



Public

	IST-004452	ICODES
Interface- and Communication-based Design of Embedded Systems		
Project Duration	2004-08-01 - 2007-07-31	Type STREP

	WP no.	Result no.	Lead participant
	WP1	D25	Polimi
Final Metrics Evaluation Definition			
Prepared by	Francesco Bruschi, Fabrizio Ferrandi, Donatella Sciuto		
Issued by	POLIMI		
Document Number	ICODES/POLIMI/P/D25		
Classification	ICODES Public		
Submission Date	2006-08-01		
Due Date	2006-07-31		
Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)			

© Copyright 2006 Bosch, ECSI, Prosilog, OFFIS, Siemens, Politecnico di Milano, Thales Communications

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

Contents

1	Introduction	2
2	Parsification and representation framework	3
3	Communication Implications	7
3.1	Communication Implications Definition	8
3.2	Direct and Indirect Implications Computation	9
3.3	System Representation and State	9
3.4	Service Analysis	10
4	Application Example	12
4.1	System Behavior	12
4.2	System Analysis	13
4.3	Non-overlap	16
4.4	Bandwidth Constraint Propagation	16
5	Conclusions	18
A	Application Example Code	19

Section 1

Introduction

This deliverable is to be considered as a refinement and completion of the deliverable D9 [2]. In deliverable D9, in fact, we proposed a mathematical formalization of a SystemC Transaction Level system model. Upon that formal representation, a set of metrics, focused on the aims of the analysis tool to be implemented, were formally defined. Since the definitions were *ahead* of the actual metrics implementation, this deliverable was planned to integrate the information provided in D9 under different points of view:

- a software framework for the parsification and management of SystemC model representations at different levels of abstractions was developed at Politecnico di Milano. Since implementation of analysis is dependent on the underlying model management technology, and since we believe that this is of scientific interest in the context of ICODES, the structure and main features of such framework are described in section 2;
- during tool implementation, different examples were tested as benchmarks. These tests, other than allowing the debugging of the implementation, induced the refinement of some of the metrics already defined, and suggested the definition of new analysis procedures. In particular, *communication implications* and *chains* were formalized and implemented, based on a simplification of *blocking components*. In this deliverable, these refinements are presented in section 3. Moreover, techniques adopted to implement these metrics are of scientific interest within the project context. The algorithms employed to extract information on communication chains are described and motivated, still in section 3;
- in order to clarify potential applications of the new defined metrics, an example is proposed. Both analysis and information exploitation are presented in section 4.

Section 2

Parsification and representation framework

In this section we introduce *Panda*, the technological framework we developed to perform parsification and management of SystemC Transaction Level Models. Even though TLM is the main target for the Analysis defined within ICODES, the framework is able to handle other levels of abstraction as well. Since `SystemC` is an extension of the C++ programming language, it was decided to use an existing C++ front-end and after a deep analysis of the various existing C++ compilers or `SystemC` analyzers it was decided to adopt the front-end capabilities of the GNU GCC compiler [1]. This choice is mainly due to the full use in `SystemC` of the C++ features, such as templates, inheritance, polymorphism, etc., and to the need of behavioral information extraction and analysis. Moreover, GCC provides an efficient, extensible and well maintained C++ open source compiler.

From the version 3.5/4.0 the front-ends parse the source language producing GENERIC trees, which are then turned into GIMPLE [5] (see Figure 2.1).

The first intermediate representation, GENERIC, is a common, language independent representation, that serves as an interface between the parser and the optimizer. GIMPLE is also a language independent, tree based representation of the source specification but it is more suited for target and language independent optimizations (e.g., inlining, constant propagation, tail call elimination, redundancy elimination, etc). With respect to GENERIC, GIMPLE is more constraining, since:

- its expressions have no more than three operands;
- there are no control flow structures, except conditional jumps;
- expressions with side-effects are only allowed on the right hand side of the assignments.

Although GIMPLE has no control flow structures, GCC builds the control flow graph (CFG) anyway, to perform language independent optimizations. During our tool analysis, we evaluated that the information provided were adequate to perform

