Soft Intellectual Properties (IPs) integration for System On Chip (SOC) design

Fateh Boutekkouk

1Department of Mathematics and Computer Science, University of Oum El Bouaghi, 04000, Algeria
fateh_boutekkouk@yahoo.fr

Abstract—This work deals with incompatible soft intellectual properties (IPs) integration for System On Chip (SOC) design. IPs Structural and dynamic aspects are modeled via UML 2.x diagrams such as structural, timing and Statecharts diagrams. From these diagrams, interfaces are generated automatically following an interface generation algorithm. Interfaces behaviors verification is performed by the model checker that is integrated in Maude language. A Maude specification including interface specification and properties are generated automatically from UML diagrams.

Index Terms—SOC, IP, Integration, UML, Maude, Formal verification

I. INTRODUCTION

Nowadays, Systems On Chip (SOC) [1] design is becoming more complex and may lead to the non satisfactory of customers requirements and the time to market constraints. To cope with this problem, it seems that Core Based Design (CBD) brings a significant improvement of design in general and to decrease the time to market window in particular [2]. The main idea behind the CBD is to reutilize existing hardware and/or software components with some customization and adaptation.

In the SOC field, designers have considered the reuse of complex hardware and software components (Intellectual Property or simply IP components), already used and tested in previous designs [2]. Reuse is essential to master the complexity of SOC design; however it does not come for free. Since most IPs are provided by different vendors, they have different interface schemes, data bit widths and operating frequencies, combining these components is an error-prone task. Designers have to find and evaluate IPs that fit particular needs and the selected IPs must be integrated together to implement the desired SOC functionality. This integration may require some adaptation and customization. The basic goal of an interface synthesis is to generate interfaces between incompatible components. For this reason, researchers in both academia and industry [3, 4, 7, 8, 9, 10] have developed many algorithms and CAD tools to explore, to optimize, and to generate interfaces between incompatible IPs. Unfortunately, most of these efforts target models and languages at lower levels of abstractions. Another problem is the fact that the generated interface may not work correctly. In this context, we have developed an UML 2.x tool that permits to both software and hardware SOC designers to model, configure, and link the incompatible IPs graphically. From UML diagrams, a set of FSMDs (Finite State Machine with Data path) modeling the interface are generated automatically. Our tool permits also formal specification generation of the interface in the Maude language [5]. The rest of this paper is organized as follows: section two is dedicated to related works concerning the synthesis of interface for incompatible protocols. Section three gives an overview of IPs and their classes Section four puts the light on Maude language. The algorithm of interface synthesis we have adopted is detailed in section five. Section six discusses the translation from UML to Maude. Our developed tool is presented in section seven before conclusion.

II. RELATED WORK

Here, we try to mention some pertinent works targeting interface generation. In [4], signal transition graph was introduced for protocol specification and the hardware interface is synthesized with asynchronous logic. In [7], the protocol specification is decomposed into basic operations, while the protocol is represented as an ordered set of relations whose execution is guarded by a condition or by a time delay. In [8], the two protocols are described using regular expressions and are translated into corresponding deterministic finite automata then interface protocol can be synthesized as an FSM by production computation algorithm. In [9, 10], a queue-based interface scheme was proposed. An algorithm which generates FSMD model for queue from timing specification of the given memory was developed. In order to generate the interface automatically, a formal model, called Protocol Sequence Graph (PSG) that captures the minimal necessary set of features representing the interface and its associated communication protocol. From given protocol specifications and clock period of the selected queue, the interface synthesis algorithm generates the FSMD for interface including the queue FSMD.

The main limitation of these approaches is that IPs communication protocols are expressed in low level models and/or programming languages such as waveforms, VHDL or C language. Another tendency to address the problem of IPs integration is the use of standards for promoting reuse in the design process. Several standards have been proposed. Among these, the Open Core Protocol (OCP) by OCP-IP [11] has gained wide industrial acceptance. However, for existing non OCP compliant IP cores, it is very expensive to customize them to comply with the OCP standard.

Our work tries to take advantages of the UML 2.x standard for IPs modeling and interface generation with minimal user inputs exploiting the algorithm proposed in [10]. In its basic
form, this algorithm was used to generate the glue logic between two incompatible IPs. Since the system may contain many incompatible IPs, we have to apply the same algorithm for each pair of communicating incompatible IPs. Our tool differs from others in:
1. The use of high level models for incompatible IPs integration and in particular the use of UML 2.x new diagrams like timing and structure diagrams.
2. IPs communication protocols are abstracted from any Hardware Description Language (HDL) and specified using UML Statecharts where actions are associated to states and expressed in the C language.
3. Our tool supports both communication protocols customization and automatic interface generation. The generated glue logic is an FSMD modeled via UML Statecharts including concurrent and hierarchic states.
4. Our tool permits formal specification generation for interfaces between IPs in the Maude language.

