Technische Universiteit
Eindhoven
University of Technology

Linking Papyrus UML Modeling Framework to SDF^3

Gabriela Breaban, Sander Stuijk, and Kees Goossens

ES Reports
ISSN 1574-9517

ERS-2016-03
1 April 2016

© 2016 Technische Universiteit Eindhoven, Electronic Systems.
All rights reserved.

http://www.es.ele.tue.nl/esreports
esreports@es.ele.tue.nl

Eindhoven University of Technology
Department of Electrical Engineering
Electronic Systems
PO Box 513
NL-5600 MB Eindhoven
The Netherlands

Linking Papyrus UML Modeling Framework to SDF^3

Gabriela Breaban, Sander Stuijk, Kees Goossens

Eindhoven University of Technology
Email:{g.breaban, s.stuijk, k.g.w.goossens}@tue.nl

I. INTRODUCTION

In the context of the OpenES project we have a developed a conversion from the Papyrus UML Modeling Framework [2] to Synchronous Dataflow Graphs. This makes it possible to use the performance analysis tool developed by TUe, SDF^3 [6]. SDF^3 computes performance metrics (such as throughput, latency) for applications modeled with Dataflow graphs. Several Dataflow flavors are supported, such as SDF (Synchronous Dataflow), CSDF (Cyclostatic Dataflow), SADF (Scenario-Aware Dataflow) and FSMSADF (Finite State Machine-based Scenario Aware Dataflow). The converter takes as input an SDF graph modeled as an UML activity diagram [4] and performs a XML format translation in order to generate a input file compatible with SDF^3. For modeling SDF graphs in UML, we used the SDF metamodel defined by Mallet et al. [1], [3]. We demonstrate our work on an MPEG-4 decoder SDF graph.

The following sections present the SDF metamodel and its usage for creating a timed SDF graph in UML (Section II) and the application of the developed bridge for the MPEG-4 decoder use case (Section IV).

II. CREATING TIMED SDF GRAPHS IN PAPYRUS

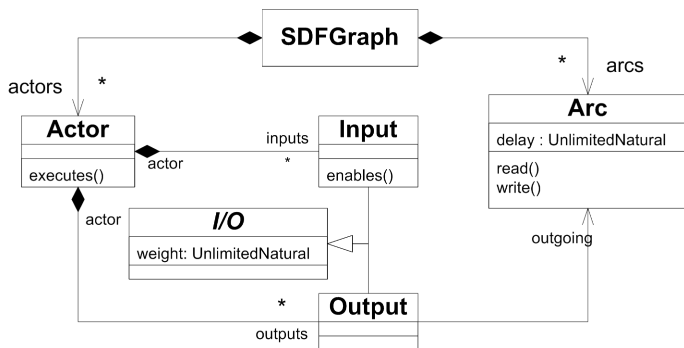

Fig. 1. SDF Metamodel for UML

Figure 1 shows the structure of the SDF metamodel. The metamodel specifies the structure of a SDF graph. A SDF graph is composed of two types of elements, actors and arcs. An actor can have input and output ports, depicted as ‘Input’,‘Output’ classes in the figure. ‘Input’ and ‘Output’ classes are subclasses of the ‘I/O’ class that has the ‘weight’ property. Each actor ‘Input’ has a ‘weight’ that represents the number of tokens consumed when an actor executes and each actor ‘Output’ has a ‘weight’ that represents the number of tokens produced. Arcs have a ‘delay’ property that represent the number of tokens initially present in the first-in first-out arc queue.

A SDF graph can be modeled by an UML activity diagram. Typically, an UML activity diagram is composed of nodes and edges. When mapping the SDF metamodel to an activity diagram, the activity nodes are used to represent SDF actors and the activity edges represent the SDF arcs that connect the actors. The actor input/output port rates are represent by the edge weights. This means that the weight value of the input/output edge corresponding to a input/output port is equal to the port firing rate. The input and output edges corresponding to a pair of connected input/output ports are connected by an activity ‘Join’ node. Thus an SDF arc is represented in activity diagram by an input edge and an output edged connected by a ‘Join’ node. Furthermore, SDF arc delays are also represented as edge weights, only that in this case the edge connects the activity ‘Initial Node’ with the ‘Join’ node corresponding to the SDF arc containing the delay.

