Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Joint software/hardware modeling with FXML/Jahuel

I. Assayad, F.X. Defaut, S. Yovine, M. Zanconi

Verimag Research Report n^o TR-2007-13

Septembre 2007

Joint software/hardware modeling with FXML/Jahuel

I. Assayad, F.X. Defaut, S. Yovine, M. Zanconi

VERIMAG, Centre Equation, 2 av. de Vignate, 38610 Gières, France

Septembre 2007

Abstract

This report presents an extension of FXML-JAHUEL for modeling hardware components. It illustrates how to jointly model software and hardware components with a simple producer/-consumer application on a bi-processor. It briefly discusses the FXML-to-PWARE translation to enable performance-aware simulation.

Keywords:

Reviewers:

Notes: Partially supported by projects MEDEA+ NEVA and Minalogic SCEPTRE.

How to cite this report:

@techreport { ,
title = { Joint software/hardware modeling with FXML/Jahuel},
authors = { I. Assayad, F.X. Defaut, S. Yovine, M. Zanconi},
institution = { Verimag Research Report },
number = {TR-2007-13},
year = { },
note = { }
}

```c
#pragma code_block
#pragma parallel writer user
main()
{
  writer(); /* /&/ */ user();
}

#pragma code_block
void writer ()
{
  while (1) { write(); }
}

#pragma code_block
void user ()
{
#pragma period 15 us
  while (1) { use(); }
}

int x = 0;

#pragma code_block
#pragma execution_time [0,5] us
void write()
{
  x++;
  printf("WRITER : x = %d\n", x);
}

#pragma code_block
#pragma execution_time [0,10] us
#pragma dependency
main.writer.write -> (x) main.user.use [0,100] us
void use()
{
  printf("USER : x = %d\n", x);
}
```

```xml
<var-list> ... </var-list>
<pnode-sum>
 <dep-list> ... </dep-list>
  <body>
  <!-- Producer -->
  <while>
   <label>Producer</label>
   <condition>
     <b-exp>
       <boolean default-value="true"/>
     </b-exp>
   </condition>
   <period>
    <time>10</time>
    <time-unit>us</time-unit>
   </period>
   <body>
      ...
      <source-code>
       _function_write();
       </source-code>
   </body>
  </while>
  <!-- Consumer -->
    ...
</pnode-sum>
```

FIG. 1 – Simple producer/consumer.

1 Software annotations using FXML

The underlying basic idea of our language, called FXML, is that computation units are concurrent by default, while explicit precedences can be expressed to limit concurrency. The granularity of computation units is not fixed, the smaller grain is the assignment or legacy code. Data dependencies are implicit, but can be explicitely added to express data dependencies in legacy code.

FXML provides a `forall` primitive to declare several concurrent iterations of the same block. This construct is similar to FORTRAN 95 with the difference that we do allow dependencies between iterations. FXML also has "parallel" and "sequential" composition.

An important difference with other languages is that basic FXML does not provide any specific synchronization or communication primitives (like channels or rendez-vous). Instead, the basic language can be extended with non-functional information about the concrete execution model (e.g., execution times, synchronization mechanisms, number of processors), and the target platform (e.g., OpenMP, Pthreads, MPI).

1.1 A simple producer/consumer

Let us start with a simple producer-consumer system to informally introduce FXML. Fig. 1 (left) shows the C program of this system, where pragmas are annotations. The abstract syntax of FXML will be given later in this section. The concrete syntax of FXML is defined as an XML schema, which is not presented here.

The pragma `parallel writer user` declares that the C functions `writer()` and `user()` invoked by `main()` are logically concurrent. The abstract syntax symbol for `parallel` is `/&/`. Functions `writer()` and `user()` are non-terminating executions, `user()` has a period of 15μs (pragma `period`). `writer()` calls `write()` which has an execution time less or equal than 5μs. `user()` calls `use()` which completes in at most 10μs. `dependency` expresses that there is a data dependency between `write()` and `use()` on `x`, with a *freshness* interval $[0, 100]$, that is, `use()` can only take place if the time elapsed since the last `write()` is less than 100μs.

A C compiler that analyzes the program (pragmas and C code) and understands the pragmas, would be able to extract a description of it in the concrete XML syntax of FXML. Fig. 1 (right) shows the XML representation of FXML.