Our choice of Maude language is due to its expressivity, simplicity, simulation, and formal verification at different levels of abstraction [5].

III. INTELLECTUAL PROPERTY (IP)

An intellectual property or a virtual core (IP) [2] is a reusable software or hardware pre-designed block and maybe delivered by third party companies. Hardware IP components may come in several forms: hard, firm or soft. An IP is hard, when all its gates and interconnects are placed and routed. It has the advantage of more predictable estimations of performance, power, and area considering the target technology. But, it is less flexible and therefore less reusable. An IP can be soft, with only an RTL (Register Transfer Level) representation. It is available in source code and therefore adaptable to different platforms at the price of less predictable estimations on performance and area. An IP can be firm, with an RTL description together with some physical floor planning or placement.

IV. MAUDE LANGUAGE

Maude [5] is a specification and programming language based on the rewriting logic [6]. Two specifications level are defined in Maude. The first level concerns the system specification, while the second one carries on the properties specification. The system specification level is provided by the rewrite theory.

The rewriting logic was introduced by Meseguer [6]. This logic having a complete semantics unifies all the formal models that express concurrence. In rewriting logic, the logic formulas are called rewriting rules. They have the following form: $R: [t] \rightarrow [t']$ if $C$. Rule $R$ indicates that term $t$ is transformed into $t'$ if a certain condition $C$ is verified. Term represents a partial state of a global state $S$ of the described system. The modification of the global state $S$ of the system to another state $S'$ is realized by the parallel rewriting of one or more terms that express the partial states. The distributed state of a concurrent system is represented as a term whose sub-terms represent the different components of the concurrent state. Three types of modules are defined in Maude. Functional modules allow defining data types and their functions through equations theory. System modules define the dynamic behavior of a system. This type of modules extends functional modules by introducing rewriting rules. Finally, there are the object-oriented modules that can be reduced to system modules.

The property specification level defines the system properties to be verified. The system is described using a system module. By evaluating the set of states reachable from an initial state, the model-checking allows to verify a given property in a state or a set of states. The Model-checking supported by Maude’s platform essentially uses the LTL (Linear Temporal Logic) logic for its simplicity and the defined decision procedures it offers.

V. INTERFACE GENERATION ALGORITHM

In this section, we try to put the light on the synthesis algorithm for interface between incompatible protocols as proposed in [10]. In [9], the interface architecture is basically composed of synchronous system interfaces as shown in Figure 1. The system components (PE1 and PE2) may operate at different frequencies and at different data rates. The interface architecture includes a buffer (FIFO queue) to smoothen the burst data transfer requests and two FSMDs (Finite State Machine with Data path) to queue and un-queue data. In the interface architecture, system components (PE1 and PE2) in Figure 1 are directly connected to its corresponding state machines and will transfer data to other component through the state machines. The state machines are responsible for receiving (sending) data from (to) the corresponding system components and writing (reading) the data to (from) the queues. We have to consider two interface protocols, the protocol between state machines and queues and the protocol between system components and state machines. The interface protocol between state machines and queues will be fixed because the queue interface is predefined. But the interface protocol between system components and state machines will be varied depending on the protocol of system components.

The queue is implemented with a memory to store large amount of data. The clock period of the queue is frequently less than the memory read access time.

Generally, a queue contains memory to store data internally. The operation of the queue is determined by memory organization and timing [9]. In order to generate a queue model from the memory timing constraints, we have to schedule the timing constraints based on given clock period of the queue. Given timing constraints of the memory and the clock period of the queue, queue generation reduces to the task of generating a state machine that implements the given queue functionality and satisfies the timing constraints. This requires scheduling of memory timing constraints into clock cycles such that no constraint is violated. Therefore, the FSMD implementation selects instances of the given timing
ranges based on the granularity given by the queue clock. Finally the queue description will be generated for integration in interface synthesis. Figure 2 shows the block diagram of a queue with a single I/O port.

A. Interface generation algorithm

Problem definition

Given:
1. Protocol descriptions of two communicating parties (producer and consumer).
2. Bit width and size for the selected memory.
3. Clock period $T_{Q\text{clk}}$ of the queue.