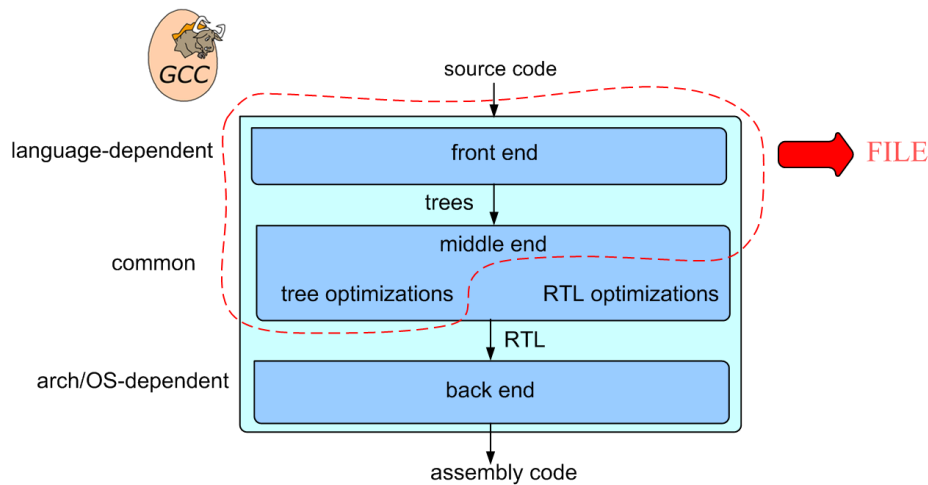


Figure 2.1: GCC internal structure.

static analysis of `SystemC` code. An important feature of the `Panda` framework is that manual modification of the `SystemC` specification is not required since all the information needed during the static analysis is present into the `GIMPLE` intermediate representation.

In order to keep the framework structure modular, we did not directly integrate `Panda` in `GCC`. Rather, we exploited `GCC` debugging features to obtain a the `GIMPLE` data structure as an ASCII file.

Actually, the dump performed with this option is performed on a *per function* basis, and this leads to several `GIMPLE` tree nodes duplications, unnecessary for our purposes. To avoid this problem we slightly modified the tree dump functions of `GCC` by removing some duplications and simplifying the format of the ASCII file.

Following the grammar of these files we built a parser able to build a manageable data structure within the `Panda` framework, thus allowing an independent analysis of the `GCC` data structures. Obviously, the extraction of `GIMPLE` information from `GCC` introduces some overhead, but at the advantage of a modular decoupling between the `GCC` compiler and our toolset. The `GCC` analysis and the `GIMPLE` parsing corresponds to the first two steps performed by the `Panda` framework to analyze the `SystemC` specification (see Figure 2.2).

The next step, `Graphs` and `Structural info` extraction, builds a layer of functions and data structures providing:

- the Control Flow Graph of each function present in the specification;
- the hierarchy of `SC_MODULES` or `SC_CHANNELS` and their connections binding identified during the analysis of module constructors hierarchically instantiated from the `sc_main`;
- the class layout in terms of:

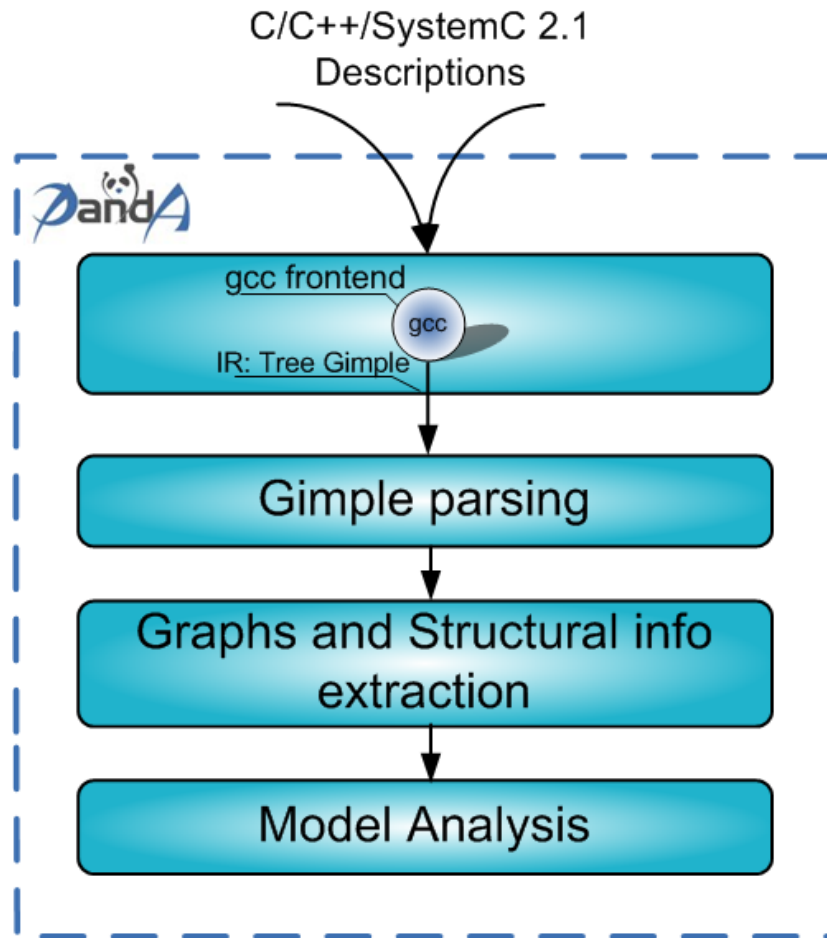


Figure 2.2: PandA metric analysis flow.