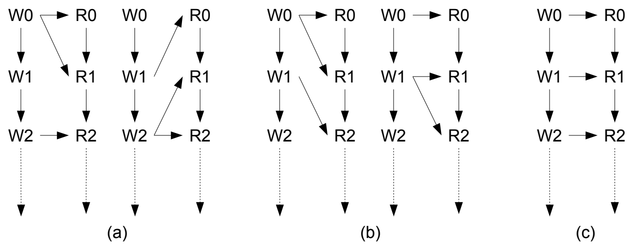

FIG. 2 – Examples of executions.

1.2 Parallel software specification in FXML

The *body* of an FXML specification is composed of blocks called *pnodes*. The basic *pnode* types are *assignments*, variable *declarations*, e.g., `var int x`, or *legacy code*, e.g., `{# p = 0 #}`. Basic *pnodes* are executed *atomically*. Legacy declarations can be used to encapsulate either pre-existing or newly developed "hazard-free" (e.g., without system calls) code, which can be safely compiled with an optimizing compiler. *Tags* can be used to provide summaries of legacy code in order to highlight dependencies hidden inside legacy declarations. For instance, the *tag* `</ p :w />`, states that variable `p` is written by the legacy code. *Pnodes* can be labeled : e.g., `W : {# x = p++ #}`. *Pnodes* inside `while` (and `for`) loops are automatically indexed. The semantics of the producer's `while` loop is a sequence $W_0 W_1 \ldots$, where W_i is the i-th occurrence of the assignment labeled `W`.

The statement `dep W` $\rightarrow$ `R` specifies a *dependency* between occurrences of *pnodes* labelled `W` and `R`. Notice that variables `x` and `p` are declared in FXML, but only used in legacy C code. This declaration, together with the annotations about variable usage, allows the compilation chain to eventually synthesize the data dependency `dep W` $\rightarrow$ `R`, if not explicitly declared, even if it is hidden in the legacy code. The arrow $\rightarrow$ means that variable `x` *must* be written at least once, before being read. This *weak* semantics can be further constrained : `W [strong]`$\rightarrow$ `R` means that *every* value of `x` *must* be read at least once (no losses). Dependencies can also be specified to relate specific iterations, e.g., `W` $(i, i) \rightarrow$ `R` specifies that the value of `x` read at the i-th iteration of the `Consumer`'s while loop, is the one written by the i-th iteration of the `Producer`'s loop. In general, indexed dependencies have the form $p\ (i, f(i)) \rightarrow q$, where $f(i)$ is affine.

The semantics of a *pnode* p, denoted $[\![p]\!]$, is a (possible infinite) *set* of partial orders, called *executions*. Fig. 2 shows examples of executions of the producer-consumer system for different types of dependencies between *pnodes* `W` and `R` : (a) *weak*, (b) *strong*, and (c) (i, i). Each execution of the composed system contains the union of the executions (in this case, total orders) of the `Producer` and `Consumer` *pnodes*, namely, $W_0 W_1 \ldots$, and $R_0 R_1 \ldots$, resp., with precedences added by the dependency declaration `dep W` $\rightarrow$ `R`. Notice that, the (i, i) dependency results in a single execution.

The semantics of FXML [1] consists of partial orders consistent with the *conjunction* of constraints imposed by dependencies. This allows, for instance, specifying the case of a (consumer-like) *pnode* C which computes, say $y = f(x_1, \ldots, x_n)$, for x_i written by (producer-like) *pnodes* P_i, $i \in [1, n]$. Another case consists of a *pnode* P *broadcasting* the value of a variable x to several consumers C_i, computing $y_i = f_i(x)$, $i \in [1, n]$. The conjunctive semantics does not allow, for instance, easily capturing the case where the value of variable x written by P is to be read by *a single* non-deterministically chosen consumer. The other case where a single consumer C needs the value produced by *any* of many producers $P_1 \ldots P_n$ is not easily specified, either.

To overcome this inconvenience, we have extended FXML with *hyper-dependencies* of the form $P_1 \ldots P_n\{\phi\} \rightarrow C$, and $P\{\phi\} \rightarrow C_1 \ldots C_n$, where ϕ specifies the composition of the individual dependencies $P_i \rightarrow C$ and $P \rightarrow C_i$. For simplicity, we only consider here the case where ϕ is the *exclusive disjunction* of the dependencies, and restrict individual depedencies to be *weak* or *strong*.

