry and a single exit point. The edges of the Control Flow Graph represent the flow of program control from one basic block to the next. According to the purposes of our analysis, only communication activities are considered relevant. Consequently, all nodes that do not contain a service invocation, a wait invocation or an event notification are collapsed, and the result is the Communication Control Flow Graph.

A process can then be represented as a tuple $t =< id_t, Cfg_t >$, where id_t is a unique identifier and Cfg_t is the Communication Control Flow Graph.

From the Communication Control Flow Graph, we can extract different sets of elements characterizing a process:

- the set e_t^s of events the process is sensitive to (that are in its *sensitivity list*);
- the set e_t^w of events the process can suspend upon until notification;
- the set of events e_t^f it can notify;
- the set of services s_t^c it can call through the component ports.

C. Services and Interfaces

In SystemC, *channels* implement *services* that can be invoked by other modules to achieve communication. An *interface* defines a set of methods that are provided, together, by a channel.

In MOIR, an *interface* can then be represented as a triplet $i =< b, d, T >$, where b represents the blocking or non-blocking characteristic of the services of the interface; $d \in \{r, w, rw\}$ represents the uni-directional or bi-directional characteristics of the interfaces, together with the data flow direction (outcoming or incoming); and $T \in DataTypes$ represents the data template parameters of the interfaces.

The set of all interfaces I_{TLM} is composed of the seven Transaction-Level interfaces defined in [6], parametric with respect to the data types.

The building blocks of the interfaces are the *services*, that is the methods that a component provides when it implements a given interface. A *service signature* is represented by a pair $s_s =< id_s, i_s >$, where id_s is a unique identifier and $i_s \in I_{TLM}$ is an interface.

Services are associated with component classes, and can be defined as tuples $s =< c, s_s, Cfg_s >$, where c is a component class, s_s a service signature and Cfg_s is the Communication Control Flow Graph associated with the service.

From the Communication Control Flow Graph we can extract different sets of elements of interest for each service:

- e_s^w the service can wait upon for notification;
- a set of of events e_s^f it can notify;
- a set of other services s_s^c it can call.

D. Synchronization Properties

A service s is non-blocking when it never waits for event notification ($e_s^w = \emptyset$), and it never calls a blocking service. The TLM non-blocking interfaces guarantee that both properties are respected by every non-blocking service s.

As far as processes are concerned, *methods* are guaranteed to be non-blocking, since they comply to the same constraints as the non-blocking services.

For a method t, $e_t^s \neq \emptyset$ because the method is sensitive to a predetermined set of events, defined outside the Control Flow Graph.

E. Connections

Connections represent the binding of module components (module classes instances) to channel components (channel classes instances). Formally, a *connection* e is represented by a tuple $e =< m, c, p, i >$ where $m \in M$ is a module, $c \in C$ is a channel, $p \in P_m$ is a port of m and $i \in I_c$ is an interface implemented by the channel c.

The set E of all the connections in the system is also the set of edges of the system representation graph.

A constraint is imposed on the connections, so that every port connects to one and only one channel. No such constraint is imposed on channel interfaces, so any number of ports requiring the same interface i can be bound to the same channel that implements i.

F. Extension to Hierarchical Models

So far, a MOIR describes a "flat" system. Actually, it is desirable to be able to describe hierarchical models, where a module can in turn be composed of several sub-components connected via services and synchronizations.

The MOIR can be easily extended to describe such hierarchical models: consider a system S defined in MOIR, where a set of services and a set of ports are implemented (respectively, required) by a special black box component, the *system environment*. If we consider S as a module class, the system environment is the generic system in which S can be instantiated. Therefore, S can be seen as a module class m_S where I_{m_S} is the set of interfaces offered to the system environment, P_{m_S} is the set of ports that will connect to interfaces provided by the system environment, and T_{m_S} is the set of processes in the system. Then, m_S can be instantiated in a system as any other component. In a hierarchical model, we will therefore call M_S the set of components instantiated within the top-level component S.

In the rest of this paper, sample models that have a single level of hierarchy will be considered. This is not meant to represent a limitation of the metrics or of the representation, but rather a simplification that allows an easier presentation of the discussed issues and solutions.

V. Metrics definition

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 dependences induced by synchronization statements.

A. 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 $t \in DataTypes$ 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 service s of an interface $i =< b, d, T >$, with signature identifier d_s we can define the width W as $W(s) = W(i) = W(T)$. Services with signature identifier c_s or e_s have a conventional $W(s) = 0$.

