CSP Transactors for Asynchronous Transaction Level Modeling and IP Reuse

Lilian Janin and Doug Edwards

Advanced Processor Technologies GroupSchool of Computer Science,
The University of Manchester, UK
{lilian.janin,doug.edwards}@manchester.ac.uk

Abstract. In synchronous circuit design, new levels of abstraction above RTL allow the designer to model, simulate, debug and explore various architectures more efficiently than before. These are known as transaction level modeling. The translation between signals at different levels of abstraction is performed by pieces of code called transactors, mainly for the purpose of simulation. This paper identifies a set of asynchronous abstractions suitable for asynchronous transaction level modeling. Based on these models, we show that asynchronous CSP-based transactors can bring many more benefits than their synchronous counterparts, while being simpler to describe. We show how they can be used to automatically generate complex SystemC templates and hardware-software links, and automatically build network-on-chip interfaces facilitating IP reuse in embedded systems. Tools were developed after the techniques described in this paper. They are used in a case study to describe an asynchronous IP from transaction levels to RTL, demonstrating the automatic generation of various complex parts of the design and the minimum amount of specifications required from the designer.

1 Introduction

Chip design is becoming increasingly complex with billions of transistors and dozens of IPs being now commonly placed on single chips. In order to tackle this complexity, Electronic System Level (ESL) design has been developed extensively for the last five years, aiming at exploiting IP reuse and integrating higher levels of abstraction than traditional HDLs (Hardware Description Languages). Designing at these new levels of abstraction above RTL is known as transaction level modeling (TLM).

Over the same period, GALS (Globally Asynchronous, Locally Synchronous) techniques have made their way into chips as they provide an easy way to interconnect multiple modules with distinct clock domains, simplifying many synchronisation problems [1]. Implementing a GALS system using ESL tools is, in theory, simply a matter of linking ‘locally synchronous’ IPs to the ‘globally asynchronous’ interconnect IP (asynchronous buses or asynchronous networks on chip (NoC) such as CHAIN [2]). Although the interconnect IP is, for a GALS system, internally asynchronous, its interface is fully synchronous (devoid of asynchronous channels) in order to be connected directly to the ports of synchronous IPs. This allows ESL tools to handle only synchronous elements. Indeed, actual ESL tools are all designed to facilitate synchronous connections.

Nonetheless, it could often be beneficial to use IPs with asynchronous interfaces. An SRAM, for example, is an internally asynchronous device which sees its performance degraded when connected to the synchronous ports of an asynchronous NoC via a synchronous wrapper. This performance degradation could easily be avoided by using asynchronous interfaces to connect the SRAM IP block directly to the asynchronous NoC. One difficulty tackled in this paper is the description of these asynchronous TLM interfaces.

Another important aspect of ESL design tools and synchronous IPs is the integration of a hierarchy of models that are reusable across levels of abstraction. As specified in the OCP-IP [11] synchronous TLM proposal, tight relations from one level of abstraction to the next ensure that decisions made in the SystemC domain are carried through to the RTL domain faithfully, removing any chance for translational error and ensuring that simulations at various levels of abstraction are consistent with each other. These relations, although not inherently synchronous, are usually described using properties specific to synchronous interfaces. This paper identifies a set of properties that allow asynchronous models to be described consistently through abstraction levels. When simple properties cannot be versatile enough to describe all the possible configurations, more complex descriptions based on programming languages are introduced. This is the case with transactors, pieces of code performing the translation between signals at different levels of abstraction.

Section2 provides an overview of the abstractions proposed by the industry as standards for synchronous circuits. It introduces the transactors, and exposes the actual limitations of synchronous TLM and transactors and how asynchronous techniques are able to help. Section 3 defines a set of asynchronous abstractions suitable for asynchronous transaction level modeling. Based on these models, Section 4 shows how asynchronous transactors can be efficiently written and how they can be used to automatically generate SystemC templates, network-on-chip interconnect for IP reuse and hardware-software cosimulation links. Section 5 describes a practical example, where an asynchronous RAM IP is easily described from TLM to RTL using the tools developed after the techniques described in this paper.

2 Related Work

With the advent of ESL design, the various levels of design abstraction above RTL are referred to as Transaction Level Modeling. The primary goal of TLM is to dramatically increase simulation speeds, while offering enough accuracy for the design task at hand. The increase in speed is achieved by the TLM abstracting away the number of events and amount of information that have to be processed during simulation to the minimum required. For example, instead of driving the individual signals of a bus protocol, the goal is to exchange only what is really necessary: the data payload. TLM also reduces the amount of detail the designer must handle, therefore making modeling easier.