2 The compilation chain : Jahuel

Jahuel is a FXML-based prototype compilation chain. Compiling a FXML specification consists in transforming it until actual executable code for a specific platform could be generated. Let $\mathcal{L}$ denote a language. Concretely, $\mathcal{L}$ is given by an XML schema, where each element definition has an associated type.

A *transformation* from $\mathcal{L}$ to $\mathcal{L}'$ is an injective map $\phi : \mathcal{L} \to \mathcal{L}'$, that is, every element of the XML schema $\mathcal{L}$ is in the set of elements $\mathcal{L}'$. Let $E_{\mathcal{L}}$ be the set of executions of type $\mathcal{L}$, and $F_\phi : E_{\mathcal{L}'} \to E_{\mathcal{L}}$ be the "forgetting" function that forgets any information specific to executions of type $\mathcal{L}'$. $\phi : \mathcal{L} \to \mathcal{L}'$ satisfies that for all executions $e' \models_{\mathcal{L}'} \phi(p)$ it follows that $F_\phi(e') \models_{\mathcal{L}} p$.

The compilation process is a sequence of transformations $\mathcal{L}_0 \mapsto^* \mathcal{L}_0 \mapsto \mathcal{L}_1 \mapsto^* \ldots \mathcal{L}_n$, where $\mathcal{L}_0$ is basic FXML. $\mathcal{L}_i \mapsto^* \mathcal{L}_i$ is a sequence of transformations from $\mathcal{L}_i$ to $\mathcal{L}_i$, resulting in a sequence of programs $p_i^1 \ldots p_i^n$, such that $[\![p_i^{k+1}]\!] \subseteq [\![p_i^k]\!]$. An example of a transformation from $\mathcal{L}_0$ to $\mathcal{L}_0$ consists in replacing *weak* dependencies by *strong* ones. $\mathcal{L}_i \mapsto \mathcal{L}_{i+1}$ is a transformation that *adds* information not expressible in $\mathcal{L}_i$. An example consists in inserting communication and synchronization mechanisms (e.g., semaphores, queues, ...) to ensure dependencies are met.

JAHUEL is a FXML-based compilation framework, constructed to be easily extended to cope with new execution models, by extending the basic FXML XML-schema, and by adding transformations. JAHUEL is implemented in Java, using the Java Architecture for XML Binding API [1], to manipulate XML documents. JAHUEL provides some general transformations which can be customized for different execution and simulation platforms. Currently, it generates executable code for, e.g., Java, C with `pthreads`, and simulation code for P-Ware [2], a SystemC-based simulation platform, for jointly predicting and analyzing performance of software and hardware components generated by JAHUEL. The compilation chain is indeed to be instanciated with the sequence of transformations to be applied. Each transformation reads an input XML file and outputs another XML file to be used by the next one, thus ensuring traceability of implementation choices. The code generation phase for the target platform is done via a stylesheet.

3 Hardware modeling

3.1 Hardware support in FXML

FXML has been extended to support hardware modeling. A hardware architecture model is composed of architecture components, their connections and timed-level transaction behavior of the components. Components and connections are described in a textual XML-based format, an extension of FXML, whereas timed-level transaction behaviors are either meta-modelled P-WARE components or predefined C++ components included from P-WARE library. FXML has also been extended to allow specifying the mapping between software and hardware components. This part would be used to specify the application deployment, that is, software and data placements, locality issues, etc. The binding between architecture components and their behavior, as well as the actual deployment, is to be done using JAHUEL.

Fig. 3 shows the structure of FXML.

[1] http ://java.sun.com/developer/technicalArticles/WebServices/jaxb/

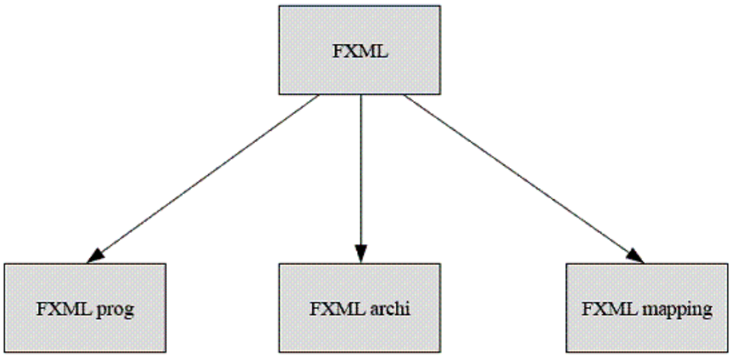

FIG. 3 – Structure of FXML for software/hardware modeling

The basic block of FXML "archi" is called *anode* for Architecture Node. An *anode* can be of the following types :

1. *Processor*, to represent processor components.
2. *Memory*, to model memory components.
3. *Bus*, to model generic bus components.