- ports,
- events
- TLM services
- generic data structures

The Control Flow Graph (CFG), isomorphic to that built by GCC, represents the sequencing of the operations as described in the language specification. Each CFG node is characterized by:

- an identifier
- the list of variables read and written
- a reference to the corresponding GIMPLE node

Calls to functions are moreover associated with the control flow graph of the called function, if this is available in the specification. Given this information a data dependency analysis is performed to identify correlation between variable uses and definitions. Currently, this analysis is approximate since the pointers alias analysis of GCC is not exploited. With the `SystemC` RTL level of abstraction this is not a problem since it does not require an exact alias analysis to identify the name of the instances, while considering TLM `SystemC` this can be a problem in case pointers can be heavily used. Up to now, the design benchmarks set considered (i.e., benchmarks provided by ICODES industrial partners and the `SystemC` 2.1 testsuite) did not require this analysis.

In addition to control and data flow graph, other graphs have also been analyzed, such as the system dependency graph [4] (SDG). In particular, in `Panda` it has been integrated a high-level synthesizer which exploits the intrinsic parallelism of this intermediate representation [3].

All graphs are managed by a `graph_manager` while the structural information is collected in a `structural_manager`. The `structural_manager` also integrates references to the behavioral information and vice versa. In fact, each module is associated with one or more processes, each one described by means of a `graph_manager`. The same holds for channel services. Moreover, each CFG node is annotated with all the `SystemC` information related to its associated operation. For instance, in case a CFG node corresponds to a wait statement, it is possible to obtain the list of events combined in *and* or in *or*. On the other hand, if a CFG node corresponds to a notify statement it is possible to obtain the associated event, while if the node corresponds to a service call it is possible to obtain the associated port. The cross annotation between structural and behavioral information allows an easy traversal of `SystemC` specifications. Furthermore, to support the Analysis, some functions able to compute the size of `SystemC` objects have also been integrated. This analysis partially exploits the information obtained through the value range analysis computed by the GCC compiler and the peculiarity of the `SystemC` language. An object of type `sc_int<8>`, for instance, has a size of eight bits, not the `sizeof` returned by GCC and stored in the GIMPLE intermediate representation.

Section 3

Communication Implications

The main application of the `PandA` framework to the purposes of the ICODES project is the analysis of Transaction Level functional Models (last step of figure 2.2). In this section we present the analysis that was added to the set defined in [2], relying on the *PandA* framework described in the previous section. In particular, the analysis defined allows the detection of some relationships among *communication actions*, that in turn can be exploited during the design to perform:

- resource allocation
- resource sharing
- performance evaluation

As already stated, Transaction Level Modeling was introduced in SystemC since the language version 2.0. The main language feature introduced, namely Interface Method Calling (IMC), offered the possibility of describing communication between system components with a semantic similar to that of Remote Procedure Calling. This form of communication modeling is so powerful that previous language elements such as signals are now described by means of service invocations. The wide expressive possibilities of IMC represented, on the other hand, a limit to the definition of precise and agreed communication representation standards, that in turn made it difficult to apply model exchanging practices and automatic analysis. In order to solve this problem, OSCI defined a library of interfaces, that define Transaction Level more precisely. These interfaces offer a restricted set of methods, both with blocking and non-blocking behavior. In order to comply with this standardization effort, we decided to consider, as input of the analysis procedure, models that are compliant to this definition of transaction level. To briefly recall the set of communication functionalities offered by the TLM interface library, we consider three kinds of services:

- *put* methods: they represent communication actions outgoing from a module;
- *get* methods: they represent data request actions;

- *transport* methods: they represent requests of computation, where a parameter is provided and a value is returned.

TLM offers these methods both with blocking and non-blocking behavior. Since our analysis is aimed at the extraction of model features prior to any implementation choice, and since non-blocking semantics is often introduced as a refinement towards implementation, we suggested, as a modeling guideline, the use of blocking versions only. The advantages of this constraint will be clearer with the description of the analysis and of its application.

