TLM-2 Base Protocol Analysis for Model-Driven Design

Salaheddine Hamza Sfar and Rached Tourki
Department of physics, electronic and microelectronic lab
Faculty of sciences at Monastir
Monastir, Tunisia

***Abstract*—The system-on-chip design cost is not only dependent on implementation and manufacturing techniques, but also on the used methodologies and design tools. In recent years, transaction level modelling (TLM) and more specifically the SystemC TLM-2 library has become the standard in writing a system-level specification. Even though TLM-2 based models are more abstract than registry-level ones, they are very challenging to develop. They are often written manually and from scratch. In this paper, we expose a more elaborate and modular structure of transaction level models based on more predictable semantics. This work will be our first stone of the building of a model-driven design, a methodology that has proven itself in software engineering.**

Keywords—Electronic system level design; systemC; transaction level modelling; model-driven engineering

I. Introduction

Over the years, a race is set to elevate the levels of abstraction of systems on chip descriptions to cope with their incessant rise in complexity. This gives birth to a new field of research called Electronic System Level (ESL). Nowadays Transaction Level modelling (TLM) is among the most promising ESL methodology. Transaction level (TL) models differ from register-transfer level (RTL) models by using neither clock nor signals. The designer describes the communication behavior of a module using function calls that define a set of transactions over a set of channels. Verification, architecture exploration or early stage software development and validation are the main use cases of these models [1]-[7].

TLM as a concept is not tied to one language, but nowadays, SystemC and its TLM-2 library established himself as the standard when writing TL models [8]-[10]. The Interoperability is the main value of this library. It is achieved by defining transactions using core interfaces (blocking, non-blocking and direct memory interfaces) between an initiator's socket and a target's socket. This establishes a transactional interconnection in which the data passing is carried in the generic payload (GP) format defining standards slots for the information's attributes. The library defines a set of phases that mark the beginning and the end of a request-response and defines a base protocol (BP) that enumerates rules to establish valid sequences between the initiator and the target. TLM-2 library offers resources to write TL models in two coding styles that correspond to two timing granularities: loosely-timed (LT) and approximately-timed (AT). The LT coding style delimits each transaction with two timing points, marking the start and the end of the transaction. While the AT coding style breaks a transaction down into multiple phases, with explicit timing points marking the transition between phases.

In this paper we focus on our expertise in TLM by detailing a coherent and a clear structure for the LT and the AT models. We depict several methods involved in communication and specify their interactions. This is in order to automate the generation of partial implementation.

The rest of this paper is organized as follows. Section 2 provides works related to the meet of model-driven design and ESL design. Section 3 sums our TL models' structuring proposal. Sections 4 and 5 focus on TL models using AT coding style for model driven design. Finally, Section 6 concludes the paper.

II. Motivation and Related Works

With our experience in developing TL models summarized in [11] we can make two essential remarks. On one hand, we note that the number of line codes easily reaches a few tens of thousands even if the TL model contains only one processor, a memory module and two or three hardware modules interconnected with shared bus. Such model is not easy to write and take a lot of time to debug since it is written manually from scratch and the development environment is very rudimentary: a text editor, a command line compiler and a classic debugger. On the other hand, we remark that as the TL model is organized as its source code contains repeatable and/or predictable parts. We believe that a dedicated tool could generate them from a schematic representation for example. A graphical user interface will be, certainly more ergonomic than the poor and conventional general-purpose programming environment. It will considerably reduce the efforts of coding and debugging. Moreover, it will facilitate the integration of third party module and the model's verification. This will surely have a positive impact on productivity and better teamwork.

One can argue that an industrial electronic design automation (EDA) tool should do the job. For example, in [12] the authors promote the capabilities of the Vista Model Builder from Mentor Graphics. The tool enables developers to express their designs in terms of general purpose graphical programming representations, such as state machines and structure diagrams, Models generated by this tool, delink functionality, power and timing from each other, in order to handle a unique modular behavioral description throughout the design flow. A breakdown is given in [3] and the authors show up that the Mentor's tool is very exciting. However, it adopts a proprietary approach, like any commercial tool, which makes customized models very difficult to develop. As long as we leave the crosswalks, the tool misinterprets the structure of the custom model.

Another promising methodology, to address the non-stoppable complexity, is the model-driven software engineering (MDSE) or model-driven engineering (MDE). MDSE is a software engineering paradigm appeared for about ten years and mainly focused on software development for specific application domains such as telecom, aerospace, healthcare, insurance and biology [13]-[23]. MDSE encompasses three major approach model-driven architecture (MDA), model-driven software development (MDSD) and domain-specific modeling (DSM). Although these terms are based on the same paradigms, there are certain nuances. All use a computation independent model as the starting model. Next, the designer captures domains-related specifications to build a platform-independent model (PIM). PIMs are formal models intimately linked to the targeted domain; however, they are completely independent of the later implementation. In the most case, they are written with UML that has been adapted via profiles to the targeted domain. A domain-specific language (DSL) [16] can be used to formalize PIMs. Via model transformation, usually automated with tools, successive platform-specific models (PSMs) are created from the PIMs to get finally a target platform. Such platforms are source code written, for example, in CORBA, J2EE, .NET, C++ or proprietary frameworks. The tools used to transform a PIM into a PSM or a PSM to another PSM or a PSM to code are transformation engines and generators that analyze certain aspects of input models and then synthesize various types of artifacts, such as simulation inputs, XML deployment descriptions, alternative model representations, or source code.

The separation of PIM and PSM is a key concept the MDA approach that enables better platform reuse, nevertheless the code generation is often partial and requires semi-automatic or manual completion.

Comparing with MDA, MDSD approach presents several differences. Transformations in MDSE focus for translating model into code. In this case, PIM contains all necessary details to be translated into code. The target platform is decomposed into three parts:

- Generic code: identical for all applications.
- Schematic code: systematically generated from architecture patterns
- Individual code: application specific.

It is clear that the generic code and the schematic code can be generated automatically; however, the individual code is not. MDSD does not aim on hundred percent code generation like MDA approach.