Determine:
1. FSMDs for state machines.
2. FSMD for the queue.

Conditions: Timing constraints are met.

Algorithm of the figure 3 shows the interface synthesis algorithm from given protocol specifications and clock period of the selected queue. We have applied the same algorithm as proposed in [10] but with two major modifications: firstly, the scheduling of actions over states is performed by the designer thus the $\text{Schedule}()$ function is removed from the algorithm. Secondly, the $\text{Make\_Dual}()$ function transforms the Statechart (instead of the Protocol sequence graph PSG) of the original protocol specification to the corresponding dual Statechart, which can be done by replacing the operators in actions with their duals.

The method $\text{Generate\_Queue}()$ will generate the queue. The generated producer interface FSMD, consumer interface FSMD and queue interface FSMD should be collapsed into a single FSMD to obtain interface FSMD. The method $\text{Add\_FSMD}()$ will collapse the producer and queue interface FSMDs into the transducer interface FSMD for the producer (FSMDT), and the consumer and queue interface FSMDs into the transducer interface FSMD for the consumer. Finally we have two FSMDs for transducer: the producer interface FSMD and the consumer interface FSMD in the transducer. For more details on this algorithm, one can refer to [10].

Algorithm $\text{Generate\_Interface} \quad (\text{FSMD}_P, \text{FSMD}_R, T_{Q\text{clk}})$

$\text{FSMD}_Q = \text{Generate\_Queue}(T_{Q\text{clk}}); \quad // \text{generate Queue FSMD}$

$\text{FSMD}_S = \text{Make\_Dual}(\text{FSMD}_P); \quad // \text{generate the dual of producer}$

$\text{FSMD}_R = \text{Make\_Dual}(\text{FSMD}_R); \quad // \text{generate the dual of consumer}$

$\text{FSMD}_D = \text{Make\_Dual}(\text{FSMD}_Q); \quad // \text{generate the dual of queue}$

$\text{For} \ i=1 \ \text{to} \ \ (\text{bw}_P / \text{bw}_R) \ \text{do}$

$\text{Add\_FSMD}(\text{FSMD}_{TS}, \text{FSMD}_S); \quad // \text{add the producer FSMD to interface FSMD}$

$\text{End for}$

$\text{For} \ i=1 \ \text{to} \ \ (\text{bw}_R / \text{bw}_P) \ \text{do}$

$\text{Add\_FSMD}(\text{FSMD}_{TS}, \text{FSMD}_R); \quad // \text{add the consumer FSMD to interface FSMD}$

$\text{End for}$

VI. PASSAGE FROM UML TO MAUDE

A. Translation of static aspects

As an example of application, we have chosen three IPs that are: ColdFire processor, ARM9TDMI processor, and TMS320C50 DSP processor [10]. The objective is to generate and verify the interfaces between these three cores with incompatible communication protocols. For more detail on this example, one can refer to [10].

In this section we will explain the translation from UML to Maude specifications.

UML objects are specified as Maude objects (class instances), so for each IP, we declare a class with a set of attributes. Input/output interfaces (signals) are specified as Maude attributes. Signals generation or assignments are specified as Maude messages. In the below example, we declare an object called $Pr$.

$\langle Pr : \text{Arm9tdmi} \mid DA : \text{addr}, state : sl, DD : \text{data}, DnRW : M, DDEN : N, nWAIT : X \rangle$

$Pr$ is an instance of the class $\text{Arm9tdmi}$ representing the $\text{Arm9tdmi}$ IP. In Maude, we declare the $\text{Arm9tdmi}$ class as follows:

$\text{op Arm9tdmi : \rightarrow Cid \ [ctor]}.$

$Pr$ has five signals that are $\text{DA}, \text{DD}, \text{DnRW}, \text{DDEN}$, and $\text{nWait}$. In Maude, we declare a signal as follows:

$\text{op DA : Int \rightarrow Attribute \ [ctor \ gather \ (&)]}.$

We add a new attribute called $\text{state}$ to specify the current state of the IP. For this we declare a new sort (type) called $\text{statevalues}$. Possible values for this type are all possible states of the IP. In Maude, we write:

$\text{sort statevalues}.$

$\text{ops s1 s2 s3 s4 s5 : \rightarrow statevalues}.$

Similarly, we specify a FSMD for instance as:
B. Translation of dynamic aspects

FSMD transitions are specified as Maude rewriting rules. On a transition, we can find incoming/outgoing events that correspond to signals reading/writing. Such events are specified in Maude as messages. Here is an example of a rewriting rule:

```
rl [rl1] :
  signaldone(Pr, A)
signalmask(Pr, G)
signalinport(Pr, data)
<Pr : Arm9tdmi | state : e1, inport : data, mask : G, done : A >
=> signalDD(Pr, 5) signalDA(Pr, 1000) signalDnRW(Pr, 1)
```

This unconditional rule enables the IP Pr to transit from the state e1 to the state s1 and to modify the values of signals DA, DD, DnRW, and DDEN. signaldone(Pr, A) is a message that specifies an input event (the value of signaldone is true). signalDD(Pr, 5) is a message that specifies an output event (we assign the value 5 to signal signalDD).