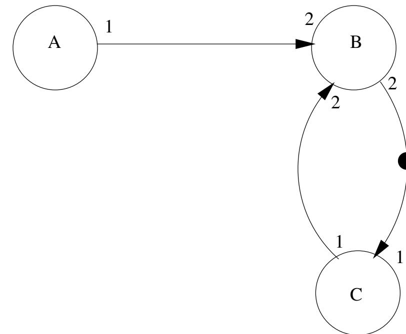

Fig. 2. Example SDF graph

To illustrate these above mentioned properties, let us refer to Figure 2 showing a SDF graph and Figure 3 showing its corresponding UML model. The SDF graph consists of three actors (Actor A, Actor B and Actor C) and three arcs. In the corresponding UML activity diagram, we can see that the actor port rates become edge weights, which are connected via ‘Join’ nodes (the yellow rectangles). An edge weight is set in Papyrus by selecting the edge and then, in the ‘Properties‘

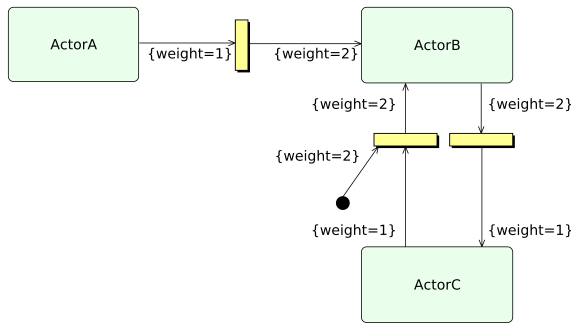

Fig. 3. Example SDF graph in UML

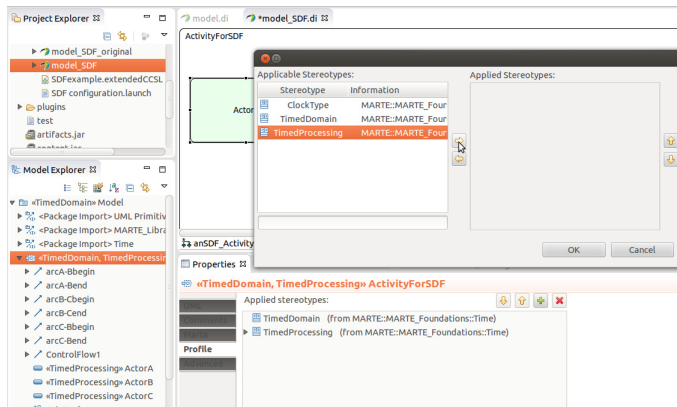

Fig. 4. Example SDF graph in UML

view, the 'UML' tab, setting the weight to the desired value. The initial tokens present on the arc from actor C to actor B are modeled by the edge having as source the initial node and as target the 'Join' node in between actor C and actor B. The edge weight models the number of initial tokens.

One important aspect that is not visible in the shown UML diagram is the possibility to model actor execution times. This can be achieved by applying the MARTE profile [5] to the UML model. The MARTE profile contains a set of stereotypes that can be used to model real-time properties in UML diagrams. The stereotypes that can be used to model execution times are 'TimedDomain' and 'TimedProcessing'. Applying these stereotypes to, for instance, UML activity nodes, gives us the possibility to specify a clock for each activity node and use this time reference to further specify a processing time for it, such as the worst case execution time (WCET). For SDF graphs, exiting works [3] recommend using the 'idealClk', which is a predefined clock part of the MARTE Time library that models the physical time.

Let us present the steps that the user needs to perform for applying the MARTE profile and its stereotypes in Papyrus. To do this, in the 'Model Explorer' view, one has to right click on the model, go to 'Import', 'Import Registered Profile' and select 'MARTE'. As a result, the prefix '<<TimedDomain >>' will appear next to the model name. Next, in the 'Model Explorer' select the activity diagram and in the 'Properties' view, 'Profile' add the 'TimedDomain' and 'TimedProcessing' applied stereotypes. Then, finally, select each actor in the activity diagram within the 'Model Explorer' and in the 'Properties' view, 'Profile', unfold the 'TimedProcessing' stereotype and set the 'on Clock' field to the value 'idealClk' and the 'duration' field to a value equal to the actor execution time. The 'idealClk' is a predefined clock in the MARTE Time library that models the physical time in seconds. Figure 4 illustrates the step of applying the stereotypes to a UML activity.