2.1 Transaction Level Modeling: Synchronous Abstractions

The two prevalent industry approaches are from OSCI (Open SystemC Initiative) and OCP-IP (Open Core Protocol -International Partnership): the former has focused its

TLM on the software development concerns, while the later has focused on SOC integration concerns. OSCI's TLM leads to a top-down methodology, where the design architect describes its design at a software level, via untimed software calls, and gradually refines it down to the RTL level. OCP-IP's TLM, on the other hand, leads to a methodology to specify reusable IPs through a bus-centric hierarchy of models seamlessly interconnecting via a common "socket" interface at different levels of abstraction. In spite of these different origins, both TLMs converge on several aspects. Both OSCI and OCP-IP define three levels of TLM above RTL, the former gradually refining an initial software-level model towards RTL and establishing a software-like top-down design methodology, and the latter gradually abstracting away the details of the communications between modules for IP reuse and easy on-chip interconnect. It is interesting that two initiatives coming from different fronts led to similar definitions.

2.2 Synchronous Transactors

One of the main objectives of ESL design is the transparent interoperability between IPs at different levels of abstraction. Transactors were created for this purpose: they are pieces of code performing the translation between transactions at different levels of abstraction. These transactors receive stimuli and translate the activity to the data structure, call-structure and temporal mechanisms used by design models at other abstractions. This enables each component in a system to communicate using the interface semantics native to its design abstraction. In a mixed abstraction level simulation environment, transactors are used to connect blocks modeled at different abstraction levels for simulation in different contexts of speed/accuracy. One main application of transactors is for testbenches: creating a transactor connected to a testbench makes the testbench insensitive to changes in the design. Even if the top-level interface of the design under verification is modified, only transactors have to be updated.

Traditionally, transactors were written as two unidirectional components or pieces of code: a *driver* from high to low levels and a *monitor* from low to high level. These allow test library functions written in an abstract, untimed way to exercise and check models at different abstraction levels. Transactors are often written in the same language as the one used to describe the models. When created manually, the process is long and error-prone, and is often as complex and time consuming as the creation of the RTL model itself. Moreover, transactors must be updated as the design interface changes, leading to many manual iterations. As transactors are complex to design, those created by hand often only connect two levels of abstraction. However many design simulations today have several abstractions integrated together. Consequently, handling only two abstraction levels with a transactor is not sufficient. Complex transactors that can handle interconnection of multiple levels of abstraction need to be automatically generated, and this requires the capture of the interface abstractions in a single formal description.

Recently, a few initiatives tried to define new languages for describing transactors more efficiently and for generating the simulated code automatically [3, 4]. A formal description of an interface abstraction hierarchy captures the temporal, data and behavioural aspects of an interface protocol at multiple levels of abstraction along with the mapping between the levels. Transactors can be generated from these descriptions to drive and monitor communications at various levels of abstraction.

Balarin and Passerone [5] proposed a methodology where interface protocols are specified formally in a way that is very similar to assertions used in verification, using regular expressions from the *Property Specification Language* [6]. Transactors are automatically generated from such a specification. Many transactors may be generated from a single interface specification, depending on which part of the design is being verified, what modes of the interface are being exercised, and which verification technology is being used (e.g. simulation vs. acceleration). In addition, formal interface specifications can be used as assertions and verified either statically or dynamically.

2.3 Proposed Improvements to the Synchronous TLM Models

This section presented the levels of abstraction defined as standards for synchronous transaction level modeling. These initiatives are extremely valuable as they standardise the development of synchronous IPs. However, they suffer a few drawbacks, which can be eased using asynchronous techniques.

OCP-IP's TLM defines a standard way to specify interconnectable -and thus highly reusable - IPs by defining an abstract bus interface which the designer needs to follow. The downside is that the interface of the IP is constrained by the TLM model. We claim that an asynchronous IP can rely on an automatic serialisation-deserialisation of any asynchronous interface, allowing IPs to *plug&play* to a network on chip without imposing any constraint to the interface.

Transactors provide a transparent translation between different levels of abstraction, enabling mixed-mode simulation, verification across levels, etc. Unfortunately, even using recent specification techniques to describe transactors more efficiently, their design often takes months [3]. The definition and common usage of standard asynchronous protocols makes asynchronous communications easier to describe. We show how asynchronous transactors can be developed very efficiently using existing languages.