Table 1 shows the correspondence between UML and Maude constructs.

<table>
<thead>
<tr>
<th>UML</th>
<th>Maude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite object</td>
<td>System module</td>
</tr>
<tr>
<td>Simple object</td>
<td>Object (Class instance)</td>
</tr>
<tr>
<td>State diagram transition</td>
<td>Rewriting rule</td>
</tr>
<tr>
<td>Required/Provided interfaces</td>
<td>Attributes</td>
</tr>
<tr>
<td>Hierarchic state</td>
<td>Flat specification</td>
</tr>
<tr>
<td>Concurrent state</td>
<td>A set of rewriting rules</td>
</tr>
<tr>
<td>Input/Output events on transitions</td>
<td>Messages</td>
</tr>
</tbody>
</table>

C. Specification of the interface properties

Two properties are defined: the `begin` property that specifies the fact that the interface FSMDs are in their beginning states. The `end` property specifies the fact that the interface FSMDs are in their end states. In Maude, we declare these two properties as follows:

```
op begin : Configuration Configuration Configuration Configuration -> Prop .
op end : Configuration Configuration Configuration Configuration -> Prop .
```

Bellow, the definition of the `end` property in Maude:

```
ceg < Pro : Arm9tdmi | state : S, done : A, outport : G1 > Cf
```

We want to verify reachability: do interface FSMDs always reach end states from begin states?

In Maude, we specify such query as follows:

```
[ ] begin(Initial1,Initial2,Initai3,Initial1) ->
end(Final1,Final2,Final3,Final4)
```

We perform formal verification, we call the ModelChecker function as follows:

```
"ModelCheck( [ ] begin(Initial1,Initial2,Initai3,Initial1) ->
end(Final1,Final2,Final3,Final4)"
```

VII. PRESENTATION OF OUR TOOL

We have developed a tool that supports UML 2.x modeling of IPs using Structure, Timing, and Statecharts diagrams. Our tool generates automatically the interface FSMD including the Queue FSMD for each pair of communicating IPs. From Memory Timing diagram, some temporal parameters are extracted to be used for FSMD queue generation. In our case, each IP is modeled via UML 2.x components with required (input) and provides (output) signals. Furthermore, each IP is parameterized by some parameters such as the HDL (e.g. VHDL, Verilog), the clock period, the power consumption, and the abstraction level of each IP. Some of these parameters (Power consumption) are not used in the algorithm of interface synthesis, rather than, we will use them to perform power estimation in our future work. In our case and regardless of the IP HDL, we assume that all IPs communication protocols actions are specified via UML Statecharts. The communication protocols actions are expressed in the C language. Figure 4 shows IPs modeling and configuration of its I/O signals. Figure 5 shows IPs modeling and connections between them. Figure 6 shows timing parameters modeling of memory read/write cycles.

Figure 7 shows IP internal behavior modeling. Figure 8 shows the interface FSMD. The latter is generated automatically. Figures 9 and 10 show the generated Maude code for the interface and the manipulation of properties to be verified respectively. Properties code is introduced by the designer and it is automatically added to the interface code to finally generate one Maude file including both interface code and properties. Figure 11 shows the result of model checker for the reachability property between the two IPs. ARM9TDMI and ColdFire processors. According to the model checker result, we can state that the reachability property is verified.
VIII. Conclusion and Perspectives

In this work, we presented our tool for IPs modeling using UML 2.x diagrams and interface formal verification between incompatible IPs using Maude language. We have exploited UML 2.x diagrams such as structure diagram for modeling and connection between IPs, timing diagram to model memory read and write operations timing constraints, and Statecharts with hierarchic and concurrent states to model IPs communication protocols and Queue behavior. Formal verification is performed by calling the model checker which is integrated in the Maude language. As a perspective, we plan to discover more properties for verification and to perform performance estimation on the generated interface.

REFERENCES