3.1 Communication Implications Definition

The main concept the analysis relies on is the *communication implications*: when a given service S , implemented by a certain module instance M , is invoked, this can in turn trigger invocation of other services throughout the system. We call the transitive closure classes of implications, with a given service s at the top, *communication implications chain* of s . The ability to automatically detect these *chains* can be exploited, in the design context, in different ways.

An interesting observation is that the rates at which data is communicated in elements of an implication chain are related. In fact, if every invocation of a service $S1$ implies the invocation of another service $S2$, the time average of the number of $S1$ invocations will be less than that of $S2$. Thus, if the bandwidth at which parameters of $S2$ is limited, this will in turn limit the average data rate at which parameters can be transmitted for $S1$. This allows to establish a set of *bandwidth constraints* relationships among connections that implement services invocations in the system.

Another possibility offered by the analysis of implication chains is to establish potential concurrency between function calls. More specifically, it is interesting to detect when two services $S1$ and $S2$ are in a relationship of:

direct implication when $S1$ directly invokes $S2$

extended implication when, every time $S1$ is invoked, $S2$ is invoked afterward. The *direct* implication is a special case of *extended implication*

If the system model is described, as recommended, using blocking calls, and if the parallelism present is explicitly described properly using language synchronization elements (waiting and notification of events), analysis of communication chains makes it possible to detect which services could be potentially invoked simultaneously, and which are always non overlapping in time. An interesting application of this information is the *resource sharing* problem: services whose invocations are always separate in time can be implemented on shared resources, without leading to conflicts.

As will be made clearer in the next section with an example, *communication implications* information is computed relying on the description of system processes as *Communication Control Flow Graphs* (CCFG). As stated in [2], CCFGs are basically subsets of the control flow graphs of the services and of the methods of the system,

composed only of nodes that describe communication relevant actions, and are obtained from processes and services *Control Flow Graphs* by collapsing all nodes that do not represent one of the following actions:

- a service invocation;
- a wait suspension;
- an event triggering.

Given a service S , in order to correctly evaluate its implications, it is necessary to take into account, while examining its control flow graph, the presence of other processes, within the same module, that could be activated. When the service is invoked, every process that has wait statements nodes could be holding on any of them. The set of all the wait nodes in CCFGs of the module to which the service belong is evaluated. When the traversal algorithm encounters a notify node, it checks if there is any other process possibly being triggered. If so, the algorithm recursively begins traversing the activated process CCFG. When such a notification occurs, it must be taken into account that parallel execution paths are generated: at least the one of the triggering process, and that of the awoken one. The traversal algorithm keeps track of the sequence with which nodes are visited, and separates nodes that are visited in parallel. This way, when the visit is over, collection of the nodes sequences visited by all the recursive traversal instances provide information that can be exploited to evaluate direct and indirect implications. A precise algorithmic description of this Analysis is provided in next section.

3.2 Direct and Indirect Implications Computation

In this section, we present the a description of the algorithmic implementation of the analysis previously described. The main algorithm we present is that for the computation of the direct and indirect implication relationships. We will focus, in particular, on the problem of obtaining all the implications of a given a service invocation. The overall system implication relationship can be deduced from the single services information

3.3 System Representation and State

The algorithm that computes the implication of a given service invocation receives, as input:

- the system representation;
- the information on the service to be analyzed s ;

In order to achieve its aim, the algorithm performs a traversal of the Communication Control Flow Graph of the service s and of the other services implicated. The traversal

is directed in such a way that, at every visiting step, all the information on all the possible execution paths is collected. To do so, the algorithm must rely on a representation of the system state that allows to handle the activation of processes through the event signaling mechanism. The state information that is necessary to this aim is:

- the waiting state of every process and service: processes and services can be, in every moment, either executing or waiting for activation. In the SystemC semantics, all the operations that happen between two suspensions in a row are considered to happen in atomic time. From this follows that, in every moment, the algorithm can properly represent the system execution information by keeping track of the waiting statement where each is supposed to be suspended. Formally, this information is represented as a set of waiting statements among those present in all the processes of a given module;
- the notification state of joint wait statements: a process can suspend itself until notification of a *set of events*. The execution semantics is that the process is activated when all the events in the set are notified. This allows to perform complex interprocess synchronization, as it will be shown in the example of section 4. To properly handle this semantics, the algorithm must keep track of the events already notified in a waiting set. Formally, this information is represented as a function that, given a waiting statement CCFG node, returns the subset of events of that statement that still need to be notified to obtain the activation.