OSCI have based their design methodology on SystemC. SystemC is appropriate for system-level design and transaction level modeling: designers can describe high level interfaces, structures and algorithms at the same time as delays and bit-level signalling. SystemC's main drawback is the complexity for the designer to describe the many required classes and to handle C++ subtleties such as exporting the abstract interfaces via inheritance in ports and channels, leading to a huge amount of work before being able to start describing the real circuit. We show how OSCI's methodology can be applied to asynchronous design and how the initial complexity of SystemC descriptions can be bypassed by using code generators based on the transactors.

The next section describes how we identified a set of abstractions applicable to asynchronous interfaces and how the translation from one level to another can be automated.

3 Asynchronous Transaction Level Modeling

The first task before being able to describe asynchronous IPs and transactors is to identify the levels of abstraction which will be useful for asynchronous transaction level modeling.

From the previous description of synchronous TLMs, we can see that synchronous communications, based on clock cycles and independent wires, are at the same time very constrained (by the clock signal) and free to implement any protocol (each wire is left to the imagination of its creator). This led to abstractions based on clock cycles: groups of events belonging to the same clock cycle and periodic events repeating every n cycles are bundled together. Asynchronous communications, on the other hand, are not constrained by a clock, and therefore abstractions cannot be based on it. However, a set of major asynchronous protocols exists, which are used in most designs, and from which a set of abstractions can be defined. This section focuses on the definition of abstraction levels based on these asynchronous protocols.

3.1 Asynchronous Abstractions

Asynchronous circuit design, based on asynchronous communications, has its roots as a high level model: the CSP (Communicating Sequential Processes) language [7], created by Hoare for describing patterns of interaction. Several CSP-based HDLs [8, 9] have been created over the years to describe and synthesise asynchronous circuits. While synchronous languages are striving to raise their level of abstraction while keeping enough control over the generated transistors, asynchronous designers have striven for years to synthesise low-level transistors from their high-level communication patterns. Asynchronous design might therefore be able to shine at the transaction level, by directly exploiting the CSP abstractions it originated from.

In synchronous TLMs, the Programmer View (PV) abstraction, based on software function calls with no communication events and no timing information, is the basis of transaction level modeling. As it is not related to any clock model, it can be defined and used in exactly the same way in an asynchronous TLM.

In OSCI's second abstraction, PV+T, timing information is added. Delays are a big problem in asynchronous simulation. In a synchronous design, every component is periodically resynchronised to the clock, minimising the imprecision by resetting any skew. In an asynchronous design, timing errors continuously accumulate, drifting to very imprecise values. Tackling the problem of asynchronous timing models is beyond the scope of this paper. So far we decided not to specify timing as part of any level of abstraction. Timing information can however still be associated to events at any level of abstraction, providing improved simulation accuracy.

In both OSCI's and OCP-IP's TLMs, the lowest abstraction above RTL is where communications are abstracted as "transfers" identical to asynchronous CSP communications. An asynchronous TLM can therefore be restricted to RTL, CSP and PV levels.

3.2 Abstraction Properties

In the context of transaction level modeling, and to enable communications to be automatically propagated and translated across levels of abstraction, we aimed to define levels of abstraction of asynchronous communications such that, given a transaction at the highest level of abstraction, a mixed-mode simulator is able to automatically generate a valid set of lowest level signals, and vice versa: given a set of low level signals, an automated process exists to regroup them appropriately to reconstitute a coherent high level transaction. For this, we identified a set of properties able to define the transitions from one level of abstraction to the next.

Such a set of properties should allow, for instance, to convert a handshake transaction from an abstraction where data validity is not taken into account (the data becomes valid at exactly the same time as the handshake request is raised) to an abstraction where data validity is considered. Depending on the specified data validity scheme, the 'converter' would need to separate the reception of the data from the reception of the request signal, possibly delaying one of them to satisfy timing validity and inserting an 'invalid data' event at the appropriate time (Fig 2). This validity scheme would be a property of the abstraction level, which should be provided by the IP connected to such a channel when simulated at or through this level of abstraction.