Let $\oplus$ be a polymorphic operator such that $(\mathbb{N}, \oplus)$ is a commutative monoid. Then, for a channel c,

$$W(c) = \bigoplus_{i | \exists e \in E, e = <m,c,p,i>, \forall m,p} W(i)$$

and for a pair of module and channel (m, c), Communication Width can be defined as

$$W(m,c) = \bigoplus_{i | \exists e \in E, e = <m,c,p,i>, \forall p} W(i)$$

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

$$\hat{W}(m_1, m_2, c) = W(m_1, c) \oplus W(m_2, c)$$

Now we can define the communication width between nodes connected through an arbitrary number of channels:

$$C(m_1, m_2) = \{c | \exists e_1 =< m_1, c, p_1, i_1 >, e_1 \in E \wedge \exists e_2 = (m_2, c, p_2, i_2), e_2 \in E\}$$

$$\hat{W}(m_1, m_2) = \bigoplus_{\forall c \in C(m_1, m_2)} \hat{W}(m_1, m_2, c)$$

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 can 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.

B. 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 service signatures, $D(m) = d \in \{r, w, rw, c\}$, where the possible values of $D(m)$ represent a read-only, write-only, read/write or control-only communication. The control-only communication is exemplified by the control operations of the non-blocking interfaces. The read only, write-only and read/write communication are, respectively, the *put*, *get* and *transport* services. This information is immediately available from the MOIR representation, and is the starting point from which the metric can be computed on pairs of modules and channels.

To define D for interfaces and modules, we need a binary operator $\oplus$ over the domain $A = \{r, w, rw, c\}$. We also introduce a unary operator $-$ that will be interpreted as a direction reversal. The semantics of the two operators are described in Table II.

We can now define

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

for an interface $i \in I_{TLM}$, and we can say that the directionality of a port p that requires an interface i is given as $D(p) = -D(i)$. So, for a single component $m \in M$,

$$D(m) = \bigoplus_{p \in P_m} D(p) + \bigoplus_{i \in I_m} D(i)$$

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

$$D(n_i, n_j) = \bigoplus_{e=(n_i, n_j, p_e, i_e) \in E} D(i_e)$$

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 communication, for instance revealing the need for locking policies.

TABLE II
OPERATORS OVER THE A DOMAIN

$\oplus$	r	w	rw	c
r	r	rw	rw	r
w	rw	w	rw	w
rw	rw	rw	rw	rw
c	r	w	rw	c

$-$	
r	w
w	r
rw	rw
c	c

C. Memory Size

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

Let Var_m be the set of all attributes of module m that are not modules themselves. We can define the Memory Size $size(v), v \in Var_m$ as the actual memory occupation for that attribute, as given by the `sizeof` C++ expression.

Then $size(m)$ can be defined as:

$$size(m) = \sum_{v_j in Var_m} size(v_j) + \sum_{m_i \in M_m} size(m_i)$$

In the case of a non-structured module, $M_m = \emptyset$, so 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.

D. Execution Classes

In order to collect information on the number of service invocations and synchronizations performed by a given service or process, we define the *Execution Classes*. This is a kind of intermediate metrics, which will be used to define a set of metrics such as minimum or maximum number of invocations.

The Execution Classes EC of a given service s or process t are defined as follows.

- For the simplest service Communication Control Flow Graph, composed of a single invocation of a service s_1 (or equivalently, of a single synchronization primitive), $EC = \{< s_1, 1 >\}$ (or, in case of wait or notify primitives on an event e_1, $EC = \{< w^{e_1}, 1 >\}$ or $EC = \{< n^{e_1}, 1 >\}$, respectively);
- For a Communication Control Flow Graph including a sequence of service invocations or synchronizations $s_1 \dots s_n$, $EC = \{< s_1, 1 > \cdots < s_n, 1 > \}$;
- For a Communication Control Flow Graph including a sequence of n invocations of the same service or synchronizations on the same event, $s_1 \dots s_1$, $EC = \{< s_1, n >\}$;
- For a Communication Control Flow Graph made of a loop construct (i.e., a cycle in the graph) containing a subgraph g such that $EC_g = \{< s_1, n >\}$, $EG = \{< s_1, [n, n \times k] >\}$ where k is the maximum number of iterations of the loop, $k = \infty$ if there is no known bound;
- For a Communication Control Flow Graph including two or more different paths (after loop reduction), each path is considered as a different execution class.

For example, the Control Flow Graph shown in Figure 2 produces the following execution classes:

$$EC_{s_{i_2}} = \{\{< w^{e_1}, 1 >, < s_{i_3}, 1 >, < n^{e_2}, 1 >\}, \{< s_{i_3}, 1 >, < n^{e_2}, 1 >\}\}$$

indicating that the service considered, depending on the execution class realized during a given invocation, may or may not block itself on event e_1. If s_{i_3} had a non-blocking behavior, this would imply that the invocation of service s_{i_2} would be blocking or non-blocking depending on the control flow. This is the maximum level of information that can be obtained from our static analysis. Profiling could then be used to provide frequency weights for the execution classes.

From the Execution Classes, we can compute an *execution frequency* metrics, which will be useful to gauge the ratios between bandwidths of different connections.