Finally, DSM approach does not favor the use of UML or UML extensions, instead the designer specify model with domain specific language. Similar to MDSD, DSM proposes to generate the solution from PIM without the need of intermediate PSM but it aims a full code generation.

After our brief presentation on the MDSE methodology, we can conclude that it adopts the same philosophy as the modern system on chip design methodologies, i.e. electronic system level design such as transaction level modeling. The points of convergence are listed in Table I.

TABLE I. CONVERGENCE BETWEEN MDSE AND ESL

Concepts	**MDSE**	**ESL (TLM)**
Abstraction	Abstract specific realization	Abstract details like signal clock
Interoperability	Not specific of any language	Assumed by SystemC standard
Separation of concerns	Application code and infrastructure code	Communication and computation

During the past years, numerous works, such as in [24]-[30], were made to adopt the MDE approach to the embedded systems and system-on-chip design. The key points of success of MDE approach is that the syntax and semantics of used models are clearly defined. Moreover, MDE tools impose domain-specific constraints and perform model checking that can detect and prevent many errors early in the life cycle.

In [30], the authors propose customizable system structure template based transaction level design (SST-TLM). These templates represent typical system structures that author named mainstream. They have two timing granularities: time-approximate and cycle-accurate. The authors develop an extensible template description (EDT) framework to enable the designer to customize architecture parameters. TL models are automatically generated by a house made tool named TL_Platform Creator. The work presented looks goods but presents some shortages. On one hand, SST-TLM is limited to three mainstreams, and the EDT efficiency depends on the contents of the TemplateDef library. On the other hand, there are no hints about neither third party block integration nor about IP-XACT standard support.

MDSD Approach and TL_Platform are the closest to our work, since automatic generation of TL models is a major motivation for us.

III. TL MODELS' STRUCTURING PROPOSAL

This section exposes key ideas that we used to structure our TL models.

A. Structuring the Module's Object Classe

As mentioned in the introduction, a TL model is no more than a set of communicating entities, each on is called a module. Each one communicates with the outside world through one or more sockets. To establish a communication, a module can act as target, initiator or alternating the two roles. In all cases, we decompose a module into two parts: a core and a wrapper to get separate communication and computation. The core implements computation while the wrapper handles the communication with the other modules. The class diagram of this organization is shown into Fig. 1.

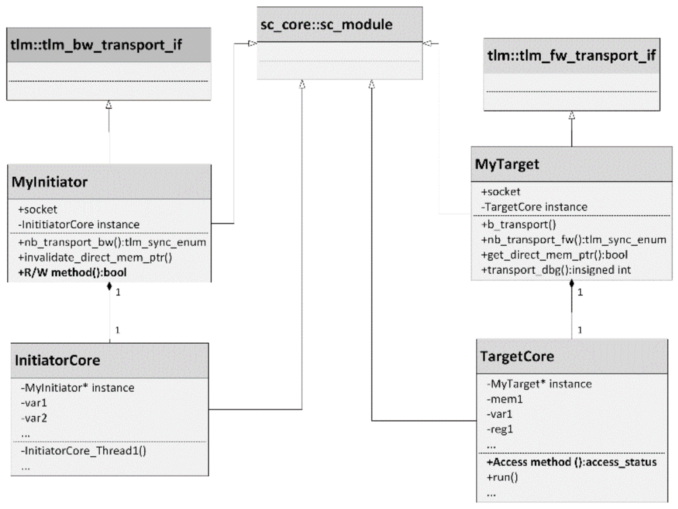


Fig. 1. UML class diagram for a pair Initiator-Target.

Class MyInitiator and class InitiatorCore represents respectively the initiator's wrapper and the initiator's core. MyInitiator inherits from two classes: sc_module and tlm::tlm_bw_transport_if<>. The last class is used to set the public member "socket" as an initiator socket instance. A Similar hierarchy is used for the target side. We made the association between a core and its wrapper by binding the wrapper's pointer to the core using the attribute "InitiatorCore instance". This solution avoids the use of additional SystemC objects to bind a wrapper to a core; objects such as ports or FIFOs might distort the model performances. In addition, by binding the wrapper's pointers to their respective cores, we make wrappers the only SystemC modules visible at the top level of the model.

Autonomous modules are specific modules widely used in system-on-a-chip design. Hardware acceleration modules are the typical case of such modules, they have a target socket and an initiator socket. They operate as target during a configuration phase and then act as master in the computation phase. Fig. 2 shows the class diagram of such module. It shows that the wrapper of such module inherits from the both TLM interface since it has two types of sockets. We consider mandatory a specific thread that will allow the autonomous module to switch between the target mode and the initiator mode. We chose to implement this thread by a finite state machine.

Modules with multiple sockets induce the problem of implementation of TLM-2 methods, since the standard permit only one implementation with standard signature. For example, in a module with multiple target sockets, the wrapper permits only one implementation of a b_transport (). A unique b_transport () cannot determine through which socket the method call has arrived and thus cannot identify the caller. One solution will be the use of a convenience socket. It provide methods to register callbacks for incoming interface method calls. Each socket will register its own b_transport method. Another solution is to define a TLM API class for each socket in the target, and then each class will inherit from nb_fw_transport_if and sc_module and implements the inherited methods. The designer instantiates these classes in the wrapper and bind them to the corresponding target sockets. The first solution is strongly advised to get an easy readable model.

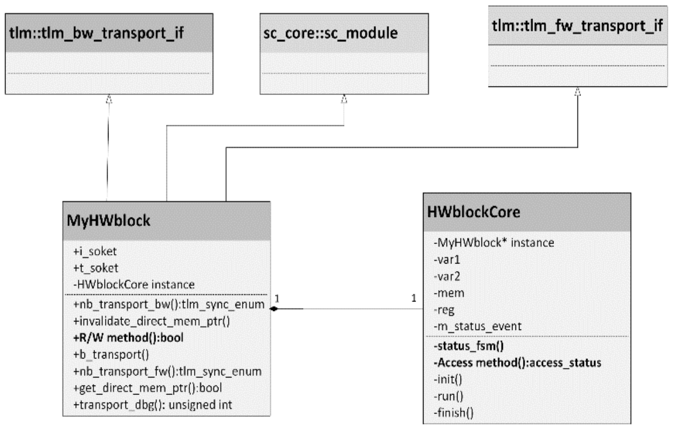


Fig. 2. Class diagram of an autonomous module.