Fig 1 shows the properties obtained by considering the difference between the PV level and the RTL level. It can be verified that setting these properties to false or zero makes for a very abstracted high-level description of the transaction, while setting these properties to real values makes for a low-level transaction. Each of these properties can be enabled or disabled, and the combined state of the properties defines a specific level of abstraction. Using these abstraction properties, Fig 3 shows how one transaction can be simulated and converted across the abstraction spectrum.

The properties can be grouped into 4 categories: Function, Structure, Protocol and Timing (Fig 1). Structure properties specify the interface of the IP blocks in terms of I/ Os, from abstract data type transfers to physical wires. Protocol properties define the succession of events being sent or received by the IP, therefore affecting its dynamic behaviour. Structure and Protocol properties are discrete properties (often just a boolean choice). They specify a precisely defined arrangement, usually observable in the final physical circuit. On the other hand, Timing properties are estimates of real world delays and can take any real value, while Function properties define the relationship between a structural entity (such as a wire) and a functionality. For example, a RnW wire would be activated to indicate a read function and deactivated to request a write. As mentioned previously, timing properties are not included in our current model. The three lower protocol properties happen to correspond exactly to the definition of handshake protocols. Transactions were the handshake protocol is abstracted away characterise the CSP abstraction, where data is sent over individual channels following a two-events protocol: request sent with data followed by acknowledge (for a push channel), and where the interleaving of different channels is specified.

4 Asynchronous Transactors

A transactor is a process able to automatically translate a signal or set of signals from one level of abstraction to another. We defined a TLM in the previous section with the aim to facilitate these translations for asynchronous IPs. A set of simple properties were proposed, that are able to direct the conversion between two contiguous levels of abstraction. Based on the levels of abstraction presented in the previous section, this section shows why asynchronous transactors are needed, how they can be written efficiently and how

Function
- Relation between structural entity and functionality

Structure
- High level group over space
  (1 transaction on multiple channels)
- High level group over time
  (multiple transactions on one channel)

Protocol
- Interleaving of channel protocols
- Protocol expansion
- Data validity (single-rail)
- Return to zero (protocol phase & processing delay)

Plus (not included in our model):

Timing
- Processing delay (delay between inputs and outputs)
- Timing constraints between inputs
  (e.g. data setup&hold time in single-rail)

Fig. 1. Abstraction properties of a handshake transaction

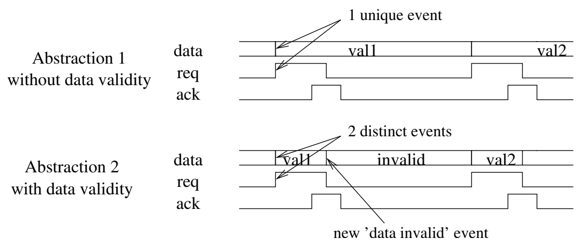

Fig. 2. Abstraction property example: Data validity

they can be used to automatically generate SystemC templates, network-on-chip interconnect for IP reuse and hardware-software cosimulation links. Our techniques are aimed at minimizing the amount of specifications required from the designer.

4.1 Automatic Translation Between Abstractions

Although we referred earlier to transactors as 'pieces of code', the aim is to perform translations across abstractions using the simplest possible specifications. For this reason, in order to automatically translate transactions between different levels of abstraction, each abstraction property is assigned a set of possible values. For example, data validity can be defined to be broad, early or late. These values are packaged with the IPs that require their signals to be translated at or through the associated level of abstraction. The major translations associated to our TLM are RTL to CSP, CSP to RTL, CSP to PV and PV to CSP. Each major translation is made of a succession of minor translations corresponding to each abstraction property.

The RTL to CSP translation is the simplest one: from a full set of events issued from the RTL simulation, we only need to extract the data payload, the initial request indicating the start of the CSP function call (send or receive) and the final acknowledge indicating the return of the CSP call. This conversion is processed in three steps, each one of them disabling in turn one abstraction property defining the handshake protocol: first, if defined in the protocol in use, the Return To Zero phase is removed (all the events not part of the RTZ are passed to the next stage), then data validity is "removed" -in fact, it is set to a default state where the data becomes valid with the incoming request (for a push channel) and never becomes invalid. Finally, the rest of the protocol is removed, leaving just enough information to generate the CSP call.

CSP to RTL, the opposite translation, is a bit more complex as information is added during the conversion process: instead of being disabled, abstraction properties are enabled and assigned real values. At the CSP level, the initial request, data payload and final acknowledge are known. From the definition of the handshake protocol, a template specifies how to rebuild the data signals at the wire level. All missing signals are introduced between the initial request and the final acknowledge in three consecutive step (Fig 3). In future work, timing constraints will be used at this stage to insert appropriate delays between constrained signals. For the moment, delays are hard-coded.