3.4 Service Analysis

The algorithm *service Analysis* takes as input the system description and the service to be analysed, s , and produces, as output:

- the set of services directly implicated by s ;
- for every service si directly implicated by s , the set of predecessors, that is the set of services that indirectly implicate si

The functions exploited in the algorithm are:

- $WS(m)$: this function scans the module m , and returns all the waiting statements contained in it;
- $first(s)$: this function returns the entry CCFG node of the service s ;
- $visit(n, seq, EN)$: this function launches a CCFG visit, starting from node n . If, while traversing the CCFG, a call to a service is encountered, this is added to seq . This way, all direct implications are tracked. Moreover, if an event notification is reached, the EN set is updated accordingly.
- $satisfied(w, EN)$: returns *true* if, according to the event notification status set EN , the activation condition of the waiting statement w is satisfied.

Algorithm 1 Produces direct implications of s

```

 $W \leftarrow WS(m)$  {initialize set  $W$  with all the waiting statements in module}
 $EN \leftarrow false$  {initialize set Event Notification to false}
 $SSEQ \leftarrow s$  {initialize set  $SSEQ$  with this service}
 $delta\_SEQ \leftarrow$ 
 $s\_first \leftarrow first(s)$  {get first CCFG node of  $s$ }
 $visit(s\_first, SSEQ, EN)$  {launch a visit of CCFG of service  $s$ }
 $any\_waking \leftarrow true$ 
while  $any\_waking = true$  do
  for all  $w \in W$  do
    if  $satisfied(w, EN)$  then
       $any\_waking \leftarrow true$ 
       $SUB\_SSEQ \leftarrow SSEQ$ 
       $visit(w, SUB\_SSEQ, EN)$ 
       $delta\_SEQ \leftarrow delta\_SEQ \cup SUB\_SSEQ$ 
    end if
  end for
   $SSEQ \leftarrow delta\_SEQ$ 
end while

```

The main goal of the algorithm is to implement a sort of static delta cycle simulation, in which all the possible execution paths are visited. At the end of the algorithm, the set $SSEQ$ contains all direct implications of s .

Note that, whenever during a CCFG visit, invocation of a service s is encountered, the $SSEQ$ set in that moment contains a subset of the methods that indirectly implicate s . These sets are collected by the $visit$ function, and are then combined to obtain the indirect implication information.

Section 4

Application Example

In this section an example of analysis and application of the information gathered is presented.

The example system represents an audio stream converter, and is composed of five different modules:

- a *stereo packet generator*;
- a *stereo packet converter*;
- a *packet sum extraction unit*;
- a *packet difference extraction unit*;
- a *speaker*.

The code of the example is reported in Appendix A.

The different components are connected as in figure 4.1.

4.1 System Behavior

The system behavior is as follows:

1. the generator produces packets that contain information that encodes two different audio channels;
2. packets are sent to the converter (via a *put* method invocation);
3. the converter sends a copy of the audio packets to the sum and difference units; we imagined that the designer is aware of the potential parallelism in the requests to the functional units, and thus models requests that are simultaneous to the functional units. The simplest way to do this is to model two processes, within the module, that are place on hold for the notification of a given event *s*. When the *put* that transmit the packet to the converter is invoked, it notifies *s*, and

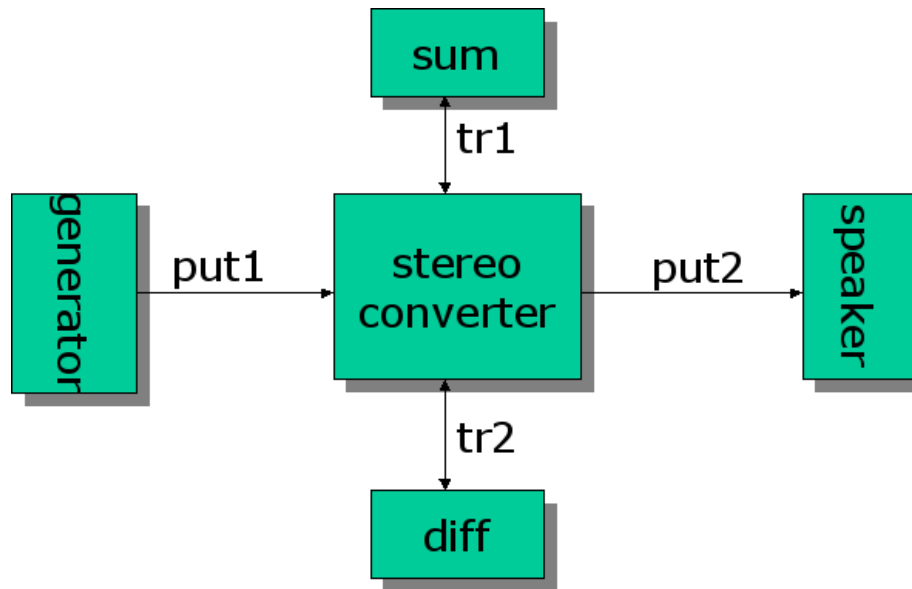


Figure 4.1: Audio Decoder System Structure.