Each of these types comes with its own set of attributes, defined by the appropriate FXML schema, together with the attributes inherited from *anode*, such as, for instance, `Qin` and `Qout`. These two attributes define, respectively, the input and output interfaces of a component. The number of `Qin` and `Qout` instances may be null, and it is not bounded. Fig. 4 gives an overview of the FXML extension for modeling hardware architectures.

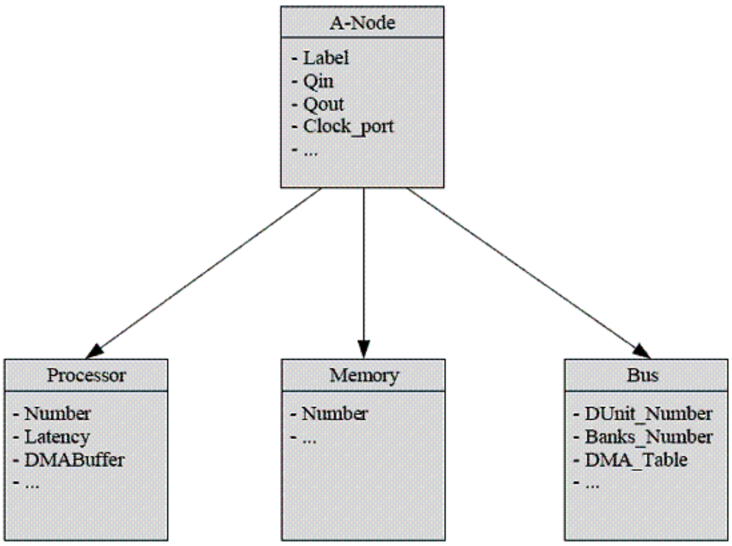

FIG. 4 – Part of FXML support for hardware specification (FXML archi)

3.2 Producer/consumer

A simple producer/consumer application on a two-processor architecture with a memory bank connected via a command and a data buses is shown in Fig. 5.

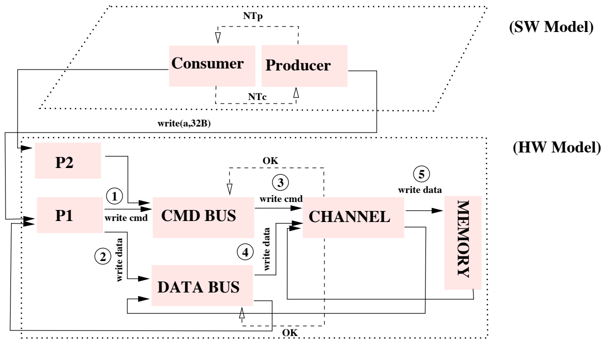

FIG. 5 – Producer/consumer application

The following listings show part of the hardware model in FXML "archi". As indicated by the usage of `qoutput` element, the first output port bus component named CHANNEL is bound to the first transaction request input queue of DMA component named DMA1.

Listing 1 – Bus label

```
<Bus><!-- CHANNEL definition -->
  <arch-label>CHANNEL</arch-label>
  <id>1</id>
  <qin>RBUS_BF</qin>
  <qoutput>
    <qout>DMA1</qout>
      <number>1</number>
  </qoutput>
  ...
  <clock-port>CLK</clock-port>
 </Bus>
  ...
```

Listing 2 – DMA label

```
<DMA><!-- DMA1 definition -->
  <arch-label>DMA1</arch-label>
  <id>1 </id>
  <qin>DMA1_BF1</qin>
  <qin>DMA1_BF2</qin>
  <qout>CHANNEL</qout>
  <clock-port>CLK</clock-port>
  <dma-dbuf-table name="DBUF1">
    <dma-dbuf>
      <src-id>1</src-id>
      <size>10</size>
      </dma-dbuf>
  </dma-dbuf-table>
 </DMA>
```

Listing 3 – Memory label

```
<memory><!-- MEMORY definition -->
<arch-label>MEMORY</arch-label>
        <id>1 </id>
        <qin>1</qin>
        <qout>CHANNEL</qout>
```

```
        <clock-port>CLK</clock-port>
</memory>
```

4 Integration in a design/ analysis/implementation flow

A proposal of integration of FXML/JAHUEL/P-WARE into a design, analysis, and implementation flow is schematically depicted in Fig. 6. This research axis will be further studied during the rest of the project.