CSP to PV and PV to CSP translations are more complex due to the need to map abstract functionalities to channels, and the need to specify the interleaving of channel events.They must be handled by transactors specified in a language featuring these characteristics.

4.2 CSP-Based Transactors

When invoked from the PV level, the task of a transactor is to specify the sequence, parallelism and interleaving of the different channels involved in a transaction. Although destined to other uses, a few freely available languages have been designed in the past to describe such dependencies: TAST-CHP [8], CH and Balsa [9]. They are all CSP-based, which is ideal for PV to CSP translations. Moreover, as they were designed for synthesis, the syntax and grammar they use to describe channel interleavings is precise down to the RTL level. Balsa was created for asynchronous design and has the advantages of being synthesisable and being a traditional, easy-to-learn, Verilog-like imperative language.

The interface of a PV-CSP transactor needs to combine the interfaces of the CSP and PV levels. PV level is represented by the signature of the function call corresponding to the transaction. CSP level is represented by the channels involved in the transaction. As a CSP-based language, Balsa is able to describe the CSP, but not the PV interface. Transactor interfaces are therefore specified by prefixing a C-style PV function interface to a Balsa block (as shown in the code example from Section 5 Step 3). In practice however, the Balsa CSP interface of a transactor can be generated from an earlier specification of the channels (Section 5 Step 2), eliminating the need to be repeated in the transactor.

The current description language for a transactor's body is only a subset of the Balsa language. There are two reasons for this: although the Balsa synthesiser can generate structural Verilog for any language construct, our generation of behavioural SystemC code is new and we have only implemented the constructs that were useful in our applications. The second reason concerns the invertibility of the language.

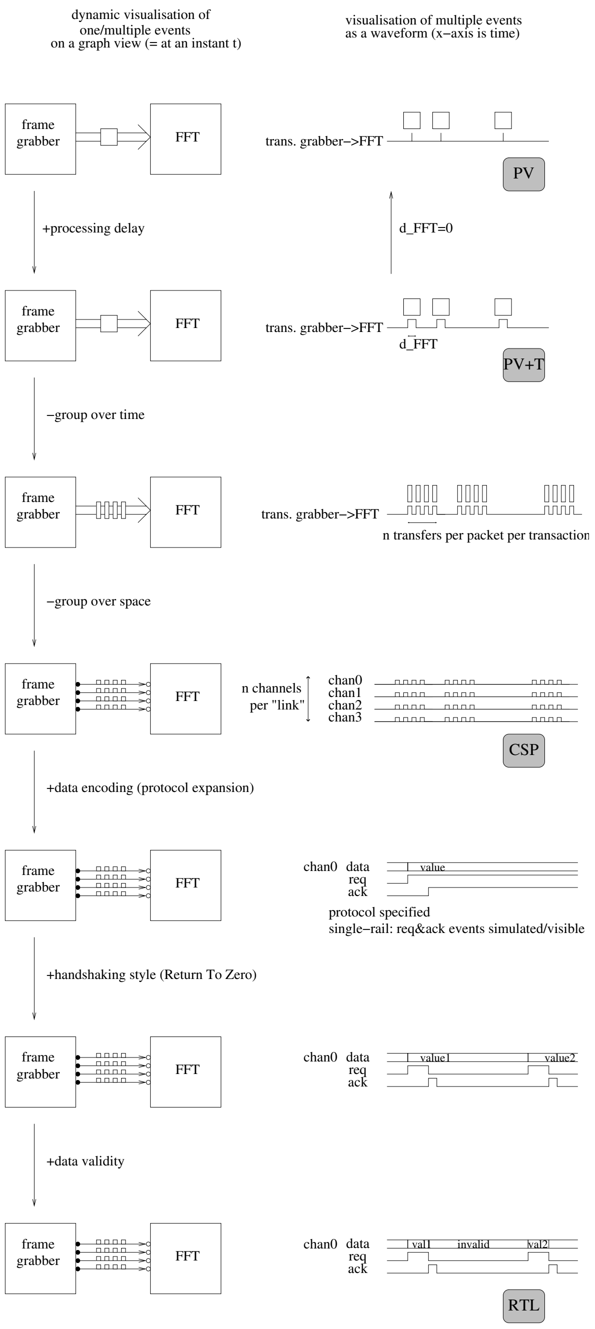
dynamic visualisation of one/multiple events on a graph view (= at an instant t)
visualisation of multiple events as a waveform (x−axis is time)
frame grabber
FFT
trans. grabber−>FFT
PV
+processing delay
d_FFT=0
d_FFT
PV+T
−group over time
n transfers per packet per transaction
−group over space
n channels per "link"
chan0
chan1
chan2
chan3
CSP
+data encoding (protocol expansion)
data
req
ack
value
protocol specified
single−rail: req&ack events simulated/visible
+handshaking style (Return To Zero)
value1
value2
+data validity
val1
invalid
val2
RTL