the processes are activated simultaneously. After having notified *s*, *put* places itself on hold for the notification of two events *sum_done* and *diff_done*, whose triggering means respectively that the sum and the difference results are available. Since the functional units produce a result, the communication between them and the converter is modeled with a *transport* method invocation;

4. the functional units perform their operations on each copy of the packet received, then they return the result of the computation;
5. each process that transmitted the packets to the functional units, once received its results, triggers an event that signals the completion of the corresponding computation;
6. when both events have been notified, *put* awakes, composes the result of the computation in a new audio packet, and then transmits it to the *speaker*.

4.2 System Analysis

The example, in its essence, presents a significant set of features that can be evaluated by a static analysis. The most interesting component of the system is the *converter*, which contains a *put* implementation and the two processes mentioned earlier.

Let us examine the CCFGs of the three methods, represented graphically in figure 4.2. In the CCFGs of *put*, after the entry node, nodes 1 and 2 represent the triggering of the event that awake the processes handling the functional units requests. Node 3

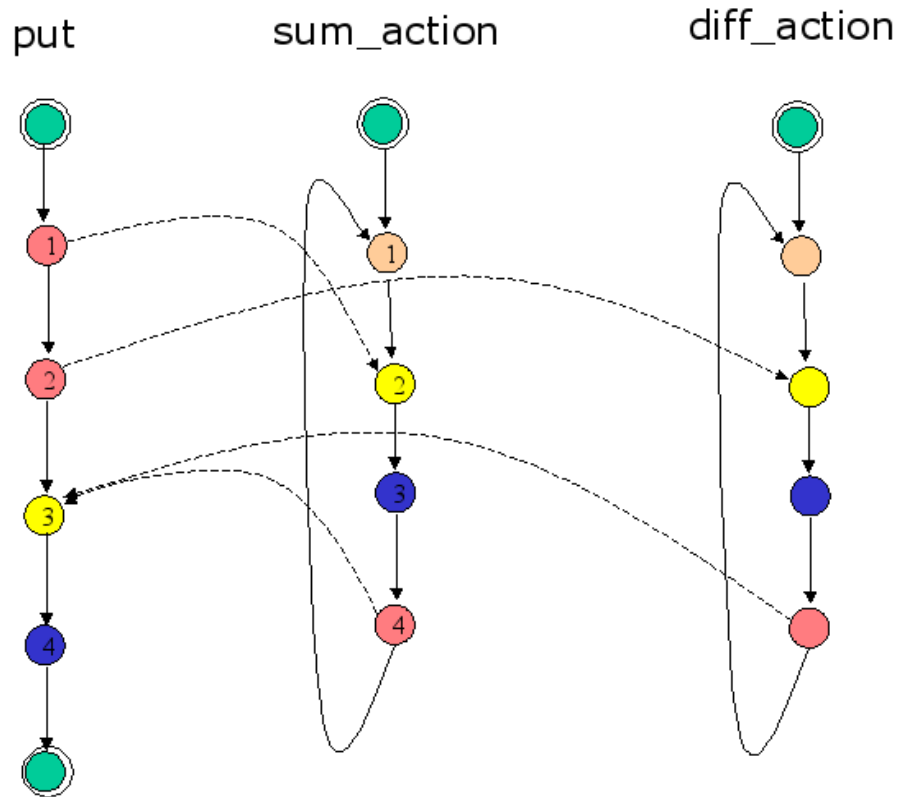


Figure 4.2: Converter Processes Interacting Communication Control Flow Graphs.

$x_i y_j$	put1	tr1	tr2	put2
put1				
tr1	*			
tr2	*	*		
put2	*	*	*	

Table 4.1: Partial ordering between services invocations

represents the suspension for notification of the functional units processes events. Node 4 represents invocation of the *put* method on the transmitter.

CCFGs of the functional units handling processes, that are totally isomorphic, are composed by node 1, which represents the entry point of a while loop, node 2, that represents the suspension for start event notification. Node 3 is the invocation of the *put* method for functional unit invocation, and node 4 is the jump to the while loop head.

Dotted lines connecting notify and corresponding wait nodes represent the synchronization information.

From the static analysis of the CCFGs, it is possible to extract *communication implication chains* for all the services implemented by components of the system. We will refer to the *put* implemented by the converter as *put1*, to the transport methods of the functional units as *tr1* and *tr2*, and to the *put* implemented by the speaker as *put2*.