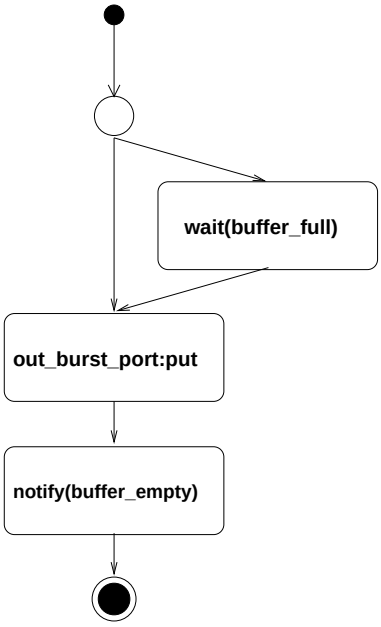

Fig. 2. Example of Communication Control Flow Graph

The Execution Frequency $EC_{s_2}(C, s_1)$ of a service s_1 in service s_2 (or in process t) for an Execution Class C is defined as the value of the second element of the tuple $< s_1, x >$ in that Execution Class of s_2 (or t, respectively).

Then, we can define a Maximum (or Minimum) Execution Frequency by defining:

$$EF_{s_2}^{max}(s_1) = max_C EC_{s_2}(C, s_1)$$

$$EF_{s_2}^{min}(s_1) = min_C EC_{s_2}(C, s_1)$$

E. Blocking Components

Intuitively, Blocking Components are chains of service invocations such that all methods within the chain are blocked until the end of the computation.

We define the blocking property as a relation between two services (or a process and a service) s_1 and s_2 that is conditioned by a set of Execution Classes of s_1. We say that s_1 is blocked waiting for s_2 under the set of Execution Classes $EC_{s_1}^{s_2} \subseteq EC_{s_1}$ if $s_1 \in ec, \forall ec \in EC_{s_1}^{s_2}$.

The definition of blocking property given above is local, that is it characterizes the relation between a service or process, and the services it may invoke. We can extend the definition by considering that, if s_3 is invoked in some Execution Classes $EC_{s_2}^{s_3} \subseteq EC_{s_2}$, then s_1 is blocked waiting for s_3 when the control flows within the execution classes $EC_{s_1}^{s_2}$ and $EC_{s_2}^{s_3}$. So, $blocking(s_1, s_2)$ is true under the condition $EC_{s_1}^{s_2} \wedge EC_{s_2}^{s_3}$.

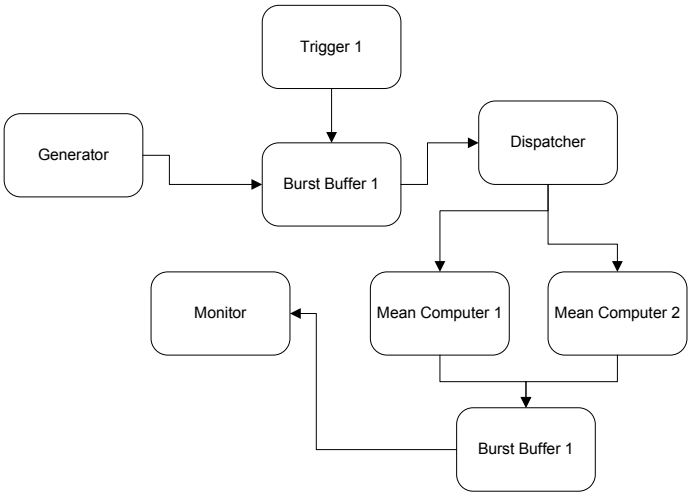

Fig. 3. Structural overview of the system employed to exemplify the application of the metrics

This shows that for the blocking relation the transitive property holds; we can then define *blocking chains* as the transitive closure of the blocking relation.

Finally, *Blocking Components* are formally defined as the chains of modules and channels that implement the services that appear in a blocking chain.

VI. Example of Metrics computation

In order to show how the metrics are applied, and what kind of information the designer can gather, we present a sample TLM system design.

A. System Definition

The overall system implements the computation of the mean of groups of 16 integers from a stream produced by a generator *g1* and stored in a *burst buffer bb1* (a structural diagram of the system in presented in Figure 3).

A *burst buffer* is a component that stores a given number of data tokens, and then outputs them in a single burst when it is properly triggered; this model is particularly interesting since it implements and requires services of different nature (control, data transmission).

The mean may be computed using two different algorithms, implemented by *im1* and *fm1*. A dispatcher *d1* is in charge of the choice between *im1* and *fm1*. Then, the result is sent to a second burst buffer *bb2*, which feeds a monitor *m1*. Two trigger modules (*t1* and *t2*) control the burst buffers.

B. System MOIR Graph

The first step towards the computation of the system design metrics is the parsing of the SystemC model to the intermediate format (MOIR). Figure 4 shows the MOIR graph for the sample system. For the sake of clarity, in addition to the nodes (components) and edges (connections) of the graph, the datatypes of the interfaces involved in each connection have been annotated on the edges of the graph.

C. Metrics Evaluation

For the purpose of this example, we use a simplified notation. Since all interfaces are of the *blocking put* type, we only need to specify the datatype. Most channels offer a single service, so the service will be identified by the component name, except in the case of the burst buffer, where the data type (bool or other) will be used to distinguish different services.