Fig. 3. From PV transaction to RTL signals

Most of the time, both PV to CSP and CSP to PV transactors are needed, and therefore two "reciprocal" Balsa descriptions need to be written (such as the two descriptions shown in Section 5 Step 3). In order to reduce the amount of work for the designer, one tool automatically generates a transactor's reciprocal description from its description.

CSP transactors enable a variety of automated code generation related to IP block interfaces. We describe three of them: Translator code generators, network on chip adapters and hardware-software links.

Translator code generators. In order to achieve the best simulation speed, the Balsa-described transactors are automatically converted to the same language as one of their attached components. If the chosen component is described in Balsa (component at CSP level), a direct Balsa simulation of component+transactor can be used. If the component uses Verilog (RTL level), a Verilog transactor is synthesised from the Balsa transactor using the Balsa synthesis tools. Finally, if the component uses SystemC (PV or CSP level), behavioural SystemC code is generated from the Balsa transactor. The main difficulty of this generator is also its main feature: fine-grained threads.

The Balsa language allows the easy description of very fine-grained threads, such as blocks containing a single assignment. On the other hand, threads in SystemC require a lot of "spaghetti coding", as each thread needs to be implemented in a separate class method. Automatically generating this kind of code from Balsa allows a clear and concise description of parallel code, which is often needed in asynchronous transactors.

Network on chip Plug&Play. One of the main motivations of System Level Design is IP reuse: describing an IP than can be integrated without any modification in other environments. OCP-IP defined a set of interfaces abstracting bus interfaces. IPs adopting these interfaces can then be connected easily to any bus. The downside with this method is that the designer is constrained to use these specific interfaces.

Asynchronous transactors are able to convert any CSP interface defined by the designer to a high-level transaction assimilable to a datas tructure. This means that transactors are able to convert CSP transactions to flows of bits. These flows can be transferred over any serial bus. In particular, the Balsa transactors can be used as adapters (serialisers and deserialisers) to automatically connect the IP to a network on chip.

Hardware-software link. The serialisation feature of the transactors can also be exploited for hardware-software links. The first application is to connect an emulator board to a software simulator via a serial link. On the hardware side (emulator board), transactors are synthesised in the same way as if they were connecting the IP to a hardware NoC. On the software side, transactors are simulated in the same way as if they were connected to a software NoC model. This configuration relies on a serial link connection between the simulator and the board.

The second application is for hardware-software (HW-SW) co-design. The automatic serialisation of the HW-SW interfaces would greatly facilitate the exploration of various HW-SW partitions.

4.3 Interface and Code Template Generators

After having described transactors, we realised that an important problem of asynchronous IP design could be solved: the complexity of writing SystemC code. Not only SystemC templates can be generated from the description of the interfaces at various levels of abstraction, but also transactors make it possible to automatically deduce an interface from its description at another level of abstraction. A step by step methodology (such as the one used in the case study presented in Section 5) shows that we can generate a complete SystemC template from interface descriptions at the PV and CSP levels, the RTL interface being deduced from CSP interface associated to the CSP to RTL abstraction properties, and large code templates being built from the transactor descriptions.

Multiple languages are used in a typical flow: SystemC at PV and CSP levels, Balsa atCSPlevelandVerilogatRTLlevel.Theycanbesynthesisedtolowerlevelsforsystem implementation on a chip or FPGA prototyping.

5 RAM IP Demonstrator

Traditionnally, synchronous transactors have been used to interface IPs with buses or NoCs using complex protocol and routing strategies. The asynchronous transactors presented in this paper are aimed more generically at linking any IP together. Some of these IPs may be busses or Nocs, but this is not compulsory, and pipelines of computational IPs can be built with asynchronous transactors. This section presents a simple example based on accesses to a RAM. This example, although lacking complexity, already implies a large amount of automatically generated code.