The communication implications for all the services are:

- *put1*: {(tr1,tr2),put2}
- *tr1*: {}
- *tr2*: {}
- *put2*: {}

From this information, it is possible to infer a set of true sentences, representing sequence and synchronization properties of the system:

- “Every time *put1* is invoked then *tr1*, *tr2* and *put2* are invoked”
- “Every time either *t1* or *t2* are invoked, *put1* has previously been invoked”
- “Every time *put2* is invoked *tr1*, *tr2* and *put1* have been invoked”
- “*tr1* is **always** invoked after *put1*”
- “*tr2* is **always** invoked after *put2*”
- “*put2* is always invoked after *tr1*, *tr2* and *put1*”

These sentences, in turn, can be derived from a partial ordering among services invocations that can be inferred from the analysis of the communication chains. The relationship is represented in table 4.1.

tr1	*		
tr2	*		
put2	*	*	*
	put1	tr1	tr2

Table 4.2: Non-overlapping relationship among services

	put1	tr1	tr2	put2
put1	B	$0.66B$	$0.66B$	B
tr1	$1.5B$	B	B	$1.5B$
tr2	$1.5B$	B	B	$1.5B$
put2	B	$0.66B$	$0.66B$	B

Table 4.3: Bandwidth Constraints Propagation Relationship

4.3 Non-overlap

If two service invocations are in an ordering relationship, and their semantic is blocking, they do not overlap their executions in time. It is then possible to define another relationship, namely the *non-overlapping* relationship. Two invocations are non-overlapping if they are in order relationship. Relying on this definition, we can obtain the non-overlapping relationship for the services of the system, represented in table 4.2.

From the analysis of the information gathered, it is evident that the only two methods that are potentially overlapping are *tr1* and *tr2*. Recalling the system behavior, this result makes sense since *tr1* and *tr2* were modeled as parallel processes.

The non-overlapping analysis has a direct application to the problem of *resource sharing*: if two services never overlap in time, they can be implemented using the same communication resource without loss of bandwidth, since they will never generate a conflict.

4.4 Bandwidth Constraint Propagation

Another application of communication implications, as stated in section 3, is the *bandwidth constraint propagation* estimation. The bandwidth constraint propagation for the example system is represented in table 4.3.

Each table entry represents the constraint that is propagated to the invocation corresponding the row, when the invocation of the column is limited to a bandwidth value B . For instance, if we limit the bandwidth at which *tr1* can be invoked to B , at equilibrium through *put1* will be transferred data with a rate of $2/3B$ at most. This is because each invocation to *tr1* requires, to be completed, a transfer of data of size corresponding to the audio packet to be elaborated, plus the return value, that is half the size of a packet (thus $3/2P$, P being the size of one packet), and every invocation of *put1* requires passage of a single packet (thus P). Note that the bandwidth constraint propagation requires, to be evaluated, both information on communication implications

and estimation of the data size required to complete each communication action. Both information are computed statically by the metrics tool.

Section 5

Conclusions

In this document we integrated the information provided in [2] from different points of view:

- parsing and model management technology developed as a backbone for the system analysis tool was described in its essential features and technologies adopted;
- *communication implications* were introduced, in order to systematize and to rationalize some metrics previously defined;
- algorithmic description of the static analysis engine was presented;
- applications of the presented analysis to an example containing interesting features (parallel communication actions, communication tokens of different sizes, nested service invocations) were presented.

Application of the presented analysis to the first executable model produced, in the “higher” portion of the design flow, could allow the designer to:

- gain “intuitive” insight into the communication dynamics of the system;
- have a clearer picture of what could be, in the subsequent implementation phases, the critical communication channels to be implemented;
- gather information that can be exploited in the synthesis of the system. In particular, it is possible to detect communication actions that are always non-overlapping. This information can be exploited for resource allocation (sharing).

For other, more detailed description of the static analysis applications, refer to [2].