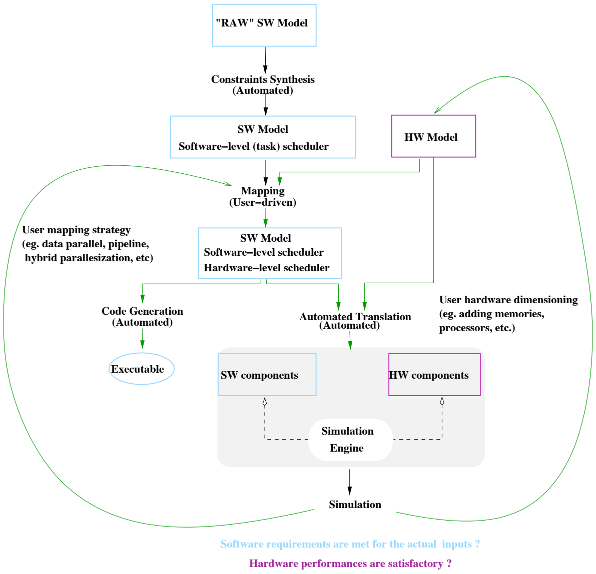

FIG. 6 – Design, analysis and implementation flow

Software and hardware models. The “raw” software model is the initial FXML specification obtained from code annotations. The model specifies parallelism and timing constraints corresponding to real-time requirements of software, that is, software tasks’ deadlines, and dependencies between software tasks and between these tasks and hardware. The hardware model is the FXML description of the hardware architecture.

Constraints synthesis. The goal of this phase is to synthesize a real-time software-level scheduler. This scheduler is hardware independent. However, it takes into account software interactions with hardware through dependencies.

The synthesized scheduler guarantees that timing requirements are met, assuming tasks’ execution times are respected. The execution times for these tasks will depend on their actual mapping and communication model, which are defined in the mapping step.

Mapping. The goal of this phase is to define a hardware-level scheduler of software. Therefore, the sofwatre graph is flattened considering all levels of hierarchy. This mapping is achieved using a choice between different types of mapping strategies, and is driven by the designer.

The following listing shows the mapping for the producer/consumer example.

Listing 4 – Part of Producer/Consumer mapping

```
<map-pnode>
 <node-label>Producer</node-label>
 <processor-label>P1</processor-label>
```

```
</map–pnode>
<map–pnode>
 <node−label>Consumer</node−label>
 <processor−label>P2</processor−label>
</map–pnode>
   ...
```

Translation to P-Ware. This phase achieves a translation of the hardware model into P-WARE C++ components, and a given mapping of the software model into software components.

Currently, for the software specification, two XML-based transformations are operated by JAHUEL.

– The *componentization* phase encapsulates the task nodes into software components, either using component names attached to nodes or by mapping each leaf node, i.e., which have no parallel or sequential node as a descendent, to a component.

– The *synchronization* step acts a pre-processor. It formats task dependencies and constraints information in such a way to make the next code generation step easier.

C++ program including software components, hardware components architecture, and a main entry is then produced by simply applying a translation stylesheet. Stylesheet contains a set of code generation templates which are applied to the XML-based models, and mapping file, where a template is a pair composed of a node name, and a rule to apply when this node is matched.

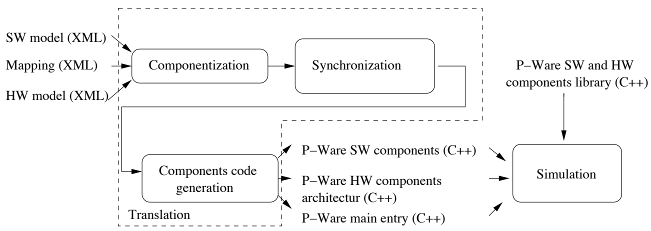

FIG. 7 – From FXML to P-Ware.

Simulation. P-WARE is used for predicting tasks and hardware components performance on an observation time interval defined by designer. As an example, the predicted performance data are available bus bandwidths, memory conflicts, cache misses, tasks communication times, synchronization times, and execution times.