The full source code, interface descriptions and tools can be found in [10].

Step 1: high-level description of the RAM IP with test harness
From a description of the RAM interface at the PV level, we can generate a SystemC template that is easy to start a project with. A typical software-level RAM interface is:

```
module RAM { void write (int addr, int val); int read (int addr); }
```

Two classes, RAM and TestHarness, are automatically generated with a top-level sc_main function. This function instantiates the two components, connects them together and starts the SystemC simulation.

Before being able to simulate something useful, the behaviour of the RAM and test harness needs to be specified: the two method declarations (read and write) can be filled in and, in this particular case, a class variable for the memory array is declared:

```
constant MEMORY_SIZE 256;
int memory[MEMORY_SIZE];

virtual void write(int addr, int val) {
        cout << "RAM received write ("
                << addr << ", "
                << val << ")\n";
        memory[addr] = val;
}
```

```
virtual int read(int addr) {
        cout << "RAM received read ("
                << addr << ")\n";
        return memory[addr];
}
```

Writing the test harness is also easy, simply by filling up the run method of the test_harness class:

```
void run() {
      // use abstract interface to communicate with RAM
      for (int i=0; i<MEMORY_SIZE; i++)
            link_to_RAM->write (i, i+10);
      cout << "Processor read at address 2 => " << link_to_RAM->read(2) << eol;
}
```

A minimum amount of coding results in the simulation of our RAM IP. A good achievement for whoever tried to do this with SystemC before.

Step 2: Refinement of the interface to the CSP level
Here we enter the asynchronous domain. The data type of each channel needs to be described. The description format of the interface is now based on the Balsa language. The CSP implementation of the RAM is based on four channels: a ‘command’ channel, which indicates whether a write or a read command is issued, and the self-explicit channels ‘address’, ‘value_write’ and ‘value_read’:

```
enum command_type is CMD_WRITE, CMD_READ end
input command:      command_type;
input address:      int;
input value_write:  int;
output value_read:  int;
```

In the same way as at the PV level, SystemC code at the CSP level can be generated from this interface, for both the RAM IP and a test harness. The RAM class is now defined with SystemC sc_port ports publishing a CSP interface:

```
template <class T>
class CSP_send_if :
virtual public sc_interface {
 public:
  virtual void send(T) = 0;
  virtual bool probe() = 0;
};

template <class T>
class CSP_receive_if :
virtual public sc_interface {
 public:
  virtual void receive(T&) = 0;
  virtual bool probe() = 0;
};
```

```
class RAM : public sc_module {
 public:
      sc_port <CSP_receive_if
            <command_type> > command;
      sc_port <CSP_receive_if <int> > address;
      sc_port <CSP_receive_if <int> > value_write;
      sc_port <CSP_send_if <int> > value_read;

      // Constructor
      SC_CTOR(RAM) { SC_THREAD(run); }

      void run() { }
};
```

At the PV level, the simulation was based on a single thread (the test harness) and standard function calls. At the CSP level, each instantiated module is a thread, thus modeling hardware more closely. The two PV methods ‘read’ and ‘write’ are replaced by a single ‘run’ thread method. A typical implementation of the main method waits for a command to be received, and completes a write operation if the command is CMD_WRITE or a read operation if it is CMD_READ: (this code is user-written here, but we will see later that it can be generated automatically).

```
void run() {
   for (;;) { // loop forever
      enum command_type cmd;
      int addr, val;

      command->receive(cmd);
      address->receive(addr);

      switch (cmd) {
      case CMD_WRITE:
         value_write->receive(val);
         cout << "RAM received write ("
            << address << ", " << val << ")\n';
         memory[addr] = val;
         break;

      case CMD_READ:
         cout << "RAM received read ("
               << addr << ")\n";
         value_read->send (memory[addr]);
         break;
} } }
```

The test harness is described at the CSP level in the same way as the RAM ‘run’ method. However, another possibility is to reuse our previously written test harness at the PV level and rely on a PV to CSP converter. This is the role of the transactor.

Step 3: Description of the transactor
The following transactor is able to perform the translation between PV’s read and write calls and CSP channels’ send and receive methods.

```
void write (int addr, int val):
{
      command <- CMD_WRITE ||
      address <- addr ||
      value_write <- val
}
```

```
int read (int addr):
{
      command <- CMD_READ ||
      address <- addr ||
      return <-value_read
}
```

This transactor is obviously able to translate PV read and write calls to CSP send and receive calls. This allows us to connect the PV test harness to the CSP RAM.