III. XSLT STYLESHEET

In order to convert an UML activity diagram that models a SDF graph from the XML format generated by Papyrus into the XML format required by SDF^3, we implemented a XSLT stylesheet. The stylesheet takes as input the UML file generated by Papyrus and generates an output XML file based on the SDF to UML activity mapping relations presented in the previous section. The stylesheet code can be found in the Appendix. It comprises a set of templates that match the UML activity elements and create the SDF^3 XML nodes. This means that each activity node of type 'OpaqueAction' is converted into an SDF actor. The actor port rates are obtained from the corresponding input/output edge weights. For each 'Join' activity node, a SDF channel is created for which the source and destination rates, as well as the number of initial tokens are obtained from the weight values of the connected input and output edges. Finally, the actor execution times are obtained from the 'duration' property of the 'TimedProcessing' elements that match the corresponding 'OpaqueAction'

IV. THE MPEG-4 DECODER USE CASE

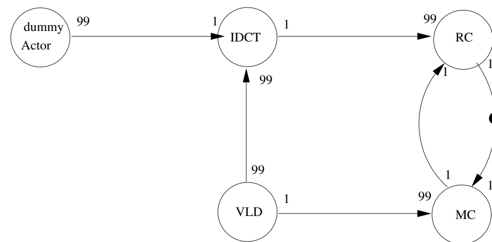

Fig. 5. MPEG-4 Decoder SDF graph

Figure 6 shows an UML activity diagram representing the SDF graph for a MPEG-4 decoder. In order to analyze this with SDF^3, we implemented a XSLT stylesheet that converts the uml file into the XML format expected by SDF^3.

To run the performance analysis for a timed SDF graph modeled by an activity diagram as explained in section II, the following steps are needed:

1) convert the uml file into the SDF^3 format by running XSLT processor using the provided "xslt_stylesheet_SDFMeta_WCET.xsl" stylesheet: example (Linux): xsltproc --output output_sdf_mpeg.xml xslt_stylesheet_SDFMeta_WCET.xsl model_SDF_MPEG.uml

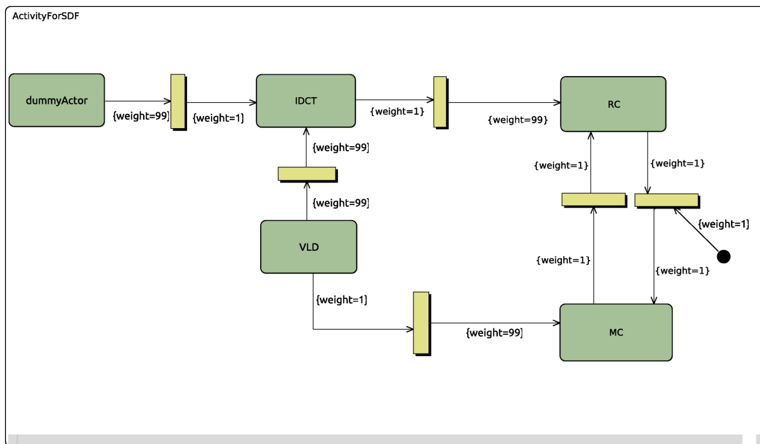

Fig. 6. MPEG-4 Decoder modeled in Papyrus

2) given that SDF^3 was installed and compiled beforehand, go to its installation directory and run sdf3analyze-fsmsadf
example (Linux): /sdf3/sdf3/build/work/Linux/bin/sdf3analyze-fsmsadf --graph output_sdf_mpeg_test.xml --algo throughput

In our example graph, the execution times for the actors are: IDCT-17, RC-350, MC-390, VLD-40 and the dummy actor has 0 execution time. The time units for these execution times are expressed in 1000 cycles on an ARM7 running at 200 MHz. For these values, the SDF^3 analysis gives a throughput value of 0.00135135 iterations per time unit.

Appendix A
XSLT Stylesheet Code

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:uml="http://www.eclipse.org/uml2/4.0.0/UML"
  xmlns:xmi="http://www.omg.org/spec/XMI/20110701"
  xmlns:Time="http://www.eclipse.org/papyrus/Time/1"
     xmlns:str="http://exslt.org/strings"
    extension-element-prefixes="str"
    exclude-result-prefixes="xmi uml Time">
<xsl:output method="xml" omit-xml-declaration="no" encoding="UTF-8" indent="yes" />

<xsl:template match="/xmi:XMI/uml:Model">
    <sdf3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0" type="
        fsmsadf" xsi:noNamespaceSchemaLocation="http://www.es.ele.tue.nl/sdf3/xsd/
        sdf3-fsmsadf.xsd">
    <applicationGraph name="g">
        <xsl:apply-templates/>
    </applicationGraph>
</sdf3>
</xsl:template>

<xsl:template match="packagedElement">
    <fsmsadf>
    <scenariograph name="sg0" type="t">
        <xsl:apply-templates select="node" mode="createActor" />
    </scenariograph>
    </fsmsadf>
    <fsmsadfProperties>
      <scenarios>
        <scenario name="s0" graph="sg0">
            <xsl:apply-templates select="node" mode="createActProperties" />
        </scenario>
      </scenarios>
    </fsmsadfProperties>
    <fsm initialstate="s0">
      <state name="s0" scenario="s0">
        <transition destination="s0"/>
      </state>
    </fsm>
</xsl:template>