B. Definition of Additional Methods

Among the TL model the one that uses the LT coding style is the simplest. In addition to the blocking transport methods, we propose two other methods. We named these proposed methods "**R/W methods**" and "**Access method**". An **R/W method** is implemented in an initiator's wrapper, whereas an **Access method** is implemented in target's core. All **R/W methods** must have a Boolean return value; on the other hand, the designer must define an enumerated type as a return value of the **Access method**. According to these defined values, the designer adjusts the response status and the delay annotation in b_transport body before return. If an **Access method** has triggered any computation in the target's core, additional delay should be considered.

In TL models with the AT coding style, things get complicated and we introduce more methods. Moreover, the **R/W methods** are slightly modified to handle non-blocking communication, the main feature of the AT coding style. When using such communication scheme, the designer should adopt TLM-2 base protocol (BP). This protocol defines a complete sequence composed of four phases as follows: (BEGIN_REQ → END_REQ → BEGIN_RESP → END_RESP). Thus, we divide the transaction into six methods: **R/W methods**, **nb_transport_bw** and **end_response_method** implemented in the initiator's wrapper, and **end_request_method**, **begin_response_method** and **nb_transport_fw** implemented in the target's wrapper.

As recommended in TLM-2 manual, we use the payload event queue (PEQ) to manage the exchange of payloads between the proposed methods. Payloads are injected into a PEQ with a delay annotation and then they emerge from the PEQ at a time calculated from the current simulation time plus the annotated delay. **End_response_method**, **end_request_method** and **begin_response_method** are made sensitive respectively to **m_end_response_PEQ**, **m_request_PEQ** and **m_response_PEQ**. Fig. 3 resumes all proposed methods.

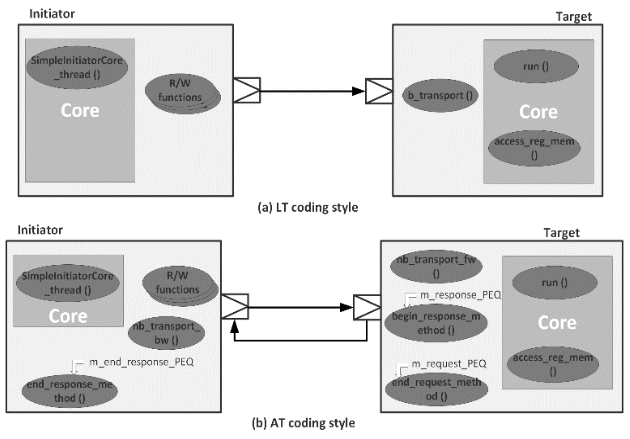


Fig. 3. Additional methods proposal.

IV. Preliminary BP Analysis

In addition to the complete sequence, the BP defines several valid sequences that omit some phases. Our preliminary analysis of the basic protocol is based on the phase diagram shown in Fig. 4. We refer to each method call by A_i where $i \in \{1,2,3,4\}$. This index marks the phase of the transaction after calling a TLM-2 non-blocking interface. The values 1, 2, 3 and 4 mark BEGIN_REQ, END_REQ, BEGIN_RESP and END_RESP, respectively. We used the index value 0 to mark the beginning of the transaction. The index values 4 and 5 denote the end of a transaction. The value 5 indicates that the return value is TLM_COMPLETED. R_{ij} refers to the call returns: indexes i and j refer respectively to the call phase and the return phase. The diagram of Fig. 4 shows all valid sequences. We can simply find them by applying the following rules:

- A valid sequence must begin with a call A_1;
- A valid sequence is an alternation between a call and a call return with the respect the precedence rules imposed by the complete sequence
- A valid sequence must end by R_{i4} or R_{i5}.

The exposed model is not restrictive to a point-to-point communication. Multipoint topology can be easily divided into a multitude of point-to-point interconnections. Fig. 5 gives an example of typical shared bus topology.