In the end, the set of services implemented is:

$$S = \{M1, BB2 : bool, BB2 : float, \\ IM1, FM1, D1, BB1 : bool, BB1 : int\}$$

The same we can do with processes, so that:

$$T = \{G1, T1, T2\}$$

Within the system we have a set of four events $E = \{e_1, e_2, e_3, e_4\}$:

- e_1 : `buffer_full` for the burst buffer *bb1*
- e_2 : `buffer_emptied` for the burst buffer *bb1*
- e_3 : `buffer_full` for the burst buffer *bb2*
- e_4 : `buffer_emptied` for the burst buffer *bb2*

We will therefore indicate the notify and wait on event e_i with the symbols n^i and w^i.

1) Execution Classes: The service provided by the monitor *m1* has a single, empty execution class:

$$EC_{M1} = \{\emptyset\}$$

For the burst buffer *bb2*, we have the following classes:

$$EC_{BB2:float} = \{\{< n^3, 1 >, < w^4, 1 >\}, \emptyset\}$$

$$EC_{BB2:bool} = \{\{< w^4, 1 >, < M1, 1 >, < n^3, 1 >\}, \\ \{< M1, 1 >, < n^3, 1 >\}\}$$

The two mean computation channels offer services that have the same execution classes:

$$EC_{IM1} = EC_{FM1} = \{\{< BB2 : float, 1 >\}\}$$

The dispatcher *d1* has the following execution classes:

$$EC_{D1} = \{\{< IM1, 1 >\}, \{< FM1, 1 >\}\}$$

For the burst buffer *bb1*, we have the following classes:

$$EC_{BB1:int} = \{\{< n^1, 1 >, < w^2, 1 >\}, \emptyset\}$$

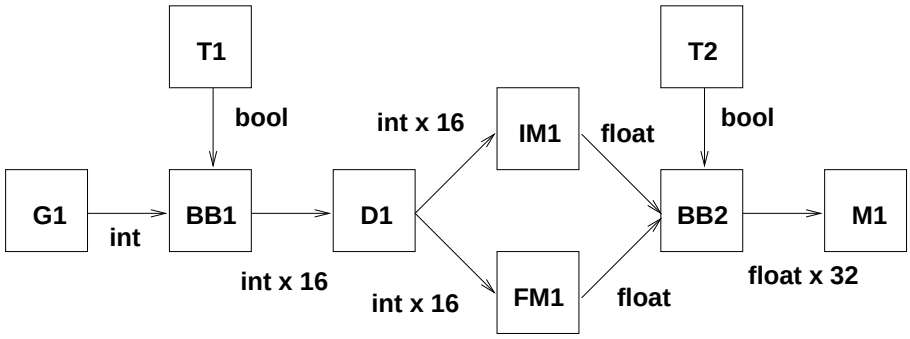

Fig. 4. System MOIR Graph with indication of the interface datatypes

$$EC_{BB1:bool} = \{\{< w^2, 1 >, < D1, 1 >, < n^1, 1 >\}, \{< D1, 1 >, < n^1, 1 >\}\}$$

The three processes are characterized by the following execution classes:

$$EC_{G1} = \{\{< BB1 : int, 1 >\}\}$$
$$EC_{T1} = \{\{< BB1 : bool, 1 >\}\}$$
$$EC_{T2} = \{\{< BB2 : bool, 1 >\}\}$$

2) Blocking Components: By indicating with ec_s^i the i-th execution class of service $s \in S$, we have that the true blocking function on s, both in the flow-insensitive and flow-sensitive version is as shown in Table III.

TABLE III
TRUE BLOCKING FUNCTION VALUES

Service	True Blocking	FS True Blocking
M1	False	False
BB2:bool	True	$ec^1_{BB2:bool}$
BB2:float	True	$ec^1_{BB2:float}$
IM1	True	$ec^1_{BB2:float}$
FM1	True	$ec^1_{BB2:float}$
D1	True	$ec^1_{BB2:float}$
BB1:bool	True	$ec^1_{BB2:float} \vee ec^1_{BB1:bool}$
BB1:int	True	$ec^1_{BB1:int}$

Then, the blocking components are computed as follows:

$$BC_{G1} = \begin{cases} \emptyset & ec^2_{BB1:int} \\ \{BB1\} & ec^1_{BB1:int} \wedge ec^2_{BB2:float} \\ \{BB1, D1, IM1, BB2\} & ec^1_{BB1:int} \wedge ec^1_{BB2:float} \\ \{BB1, D1, FM1, BB2\} & ec^1_{BB1:int} \wedge ec^1_{BB2:float} \end{cases}$$

$$BC_{T1} = \begin{cases} \emptyset & ec^2_{BB1:bool} \wedge ec^2_{BB2:float} \\ \{BB1\} & ec^2_{BB2:float} \wedge ec^1_{BB1:bool} \\ \{BB1, D1, IM1, BB2\} & ec^1_{BB2:float} \wedge ec^1_{BB1:bool} \\ \{BB1, D1, FM1, BB2\} & ec^1_{BB2:float} \wedge ec^1_{BB1:bool} \end{cases}$$

$$BC_{T2} = \begin{cases} \emptyset & ec^2_{BB2:bool} \\ \{BB2\} & ec^1_{BB2:bool} \end{cases}$$

3) Memory Size: Table IV shows the memory size metric computed for the components present in the system.