Appendix A

Application Example Code

```
#include "systemc.h"
#include "tlm_interfaces/tlm_core_ifs.h"

#define PACKETLEN 3

typedef struct audio_packet
{
    int samples[PACKETLEN];
} audio_packet;

typedef struct stereo_audio_packet
{
    audio_packet right;
    audio_packet left;
} stereo_audio_packet;

typedef struct diff_audio_packet
{
    audio_packet sum;
    audio_packet diff;
} audio_diff_packet;

class generator: public sc_module
{
public:
    sc_port<tlm::tlm_blocking_put_if<
        stereo_audio_packet>> out_port;

    void action()
    {
```

```

        wait(1,SC_NS);
        while(true)
        {
            int sample;
            stereo_audio_packet packet;

            for (int i=0;i<PACKET_LEN;i++)
            {
                cin >> sample;
                packet.right.samples[i]=
                    sample;
                cin >> sample;
                packet.left.samples[i]=
                    sample;
            }

            out_port->put(packet);
        }
    }
    SC_CTOR(generator)
    {
        SC_THREAD(action);
    }
};

class stereo_converter: public sc_channel, public tlm::
    tlm_blocking_put_if<stereo_audio_packet>
{
    public:
    sc_port<tlm:: tlm_transport_if<stereo_audio_packet
        , audio_packet> > sum_port;
    sc_port<tlm:: tlm_transport_if<stereo_audio_packet
        , audio_packet> > diff_port;
    sc_port<tlm:: tlm_blocking_put_if<
        diff_audio_packet> > out_diff_packet_port;

    sc_event sum_fire;
    sc_event diff_fire;
    sc_event sum_done;
    sc_event diff_done;

    stereo_audio_packet _packet;
    audio_packet _diff;
    audio_packet _sum;

```

```
void put(const stereo_audio_packet& packet)
{
    _packet=packet;
    sum_fire.notify();
    diff_fire.notify();
    wait(sum_done & diff_done);
    diff_audio_packet _diff_packet;
    _diff_packet.diff=_diff;
    _diff_packet.sum=_sum;
    out_diff_packet_port->put(_diff_packet);
}

void sum_action()
{
    while(true)
    {
        wait(sum_fire);
        _diff=diff_port->transport(
            _packet);
        sum_done.notify();
    }
}

void diff_action()
{
    while(true)
    {
        wait(diff_fire);
        _sum=sum_port->transport(_packet)
        ;
        diff_done.notify();
    }
}

SC_CTOR(stereo_converter)
{
    SC_THREAD(diff_action);
    SC_THREAD(sum_action);
}

};

class stereo_to_diff: public sc_channel, public tlm::
    tlm_transport_if<stereo_audio_packet, audio_packet>
{
```

```
public:
    audio_packet transport(const stereo_audio_packet&
        packet)
    {
        audio_packet* diff=new audio_packet;
        for (int i=0;i<PACKET_LEN;i++)
            diff->samples[i]=packet.left.
                samples[i]-packet.right.
                samples[i];

        return *diff;
    }
    SC_CTOR(stereo_to_diff)
    {};
};

class stereo_to_sum: public sc_channel, public tlm::
    tlm_transport_if<stereo_audio_packet, audio_packet>
{
    public:
        audio_packet transport(const stereo_audio_packet&
            packet)
        {
            cout << "transport invoked on stereo to
                sum" << endl;
            audio_packet* sum=new audio_packet;
            for (int i=0;i<PACKET_LEN; i++)
                sum->samples[i]=packet.left.
                    samples[i]+packet.right.
                    samples[i];

            return *sum;
        }

        SC_CTOR(stereo_to_sum)
        {};
};

class diff_packet_monitor: public sc_channel, public tlm
    ::tlm_blocking_put_if<diff_audio_packet>
{
    public:
        void put(const diff_audio_packet& packet)
        {
            cout << "packet arrived" << endl;
        }
};
```

```
        SC_CTOR( diff_packet_monitor )
        {;}
};

int sc_main( int argc , char** argv )
{
    generator g1("g1");
    stereo_converter sc1("sc1");
    stereo_to_sum sts1("sts1");
    stereo_to_diff std1("std1");
    diff_packet_monitor dpm1("dpm1");

    g1.out_port(sc1);
    sc1.sum_port(sts1);
    sc1.diff_port(std1);
    sc1.out_diff_packet_port(dpm1);

    sc_start(-1);
}
```

Bibliography

- [1] GCC - GNU Compiler Collection. <http://gcc.gnu.org>.
- [2] G. Agosta, F. Bruschi, and D. Sciuto. Metrics for the early communication cost estimation definition. Icodes deliverable, Politecnico di Milano, 2005.
- [3] R. Cordone, F. Ferrandi, M. D. Santambrogio, G. Palermo, and D. Sciuto. Using speculative computation and parallelizing techniques to improve scheduling of control based designs. In *Proceedings of the 2006 Conference on Asia South Pacific Design Automation: ASP-DAC 2006, Yokohama, Japan, January 24-27*, pages 898–904, 2006.
- [4] M. Girkar and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):166–178, March 1992.
- [5] J. Merril. Generic and gimple: A new tree representation for entire functions. In *Proceedings of GCC Developers Summit*, pages 171 – 180, 2003.