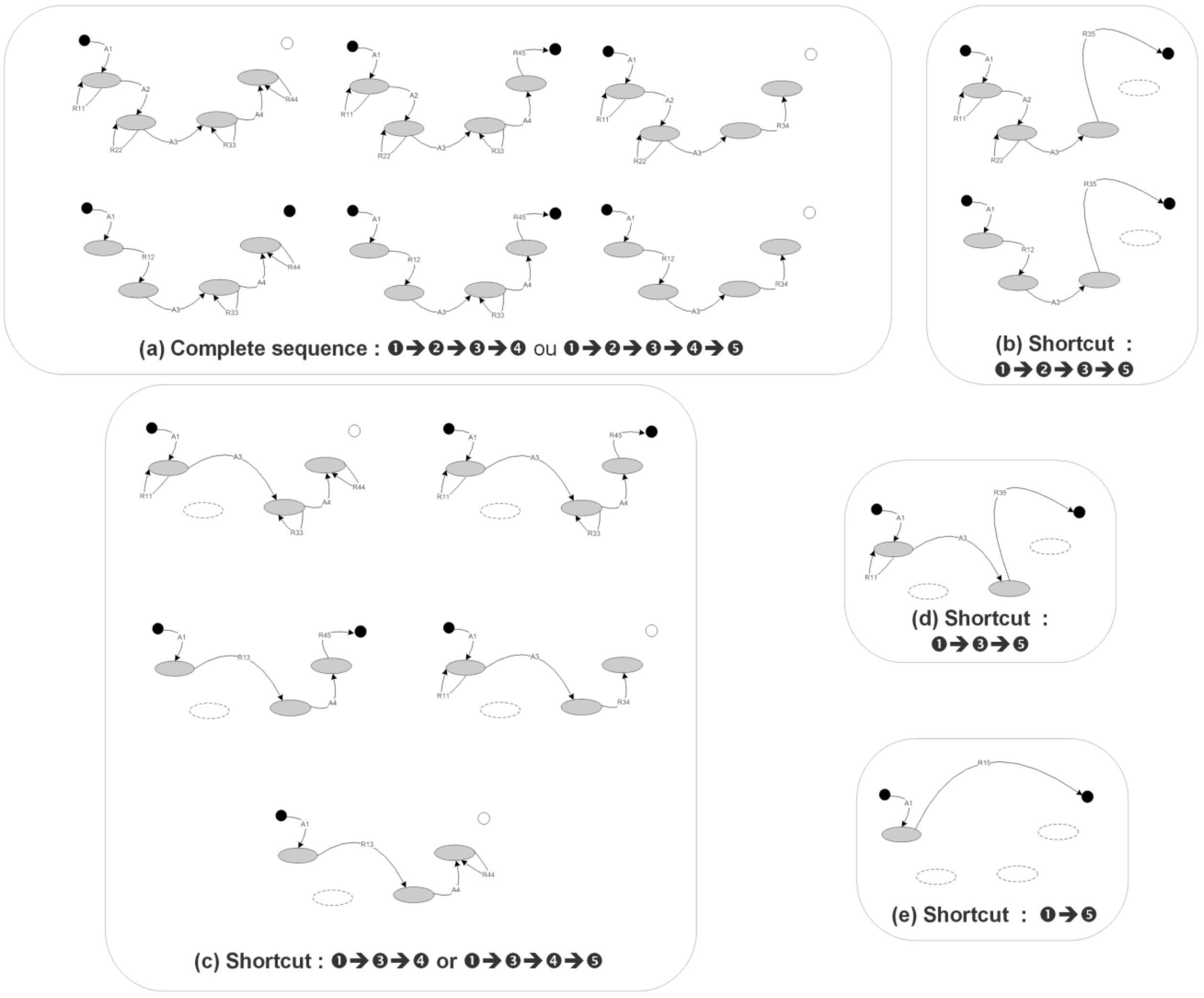


Fig. 4. Base protocol permitted sequences.

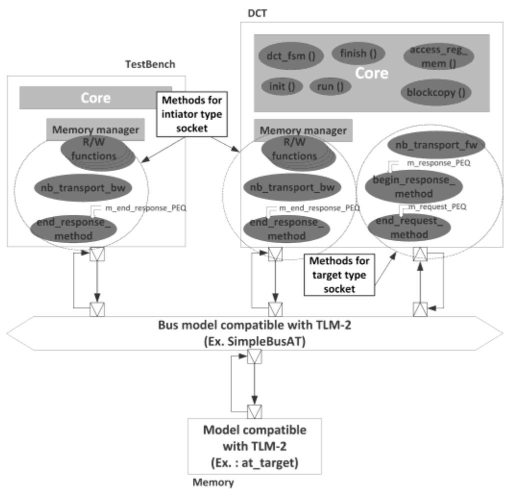


Fig. 5. Typical shared bus architecture.

V. Deeper BP Analysis

In the two previous sections, we have shown that the structuring of the models, that is to say the class diagrams and the implementation of the different methods, is an essential step for the automatic code generation. The preliminary study of the base protocol also shows that the input information will evidently be one of the fifteen phase diagrams of Fig. 4.

In this section, we will detail what temporal constraints are concerned in each associated temporal constraints graphs and how to insert them into the proposed methods. We must keep in mind that the designer may not "master" the behavior of all system's components, especially when he integrates third party TL modules in his design. For example, when an **R/W method** carries out the A_1 call, it advances transaction phase to BEGIN_REQ. Next, the interface of the forward path can accept the transaction, change the transaction phase or complete the transaction. Therefore, the designer of the target will have the choice between four situations. Every choice, he makes, will have an impact on the progress of the transaction. We say that BEGIN_REQ is the first point of divergence that offers several possible evolutions of the transaction. Similarly, BEGIN_RESP is the second point of divergence and END_RESP is the third one. So A_1 return call can move the transaction from the first point of divergence to the second or the third one, or terminates the transaction. If the transaction is moving towards its second point of divergence, it is up to the designer of the initiator who decides how the transaction has to evolve. That is to say, he can decide to move to the third point of divergence or complete the transaction. It is obvious that if a transaction is in his third point of divergence, it is still the interface of the forward path in the target's wrapper, which then decides how to finish the transaction.

Fig. 6 reorganizes all possible transaction sequences of BP by taking into account the key ideas mentioned above. It shows that there are only eight possible graphs of temporal constraints.

The base protocol involves three timing constraints, the target sets two and the initiator sets only one. The target sets the **request_accept_delay**: it is the minimum time that the initiator must comply before sending another request. It separates BEGIN_REQ and END_REQ. Suppose we have a transaction with write command, and then BEGIN_REQ marks the moment when the data is ready to be transferred from the initiator to the target. Thus, it marks the moment of sending the first byte. It is then natural that the target will delay END_REQ until it receives the last byte. Nevertheless, according to BP rules, the target is not obliged to notify END_REQ, it may skip this phase to go directly notify the BEGIN_RESP. In this case, the target sets the latency: it is the delay between BEGIN_REQ and BEGIN_RESP. It is the minimum time required for the target to react to the requested order. If the target has already notified the END_REQ, it can delay the BEGIN_RESP with a **read_delay** or **write_delay.** Therefore, we can say that:

<target> latency=<target>request_accept_delay+<target> delay

The initiator configures a single time constraint called **response_accept_delay**: it separates BEGIN_RESP and END_RESP. To understand the meaning of this delay, consider a transaction with read command. BEGIN_RESP marks the moment when the data is made available to the initiator. This is, also, the moment when the first byte starting to transit to the initiator. Therefore, the initiator notifies the end of the response when receiving the last byte. Of course, relying on the BP's rules, this is not an obligation. Fig. 7 and 8 give code details of the eight sequences.