TABLE IV
MEMORY SIZE METRIC COMPUTATION

Component	Size
M1	none
BB2	1056 bytes
IM1	none
FM1	none
D1	512 bytes
BB1	544 bytes
G1	none
T1	none
T2	none

VII. POTENTIAL APPLICATIONS

The metrics so far defined and computed can be exploited to infer other information useful to perform different implementation choices, such as the measurement of the communication components (e.g. bus and fifo width), bandwidth measurement and allocation, assignment of communication functions to shared resources

(such as bus and shared memories). In this section, the application of such metrics has been formalized and exemplified.

A. Communication Channels Width

A direct application of the communication width is the estimation of the width of the buses or fifos needed to implement a given communication service.

The communication width metric is itself parametric with respect to the operator used to combine the fine grain (method-level) data. Different choices of the $\oplus$ operator can be used for different purposes. Let us review the most significant options, considering the effect on the communication width metric computed on two modules connected by a channel.

If $\oplus$ is max, then, given a fixed serial bandwidth BW_{serial} and a number of wires n_{wires}, the designer imposes an upper bound to the transmission delay for the parameters of the single service invocation. That is, the following inequality is imposed:

$$\underset{s \in S}{max}(t_s) \leq \frac{CW_{max}}{BW_{serial} \times n_{wires}}$$

where CW_{max} is the communication width metric, using the maximum operator as $\oplus$, t_s is the communication delay for the parameters of service s, and S is the set of all services considered.

If, on the other hand, the average operator is used instead of $\oplus$, then the designer is imposing an average bandwidth. If $\overline{t_s}$ is the average communication delay for a service in S, then

$$\overline{t_s} = \frac{CW_{avg}}{BW_{serial} \times n_{wires}}$$

By weighting the communication widths of each service by weights $p(s), s \in S$, it is also possible to take into account the distribution of service invocations – either computed by means of simulation or derived from the Execution Classes. In this case,

$$\overline{t_s} = \frac{CW_{avg}}{BW_{serial} \times n_{wires}} = \frac{\sum_{s \in S} p(s) \times CW(s)}{BW_{serial} \times n_{wires}}$$

Last, if the $\oplus$ operator is replaced with the arithmetic sum, the designer is imposing an upper bound to the transmission delay, assuming that threads within the initiator module can fire different methods offered by the channel interface. In this case, the upper bound is defined by the following inequality:

$$\sum_{s \in S} t_s \leq \frac{CW_{+}}{BW_{serial} \times n_{wires}}$$

where CW_{+} is the communication width metric, using the sum operator as $\oplus$, while t_s is the communication delay for the parameters of service s and S is the set of all services considered.

B. Bandwidth Constraints Propagation

An application of the combined information obtained by blocking chains, execution frequency and communication width is the estimation of *Bandwidth Constraints Propagation*.

Suppose, in the simplest case, that a module invokes a service s_1 on a given channel. If the communication width of s_1 is w_1, and s_1 invokes another blocking service s_2, of communication width w_2, it can be inferred that: for every amount w_1 of data that is sent through the communication link that implements s_1, at least an amount w_2 of data will have to be sent through the link on which is implemented s_2 (we will refer to this link with l_2). If, for some reason, the bandwidth of l_2 is limited, this will be reflected by the maximum data rate transmission of s_1. In particular, if the bandwidth on l_2 is limited to bw_2, the data rate of s_1 will not be able to go over:

$$bw_1 \leq \frac{w_1}{w_2} \times bw_2$$

.

With the combined use of the aforementioned metrics, it is possible to generalize this reasoning to the complex cases of:

- services that can be either blocking or non–blocking;
- bandwidth ratios between services that are implemented by channels not directly connected (i.e., that are “distant” in the system structure).

Since bandwidth is, in the general case, data dependent, information that can be obtained is, most likely, upper and lower bounds to the bandwidth ratio of two different connections.