As shown on the figure 6 by the two loops, the designer may re-iterate the design cycle to analyze another possible implementation, either by using different mapping for tasks, and/or by using different hardware architecture configuration. Therefore, design loops convergence is controlled by designer.

Code generation. This phase generates the actual code to be compiled onto the real hardware.

Références

[1] I. Assayad, V. Bertin, F-X. Defaut, Ph. Gerner, O. Quevreux, S. Yovine. Jahuel : A formal framework for software synthesis. In Proceedings of ICFEM 2005 Seventh International Conference on Formal Engineering Methods". 1-4 November 2005, Manchester, UK. LNCS 3785, Pages : 204-218. Springer, 2005. 1.2

[2] I. Assayad, S. Yovine. P-Ware : A precise and scalable component-based simulation tool for embedded multiprocessor industrial applications. EUROMICRO Conference on Digital System Design (DSD 2007), August 2007. IEEE Computer Society Press. 2

A XML schema of FXML “archi” V.0

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="1.0"
   xmlns="http://www.verimag.fr/Step1_PreProcessing/fxmlarchi"
   targetNamespace="http://www.verimag.fr/Step1_PreProcessing/fxmlarchi">


<xs:element name="testtest" type="xs:string"/>

<xs:element name="architecture">
<xs:complexType>
<xs:sequence>
<xs:element ref="anode" minOccurs="1" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:complexType name="anode-type">
<xs:sequence>
<xs:element name="arch-label" type="xs:string" minOccurs="1" maxOccurs="1"/>
<xs:element name="qin" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="qout" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element name="anode" type="anode-type" abstract="true"/>

<xs:complexType name="processor-type">
<xs:complexContent>
<xs:extension base="anode-type">
<xs:sequence>
<xs:element name="clock-port" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="number" type="xs:integer" minOccurs="1" maxOccurs="1"/>
<xs:element name="latency" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="dmadbuf" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="database" type="xs:string" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="processor" type="processor-type" substitutionGroup="anode"/>


<!-- ####################
 # Memory definition #
 #################### -->


<xs:complexType name="memory-type">
<xs:complexContent>
<xs:extension base="anode-type">
<xs:sequence>
<xs:element name="clock-port" type="xs:string" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="memory" type="memory-type" substitutionGroup="anode"/>


<!-- #################
 # Bus definition #
 ################# -->


<xs:complexType name="bus-type">
<xs:complexContent>
<xs:extension base="anode-type">
<xs:sequence>
<xs:element name="clock-port" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="dunit-number" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="dmatable" type="xs:string" minOccurs="0" maxOccurs="1"/>
<xs:element name="banks-number" type="xs:string" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="bus" type="bus-type" abstract="true" substitutionGroup="anode"/>

<xs:element name="Rbus" type="bus-type" substitutionGroup="bus"/>

<xs:element name="PRbus" type="bus-type" substitutionGroup="bus"/>

<xs:element name="Wbus" type="bus-type" substitutionGroup="bus"/>

<xs:element name="PWbus" type="bus-type" substitutionGroup="bus"/>

<xs:element name="Cbus" type="bus-type" substitutionGroup="bus"/>


<!-- ####################
 # Buffer definition #
 #################### -->


<xs:complexType name="buffer-type">
<xs:complexContent>
<xs:extension base="anode-type">
<xs:sequence>
<xs:element name="size" type="xs:integer" minOccurs="1" maxOccurs="1"/>
</xs:sequence>
```

```
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="buffer" type="buffer-type" abstract="true" substitutionGroup="anode"/>

<xs:element name="Data-buffer" type="buffer-type" substitutionGroup="buffer"/>

<xs:element name="Transaction-buffer" type="buffer-type" substitutionGroup="buffer"/>


<!-- ####################
 # Clock definition #
 #################### -->


<xs:complexType name="clock-port-type">
<xs:complexContent>
<xs:extension base="anode-type">
<xs:sequence>
<xs:element name="period" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<xs:element name="duty-cycle" type="xs:integer" minOccurs="0" maxOccurs="1"/>
<xs:element name="start-time" type="xs:integer" minOccurs="0" maxOccurs="1"/>
</xs:sequence>
</xs:extension>
</xs:complexContent>
</xs:complexType>

<xs:element name="clock-port" type="clock-port-type" substitutionGroup="anode"/>


</xs:schema>
```