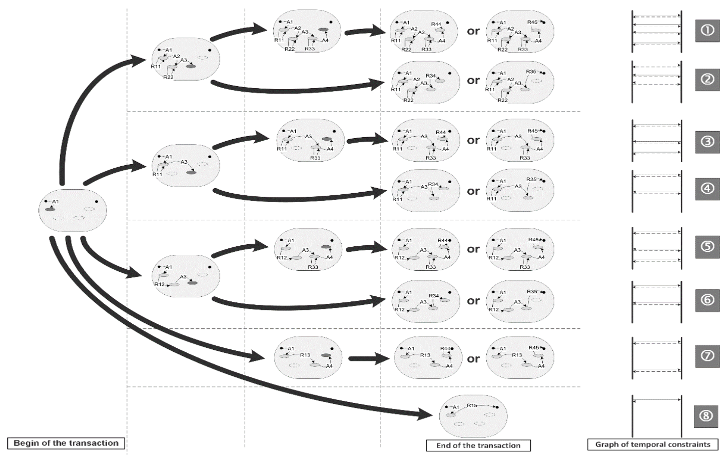
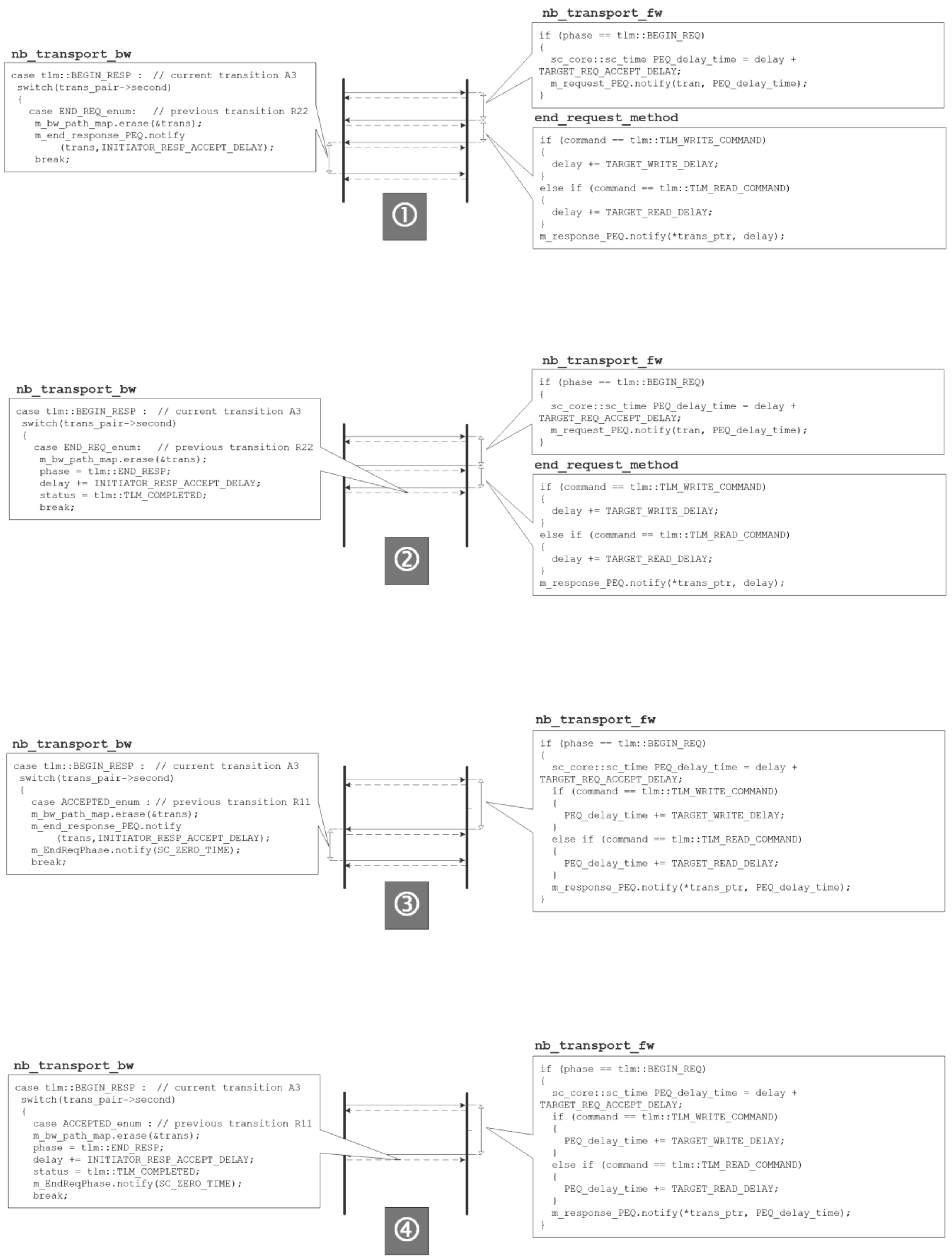


Fig. 6. All permitted transaction sequences in TLM-2 base protocol.



nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case END_REQ_enum:   // previous transition R22
    m_bw_path_map.erase(&trans);
    m_end_response_PEQ.notify
        (trans,INITIATOR_RESP_ACCEPT_DELAY);
    break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  m_request_PEQ.notify(tran, PEQ_delay_time);
}
```

end_request_method

```
if (command == tlm::TLM_WRITE_COMMAND)
{
  delay += TARGET_WRITE_DElAY;
}
else if (command == tlm::TLM_READ_COMMAND)
{
  delay += TARGET_READ_DElAY;
}
m_response_PEQ.notify(*trans_ptr, delay);
```

nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case END_REQ_enum:   // previous transition R22
    m_bw_path_map.erase(&trans);
    phase = tlm::END_RESP;
    delay += INITIATOR_RESP_ACCEPT_DELAY;
    status = tlm::TLM_COMPLETED;
    break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  m_request_PEQ.notify(tran, PEQ_delay_time);
}
```

end_request_method

```
if (command == tlm::TLM_WRITE_COMMAND)
{
  delay += TARGET_WRITE_DElAY;
}
else if (command == tlm::TLM_READ_COMMAND)
{
  delay += TARGET_READ_DElAY;
}
m_response_PEQ.notify(*trans_ptr, delay);
```

nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case ACCEPTED_enum : // previous transition R11
   m_bw_path_map.erase(&trans);
   m_end_response_PEQ.notify
        (trans,INITIATOR_RESP_ACCEPT_DELAY);
   m_EndReqPhase.notify(SC_ZERO_TIME);
   break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  if (command == tlm::TLM_WRITE_COMMAND)
  {
    PEQ_delay_time += TARGET_WRITE_DElAY;
  }
  else if (command == tlm::TLM_READ_COMMAND)
  {
    PEQ_delay_time += TARGET_READ_DElAY;
  }
  m_response_PEQ.notify(*trans_ptr, PEQ_delay_time);
}
```

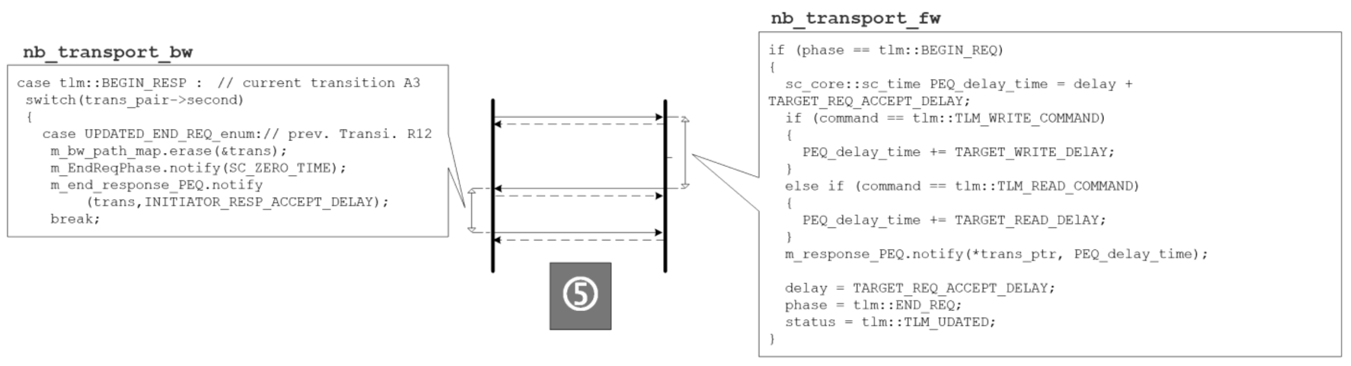
nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case ACCEPTED_enum : // previous transition R11
   m_bw_path_map.erase(&trans);
   phase = tlm::END_RESP;
   delay += INITIATOR_RESP_ACCEPT_DELAY;
   status = tlm::TLM_COMPLETED;
   m_EndReqPhase.notify(SC_ZERO_TIME);
   break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  if (command == tlm::TLM_WRITE_COMMAND)
  {
    PEQ_delay_time += TARGET_WRITE_DElAY;
  }
  else if (command == tlm::TLM_READ_COMMAND)
  {
    PEQ_delay_time += TARGET_READ_DElAY;
  }
  m_response_PEQ.notify(*trans_ptr, PEQ_delay_time);
}
```

Fig. 7. Implementations of temporal constraints.



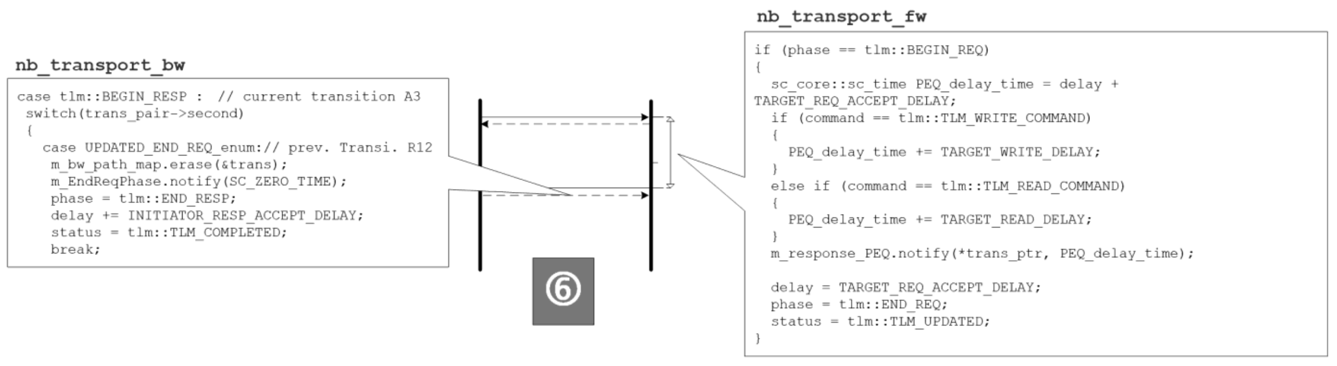
nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case UPDATED_END_REQ_enum:// prev. Transi. R12
    m_bw_path_map.erase(&trans);
    m_EndReqPhase.notify(SC_ZERO_TIME);
    m_end_response_PEQ.notify
        (trans,INITIATOR_RESP_ACCEPT_DELAY);
    break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  if (command == tlm::TLM_WRITE_COMMAND)
  {
    PEQ_delay_time += TARGET_WRITE_DElAY;
  }
  else if (command == tlm::TLM_READ_COMMAND)
  {
    PEQ_delay_time += TARGET_READ_DElAY;
  }
  m_response_PEQ.notify(*trans_ptr, PEQ_delay_time);

  delay = TARGET_REQ_ACCEPT_DELAY;
  phase = tlm::END_REQ;
  status = tlm::TLM_UDATED;
}
```



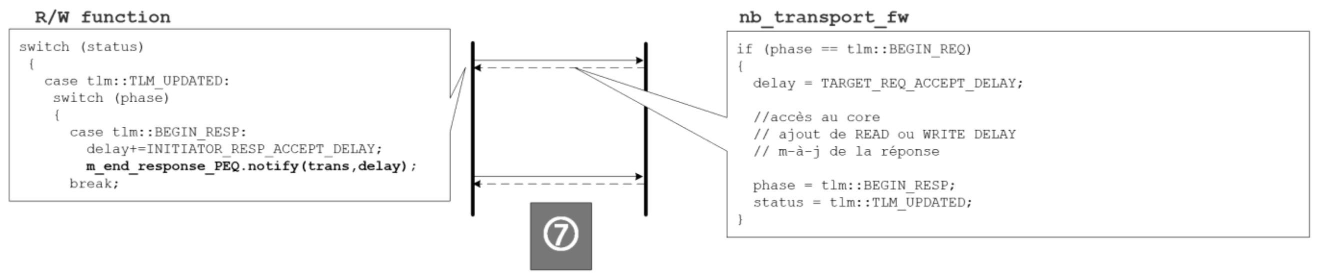
nb_transport_bw