Let us consider the application of this principle to the modeling example proposed. If we consider the blocking component BC_{T1}, we observe that all the execution classes for which $BB2$ is invoked are present in the condition list of all the blocking chains that contain the execution classes for which the $D1$ is invoked. From this information, we can infer that every time $D1$ is invoked, $BB2$ is invoked. Thus, if *all* the connections to $BB2$ are limited in bandwidth, the limit is propagated to $D1$. Since $BB2$ can be invoked by either $IM1$ and $IF1$, the bandwidth constraint on $D1$, when $IM1$ and $IF1$ are limited to BW_{IM1} and BW_{IF1} will be:

$$\frac{BW_{FM1}}{W_{FM1}} + \frac{BW_{IM1}}{W_{IM1}} \geq \frac{BW_{BB1}}{W_{BB1}}$$

To experimentally assess this relation, we simulated the model imposing fixed communication latencies on both $FM1$ and $IM1$, and measured the average bandwidth with which $D1$ was invoked by $BB1$. Since $W_{IM1} = W_{FM1} = 1$ and $W_{BB1} = 16$ (width is measured in words, and both integers and floating point are encoded with a single word):

$$BW_{FM1} + BM_{IM1} \geq \frac{BW_{BB1}}{16}$$

The results of the simulation are shown in Table V, where the relationship between bandwidths of the considered model connections is presented. Columns correspond to simulations in which a bandwidth constraint was imposed to a particular connection. It is possible to observe that the bandwidth propagation relation holds for all configurations.

C. Communication Resource Sharing

Another possibility offered by blocking components is to characterize sets of communication services that can be implemented with a shared resource, for example a bus or a shared memory, with a minimum impact on the bandwidth.

Let us consider two services, s_1 and s_2, and assume that there is a blocking component bc such that $s_1 \in bc$ and $s_2 \in bc$. This means that all the data transfers caused by the execution of s_1 and s_2 will happen sequentially, without any overlap (not considering pipelining, of course). If s_1 and s_2 data transfers are assigned to the same communication resource, they will likely cause negligible access conflicts. Again, such situations can be directly detected on simple models, but their location becomes rapidly unfeasible as the number of channels and services grows.

D. Pipelining

In addition to the detection of potential shared resource, pipelining opportunities can also be investigated by means of the blocking components. If the blocking component chain is unidirectional – that is, data flow through components of the chain in a single direction, then an opportunity for pipelining arises. This can be detected by computing the directionality metric on the services of the blocking chain. If BC is a blocking chain, and s_i, s_j are services within BC, then we can say that the portion of BC between s_i and s_j is a pipelining candidate if

$$\forall k, i < k < j, D(s_k, s_{k+1}) = D(s_i, s_{i+1})$$

Of course, the opportunity of pipelining for performance improvements conflicts with the potential benefits of resource sharing for the reduction of resource requirements, so a tradeoff must be considered. The bandwidth bounds discussed in Section VII-A can be applied to estimate the tradeoff point.

In the example proposed, we can observe that, for every execution class, $DD1$ and $BB1$ always belong to the same blocking component. This means that implementing connections to $DD1$ and $BB2$ with a shared connection should not cause a significant slowdown. In order to verify this hypothesis, we simulated the effect of shared implementation of connections. We considered all the possible couples, and simulated resource sharing among them. The overall slowdown was computed as the ratio between non shared implementation average throughput and the current shared implementation throughput. Results are shown in Table VI. Sharing configurations with slowdown of 1.0 (no slowdown) are exactly those foreseen by means of the metrics.

VIII. Industrial Application Example

The analysis previously described has been implemented in an automatic tool. The tool has parsing and internal model representation capabilities, and provides a general framework for the analysis of high-level C-based models. In order to prove the effectiveness of the analysis proposed in a realistic context, we applied the information extracted to the design of a module that is part of a telecommunication application developed at Nokia Siemens Network [10]. The size of the model necessarily required automatic computation of the metrics, analysis “by hand” being too complex.

The high-level model considered represents the subsystem of a base station that implements the ATM over IP service. It is composed of 16 modules, connected by 22 connections (see Figure 5). The whole code is more than 6000 lines long. All communications are modeled as blocking Transaction Level Model service invocations. The implementation problem considered is the optimization of the maximum achievable throughput, using the minimum possible set of communication resources. In order to do so, we performed a *non-overlapping* analysis, to maximize resource sharing avoiding access conflicts.

The original problem space can be put into correspondence with the set of all the possible partitions of all the 22 connections contained in the model, making an exhaustive exploration clearly unfeasible.

The analysis tool implements *non-overlapping* analysis, automatically computing a representation of the non-overlapping relation for every couple of connections.

TABLE V

RELATION BETWEEN BANDWIDTHS OBTAINED BY SIMULATION

	none	bb1_d1	im1_bb2	bb2_m1	fm1_bb2
bb1_d1	7.58	2.05	6.64	2.54	1.33
im1_bb2	.20	.06	.18	.07	.04
bb2_m1	.45	.12	.40	.15	.08
fm1_bb2	.28	.08	.25	.10	.05

TABLE VI

SLOWDOWN DUE TO COMMUNICATION RESOURCE SHARING

	bb1_d1	im1_bb2	bb2_m1	fm1_bb2
g1_bb1	1.02	1.02	1.50	1.02
bb1_d1		1.00	1.52	1.00
im1_bb2			1.19	1.00
bb2_m1				1.32