What is really interesting is that, from this PV to CSP transactor, it is possible to generate most of the CSP level implementation of the RAM automatically. In our example, the Balsa reciprocal transactor (CSP to PV) is generated as below:

```
select command, address then
      case command in
            CMD_WRITE:
                  select value then
                        extern write (address, value_write)
                  end
            CMD_READ:
                  value_read <- extern read (address)
      end
end
```

The SystemC code generated from this new transactor is exactly the ‘run’ method implemented by hand previously.

Step 4: Refinement/synthesis to RTL
In order to go from CSP to RTL level, the handshake protocol used by each channel needs to be specified. A possible description is as follow (using multiple protocols):

command:	push	dual-rail	2-phase
address:	push	1-of-4	4-phase
value_write:	push	single-rail	4-phase
value_read:	push	dual-rail	4-phase

From these specifications, an RTL-level SystemC interface can be generated. An RTL-level 'run' method can also be automatically generated. However, using SystemC at the RTL level involves some efforts in order to interleave the handshakes appropriately and the designer ends up with a non-synthesisable RTL description. This task can be done a lot more easily by using a language such as Balsa to synthesise the RTL automatically from the CSP-style description. Another useful alternative is to use a HDL such as Verilog for the description at the RTL level. This is the path explored for this application: we decided to target an FPGA and use its RAM structures to implement the RAM IP. This allows us, in the future, to access the FPGA's RAM structures from any language, even if this language's synthesis tools would not automatically use these on-board RAMs. This is for instance the case with the Balsa language.

The Verilog interface template is generated from the IP interface specifications. Even more helpful, a template of the Verilog implementation can be generated from the transactor description: this is a synthesised version of the Balsa "CSP to PV" transactor, preceded by protocol converters able to adapt each channel's RTL signals to the CSP input of the transactor. Although it does not result in the most efficient implementation, this generated template is ideal to start the RTL level design and can be optimised by hand if necessary. The template automatically takes care of sending requests on outgoing channels and waiting for incoming requests on incoming channels. The only part to be filled in is the handling of the read and write commands, but this time in Verilog at the RTL level.

6 Conclusions

We identified the PV and CSP levels as a set of asynchronous abstractions suitable for asynchronous transaction level modeling above RTL. Based on these models, we showed that asynchronous CSP-based transactors can bring many more benefits than their synchronous counterparts, while being simple to describe. We showed how they can be used to automatically generate complex SystemC templates and hardware-software links, and automatically build network-on-chip interfaces facilitating IP reuse. Both CSP to PV and PV to CSP translations were generated from a unique transactor description. In order to do this, we were able to make a useful subset of the Balsa language invertible. Tools were developed after the techniques described in this paper, and a case study demonstrated the automatic generation of various complex parts of the design and the minimum amount of specifications required from the designer.

References

[1] Moore, S., Taylor, G., Robinson, P., Mullins, R.: Point to Point GALS Interconnect. In: Eighth International Symposium on Asynchronus Circuits and Systems (ASYNC'02) (2002)
[2] Bainbridge, W.J., Furber, S.B.: CHAIN: A Delay Insensitive CHip Area INterconnect. IEEE Micro special issue on Design and Test of System on Chip 22(5), 16–23 (2002)
[3] Parker, D.: Transactor generation using the CY language. SpiraTech Limited
[4] TransactorWizard, http://www.sdvinc.com
[5] Balarin, F., Passerone, R.: Functional Verification Methodology Based on Formal Interface Specification and Transactor Generation. Design, Automation and Test in Europe (2006)
[6] Property Specification Language: Reference Manual http://www.accellera.org/pslv101.pdf
[7] Hoare, C.A.R.: Communicating sequential processes. Comms of the ACM 21(8) (1978)
[8] Renaudin, M., Rigaud, J.B., et al.: TAST CAD Tools." ASYNC'02 Tutorial (2002)
[9] Edwards, D., Bardsley, A.: Balsa: An Asynchronous Hardware Synthesis Language. The Computer Journal 45(1), 12–18
[10] Balsamics, http://www.cs.manchester.ac.uk/apt/projects/tools/balsamics/transactors
[11] Open Core Protocol International Partnership, http://www.ocpip.org