```
case tlm::BEGIN_RESP :  // current transition A3
 switch(trans_pair->second)
 {
   case UPDATED_END_REQ_enum:// prev. Transi. R12
    m_bw_path_map.erase(&trans);
    m_EndReqPhase.notify(SC_ZERO_TIME);
    phase = tlm::END_RESP;
    delay += INITIATOR_RESP_ACCEPT_DELAY;
    status = tlm::TLM_COMPLETED;
    break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  sc_core::sc_time PEQ_delay_time = delay +
TARGET_REQ_ACCEPT_DELAY;
  if (command == tlm::TLM_WRITE_COMMAND)
  {
    PEQ_delay_time += TARGET_WRITE_DELAY;
  }
  else if (command == tlm::TLM_READ_COMMAND)
  {
    PEQ_delay_time += TARGET_READ_DElAY;
  }
  m_response_PEQ.notify(*trans_ptr, PEQ_delay_time);

  delay = TARGET_REQ_ACCEPT_DELAY;
  phase = tlm::END_REQ;
  status = tlm::TLM_UPDATED;
}
```



R/W function

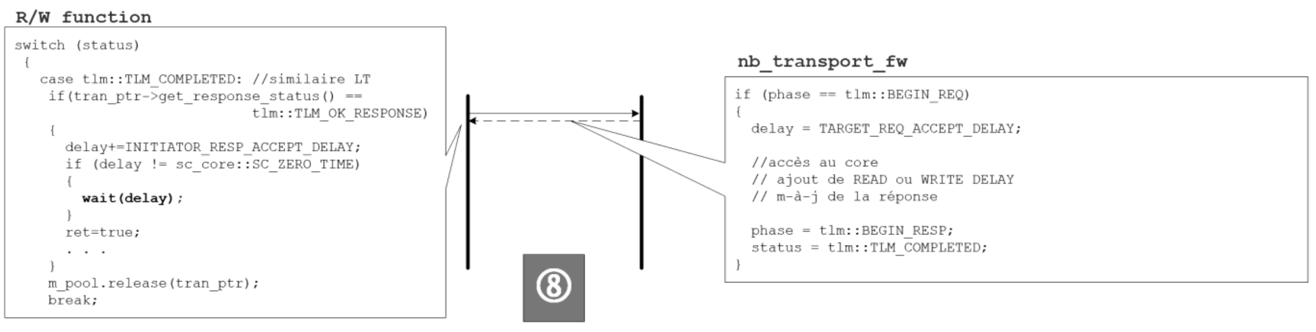
```
switch (status)
 {
   case tlm::TLM_UPDATED:
    switch (phase)
    {
      case tlm::BEGIN_RESP:
        delay+=INITIATOR_RESP_ACCEPT_DELAY;
        m_end_response_PEQ.notify(trans,delay);
      break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  delay = TARGET_REQ_ACCEPT_DELAY;

  //accès au core
  // ajout de READ ou WRITE DELAY
  // m-à-j de la réponse

  phase = tlm::BEGIN_RESP;
  status = tlm::TLM_UPDATED;
}
```



R/W function

```
switch (status)
 {
   case tlm::TLM_COMPLETED: //similaire LT
    if(tran_ptr->get_response_status() ==
                            tlm::TLM_OK_RESPONSE)
    {
      delay+=INITIATOR_RESP_ACCEPT_DELAY;
      if (delay != sc_core::SC_ZERO_TIME)
      {
        wait(delay);
      }
      ret=true;
      . . .
    }
    m_pool.release(tran_ptr);
    break;
```

nb_transport_fw

```
if (phase == tlm::BEGIN_REQ)
{
  delay = TARGET_REQ_ACCEPT_DELAY;