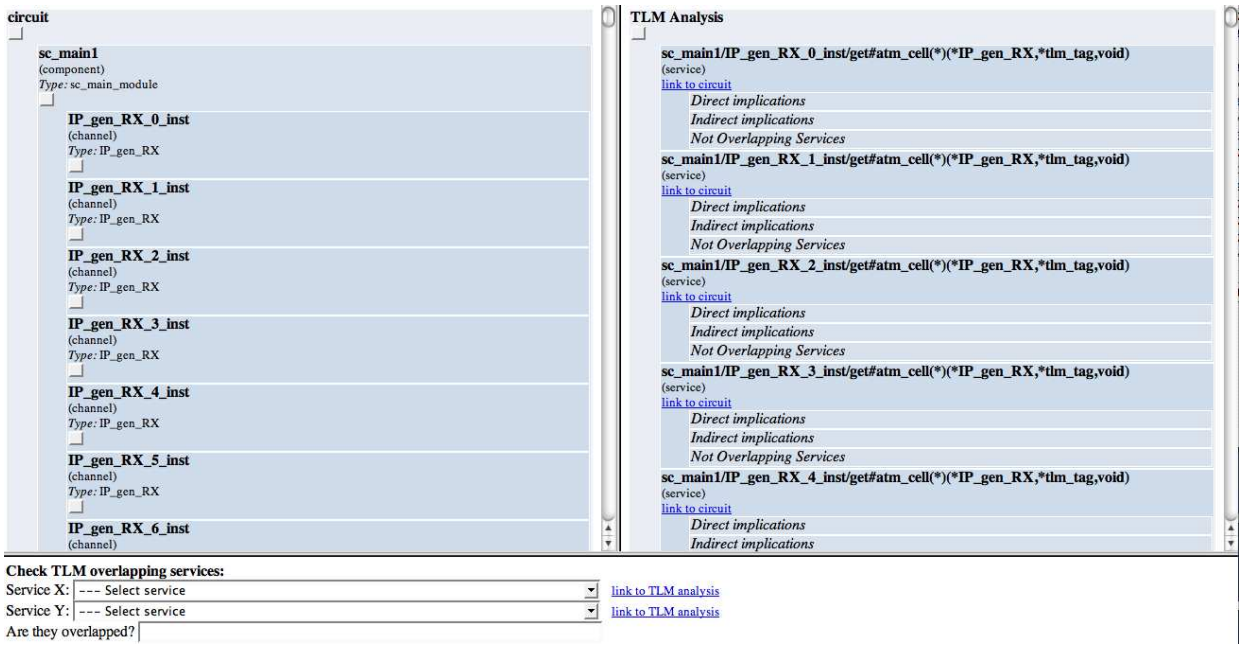

Fig. 6. Hypertext-based navigable front-end to the structural metrics as extracted by the computation tool

On a Dual Core 2Ghz Pentium, the analysis took 344 seconds to be performed.

The information produced by the analysis was post-processed and converted in a navigable html format (see Figure 6).

From this information, classes of maximum size of reciprocally non-overlapping connections are derived as maximal cliques of the non-overlapping relation graph.

The static analysis of the model highlighted a partition of the communications into five non-conflicting classes. Each class can then be mapped onto a single communication resource, without causing any access conflict, and thus avoiding any bandwidth degradation.

In Figure 7 the communication implementation structure is represented as a set of interconnected shared resources.

Moreover, the directionality metrics suggests that there are candidates for pipeline implementation (see Section VII-D). In the present case, the non-overlapping class comprehending communication between the components *OAM_DEMUX*, *DEMUX*, the set of *A2IP* can be implemented as a set of interstage pipeline buffers, since all its communications actions happen in the same direction.

IX. CONCLUDING REMARKS

We presented a framework for the automatic analysis of system models described at transaction level. This