  //accès au core
  // ajout de READ ou WRITE DELAY
  // m-à-j de la réponse

  phase = tlm::BEGIN_RESP;
  status = tlm::TLM_COMPLETED;
}
```

Fig. 8.   Implementations of temporal constraints (cont.).

The sequence N° 1 represents the complete sequence. In this case, all methods are mandatory. The **request_accept_delay** delays **end_request_method** against R/W method and so against **nb_transport_fw**. This later annotates **request_accept_delay** when the transaction is injected in **m_end_request_PEQ**. This PEQ is in the list of sensitivity of **end_request_method**. In turn, **end_request_method** annotates read or write delay when the transaction is injected in **m_response_PEQ**. In the initiator side, it is **nb_transport_bw** that annotates **response_accept_delay** when the transaction is injected in **m_end_response_PEQ**. Finally, it is **end_response_method** that makes the second call of **nb_transport_fw** and then restores the transaction object to the memory manager.

Sequence N° 2 is similar to Sequence N° 1, but here no need to **end_response_method** and it is **begin_response_method** that restores the transaction object to the memory manager.

In sequence N°4, target omits the end request phase and starts directly the response phase. The target, then, injects payload in **m_response_PEQ** with an annotation equal to its latency.

In the sequence N°5, we are in the situation where **nb_transport_fw** changes the phase of the transaction to END_REQ and at the same time the target calls **begin_response_method** with a delay equal to its latency. Therefore, the target injects payload in **m_response_PEQ** with an annotation equal to its latency and at the same time, it annotates **request_accept_delay**. The initiator will honor this constraint by calling wait () within the **R/W method**. This situation must not be confused with sequence N°3 where **nb_transport_fw** returns TLM_ACCEPTED.

Situations N°7 and N°8 are particular, since there are no calls of backward interface and R/W function deals directly with the target. In the first case, it is in charge to inject payload in **m_end_response_PEQ**. The delay annotated is the delay returned by **nb_transport_fw** plus **response_accept_dela**y. In the second case, no injection in PEQ is needed, since transaction is completed. After calling nb_transport_fw, the **R/W method** just calls wait to fulfil a global delay equal to the target's latency plus **response_accept_delay**.

VI. Conclusions

In this paper, we presented a well-structured transaction level model based on SystemC TLM-2 library. Our structuring expertise shows us that many semantics can be easily integrated into a model driven design flow. We must keep in mind that SystemC is none other than a set of STL added to the C ++ language, so class diagrams can be converted into SystemC code. On the other hand, if the designer chooses one of the allowed sequences of the base protocol for a socket pair, the code of all proposed methods is predictable, as is the temporal annotation scheme.

We believe that we have established a detailed specification of the tool that will automatize the generation of our TL model.

References

[1] M. Glasser, “Transaction-Level Modeling,” Open Verification Methodology Cookbook, pp. 49–68, 2009.

[2] J. Hu, T. Li, and S. Li, “Equivalence checking between SLM and TLM using coverage directed simulation,” Front. Comput. Sci., vol. 9, no. 6, pp. 934–943, Oct. 2015.

[3] B. Bailey and G. Martin, “Transaction-Level Platform Creation,” ESL Models and their Application, pp. 309–359, Nov. 2009.

[4] G. B. Vece, M. Conti, and S. Orcioni, “Transaction-level power analysis of VLSI digital systems,” Integration, the VLSI Journal, vol. 50, pp. 116–126, Jun. 2015.

[5] K. Popovici, F. Rousseau, A. A. Jerraya, and M. Wolf, “Transaction-Accurate Architecture Design,” Embedded Software Design and Programming of Multiprocessor System-on-Chip, pp. 151–182, 2010.

[6] S. Kundu, S. Lerner, and R. K. Gupta, “Translation Validation of High-Level Synthesis,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 29, no. 4, pp. 566–579, Apr. 2010.

[7] V. Lahtinen, J. Siirtola, and T. Mäkeläinen, “Transaction-Level Modeling in Communication Engine Design: A Case Study,” Advances in Design and Specification Languages for Embedded Systems, pp. 145–156.

[8] S. Rigo, B. Albertini, and R. Azevedo, “Transaction Level Modeling,” Electronic System Level Design, pp. 25–36, 2011.

[9] A. Banerjee and B. Sur, “SystemC and SystemC-AMS in Practice,” 2014.

[10] “IEEE Standard for Standard SystemC Language Reference Manual.”

[11] S. H. Sfar, I. Bennour et R. Tourki “Stepwise SystemC/TLM-2 models structuring and optimizations”, Proceeding de IDT 2016, 11th International Design & Test Symposium, 18-20 december 2016, Hammamet, Tunisia

[12] Y. Vellery and R. Rachamim,”Realizing ESL with Scalable Transaction Level Models”,2010.

[13] T. Stahl and M. Völter, “Model-Driven Software Development”, Wiley, 2006.

[14] D. Frankel, “Model Driven Architecture”, Wiley, 2003.

[15] S. Kelly and J. Tolvanen, “Domain-Specific Modeling – Enabling Full Code Generation”, Wiley, 2008.

[16] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema, “Developing Applications Using Model-Driven Design Environments,” Computer, vol. 39, no. 2, pp. 33–40, Feb. 2006.

[17] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff, “CALM and Cadena: Metamodeling for Component-Based Product-Line Development,” Computer, vol. 39, no. 2, pp. 42–50, Feb. 2006.

[18] J. Gray, Yuehua Lin, and Jing Zhang, “Automating Change Evolution in Model-Driven Engineering,” Computer, vol. 39, no. 2, pp. 51–58, Feb. 2006.

[19] R. B. France, S. Ghosh, T. Dinh-Trong, and A. Solberg, “Model-Driven Development Using UML 2.0: Promises and Pitfalls,” Computer, vol. 39, no. 2, pp. 59–66, Feb. 2006.

[20] G. A. Moreno and P. Merson, “Model-Driven Performance Analysis,” Lecture Notes in Computer Science, pp. 135–151, 2008

[21] K. Balasubramanian, D. C. Schmidt, Z. Molnár, and Á. Lédeczi, “System Integration Using Model-Driven Engineering,” Designing Software-Intensive Systems, Designing Software-Intensive Systems: Methods and Principles, pp. 474–504, 2009.

[22] P. S. Kaliappan, H. König, and S. Schmerl, “Model-Driven Protocol Design Based on Component Oriented Modeling,” Lecture Notes in Computer Science, pp. 613–629, 2010.

[23] V. G. Diaz, J. M.C. Lovelle, B.C. P Garcia-Bustelo and O. S. Martinez, “Progressions and Innovations in Model-Driven Software Engineering”, IGI Global, 2013.

[24] Y. Jiang, H. Liu, H. Song, H. Kong, M. Gu, J. Sun, and L. Sha, “Safety-Assured Formal Model-Driven Design of the Multifunction Vehicle Bus Controller,” Lecture Notes in Computer Science, pp. 757–763, 2016.

[25] G. Martin and W. Müller, Eds., “UML for SOC Design,” 2005.

[26] E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini, “SystemC/C-based model-driven design for embedded systems,” ACM Transactions on Embedded Computing Systems, vol. 8, no. 4, pp. 1–37, Jul. 2009.

[27] I. Ben-Hafaiedh, S. Graf, and M. Jaber, “Model-based design and distributed implementation of bus arbiter for multiprocessors,” 2011 18th IEEE International Conference on Electronics, Circuits, and Systems, Dec. 2011.

[28] C. Teodorov, D. Picard, and L. Lagadec, “FPGA physical-design automation using Model-Driven Engineering”, 6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC), Jun. 2011.

[29] E. Riccobene and P. Scandurra, “Integrating the SysML and the SystemC-UML profiles in a model-driven embedded system design flow”, Design Automation for Embedded Systems, vol. 16, no. 3, pp. 53–91, Sep. 2012.

[30] D. Wang, Y. Ye, and S. Li, “System structure template based transaction level modeling”, 2011 Chinese Control and Decision Conference (CCDC), May 2011.