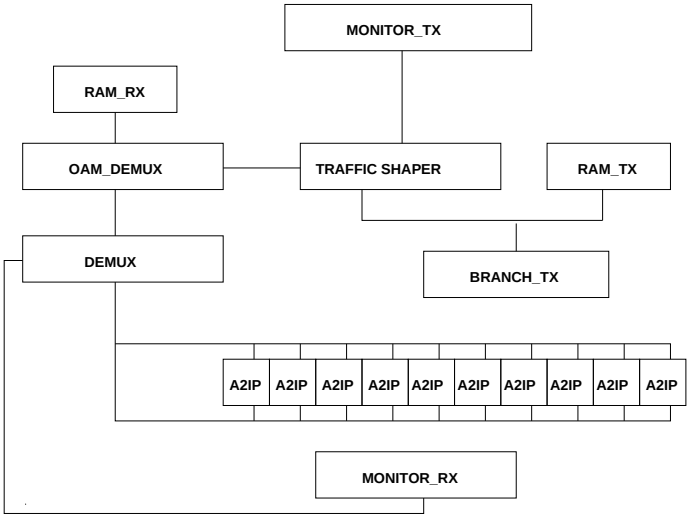

Fig. 5. Structural representation of the system under analysis

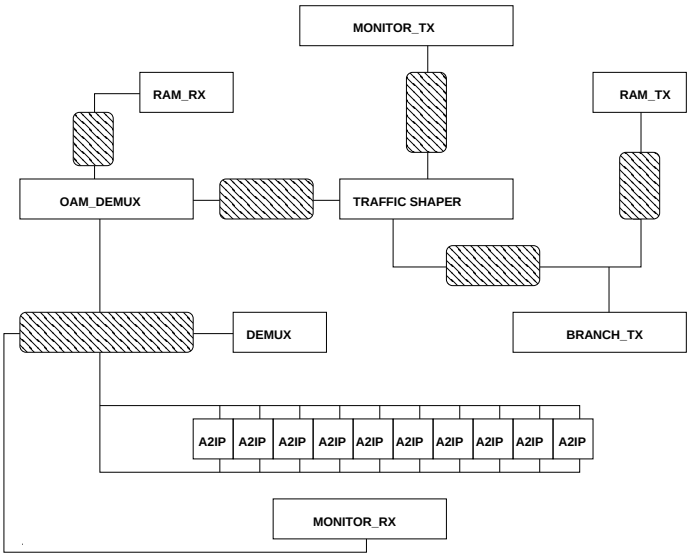

Fig. 7. Implementation of the system communication with five shared resources

level of abstraction is effective in modeling early executable descriptions of systems under design, as well as first refinement phases. Thus, it is particularly suitable as front–end language for the design of hw/sw systems, and is producing increasing interest, in the EDA field, towards methodologies and tools that allow to best exploit the information contained in such models.

We take under consideration the SystemC transaction level of abstraction modeling, as defined by the OSCI steering consortium. Structural and behavioral features of models were abstracted and mathematically formalized. Relying on the formalization, a set of metrics for the estimation of the communication design choices effects was defined, and their computation exemplified. Different design application scenarios were drawn, and the techniques proposed were applied to the example proposed. Static analysis techniques can be particularly valuable for large designs. In these designs, structural properties and features cannot be detected manually or by exhaustive simulation as they would be in simpler design cases. On the other hand, the proposed methodology allows these features to be highlighted automatically, without need for time-consuming simulation.

REFERENCES

[1] SystemC Website. http://www.systemc.org.

[2] Giovanni Agosta, Francesco Bruschi, and Donatella Sciuto. Static analysis of transaction-level models. In *DAC '03: Proceedings of the 40th conference on Design automation*, pages 448–453, New York, NY, USA, 2003. ACM Press.

[3] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SytemC*. Kluwer Academic Publishers, 2002.

[4] Christopher K. Lennard, Patrick Schaumont, Gjalt de Jong, Anssi Haverinen, and Pete Hardee. Standards for system-level design: practical reality or solution in search of a question? In *Proceedings of the conference on Design, automation and test in Europe*, pages 576–585. ACM Press, 2000.

[5] Y. Le Moullec, N. Ben Amor, J-Ph. Diguet, M. Abid, and J-L. Philippe. Multi-granularity metrics for the era of strongly personalized socs. In *DATE '03: Proceedings of the conference on Design, Automation and Test in Europe*, page 10674, Washington, DC, USA, 2003. IEEE Computer Society.

[6] Adam Rose, Stuart Swan, John Pierce, and Jean-Michel Fernandez. Transaction level modeling in systemc. http://www.systemc.org.

[7] F. Salice, L. Del Vecchio, L. Pomante, and W. Fornaciari. Partitioning of embedded applications onto heterogeneous multiprocessor architectures. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 661–665, New York, NY, USA, 2003. ACM Press.

[8] Donatella Sciuto, Fabio Salice, Luigi Pomante, and William Fornaciari. Metrics for design space exploration of heterogeneous multiprocessor embedded systems. In *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign*, pages 55–60, New York, NY, USA, 2002. ACM Press.

[9] A. Siebenborn, O. Bringmann, and W. Rosenstiel. Worst-case performance analysis of parallel, communicating software processes. In *Proceedings of the Tenth International Workshop on Hardware/Software Codesign*, pages 37–42, 2002.

[10] Francesca Tonetta. Design example 3, implementation based on the icodes tools and methodology. Deliverable ICODES/SIEMENS/R/D38/1.0, IST-04452 ICODES Project, 2007.

[11] F. Vahid and D. Gajski. Closeness metrics for system-level functional partitioning. In *Proceedings of the European Design Automation Conference (EuroDAC)*, 1995.

[12] Sungjoo Yoo, Gabriela Nicolescu, Damien Lyonnard, Amer Baghdadi, and Ahmed A. Jerraya. A generic wrapper architecture for multi-processor soc cosimulation and design. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 195–200. ACM Press, 2001.

[13] Jianwen Zhu, Daniel D. Gajski, and Rainer Doemer. Syntax and semantics of the spec C+ language. In *Proceedings of the SASIMI Workshop*, pages 75–82, 1997.