<!-- template to create the actor nodes -->
<xsl:template match="node" mode="createActor">
     <xsl:if test="@xmi:type='uml:OpaqueAction'">
        <xsl:variable name="stringListOut" select="str:tokenize(@outgoing,' ')"/>
        <xsl:variable name="stringListIn" select="str:tokenize(@incoming,' ')"/>
                <actor name="{@name}" type="Actor">
                  <xsl:variable name="crtNode" select="../edge" />
                  <xsl:for-each select="$stringListOut">
                          <xsl:variable name="i" select="position()" />
                          <xsl:variable name="elemName" select='.'/>
                              <xsl:apply-templates select="$crtNode" mode="findPort">
                                <xsl:with-param name="edgeName" select="$elemName"/>
```

```
                    <xsl:with-param name="direction" select="'out'"/>
                    <xsl:with-param name="pname" select="$elemName" />
                  </xsl:apply-templates>
            </xsl:for-each>
            <xsl:for-each select="$stringListIn">
                    <xsl:variable name="i" select="position()" />
                    <xsl:variable name="elemName" select='.'/>
                        <xsl:apply-templates select="$crtNode" mode="findPort">
                          <xsl:with-param name="edgeName" select="$elemName"/>
                          <xsl:with-param name="direction" select="'in'"/>
                          <xsl:with-param name="pname" select="$elemName" />
                        </xsl:apply-templates>
            </xsl:for-each>
          </actor>
    </xsl:if>

    <xsl:if test="@xmi:type='uml:JoinNode'">
       <xsl:variable name="chName" select="@xmi:id"/>
       <xsl:variable name="crtNode1" select=".." />
       <xsl:apply-templates select="$crtNode1" mode="findSrcDstInitTokens">
         <xsl:with-param name="channelName" select="$chName" />
       </xsl:apply-templates>
    </xsl:if>
</xsl:template>

<!-- template to create the actor port nodes -->
<xsl:template match="*" mode="findPort">
    <xsl:param name="edgeName"/>
    <xsl:param name="direction"/>
    <xsl:param name="pname" />
    <xsl:if test="@xmi:id=$edgeName">
       <xsl:variable name="prate" select="weight/@value" />
       <port name="{$pname}" type="{$direction}" rate="{$prate}"/>
    </xsl:if>
</xsl:template>


<!-- template to create the channel nodes -->
<xsl:template match="*" mode="findSrcDstInitTokens">
    <xsl:param name="channelName"/>

    <xsl:variable name="srcPort" select="edge[(@target=$channelName) and (weight/
        @name='outputWeight')]/@xmi:id"/>
    <xsl:variable name="srcActorId" select="edge[(@target=$channelName) and (weight/
        @name='outputWeight')]/@source"/>
    <xsl:variable name="srcActor" select="node[(@xmi:id=$srcActorId)]/@name"/>
    <xsl:variable name="dstPort" select="edge[(@source=$channelName) and (weight/
        @name='inputWeight')]/@xmi:id"/>
    <xsl:variable name="dstActorId" select="edge[(@source=$channelName) and (weight/
        @name='inputWeight')]/@target"/>
    <xsl:variable name="dstActor" select="node[(@xmi:id=$dstActorId)]/@name"/>
    <xsl:variable name="initTokens" select="edge[(@target=$channelName) and (weight/
        @name='initialDelay')]/weight/@value"/>
    <xsl:variable name="egdeNodesCnt" select="count(edge[(@target=$channelName) and (
        weight/@name='initialDelay')])" />
```

```
    <xsl:if test="$egdeNodesCnt␣=␣0">
      <channel name="{$channelName}" srcActor="{$srcActor}" srcPort="{$srcPort}"
          dstActor="{$dstActor}" dstPort="{$dstPort}"/>
    </xsl:if>
    <xsl:if test="$egdeNodesCnt␣!=␣0">
      <channel name="{$channelName}" srcActor="{$srcActor}" srcPort="{$srcPort}"
          dstActor="{$dstActor}" dstPort="{$dstPort}" initialTokens="{$initTokens}" />
    </xsl:if>
</xsl:template>

<!-- template to create the actor properties -->
<xsl:template match="node" mode="createActProperties">
    <xsl:if test="@xmi:type='uml:OpaqueAction'">
        <xsl:variable name="actId" select="@xmi:id" />
        <xsl:variable name="actName" select="@name" />
        <xsl:variable name="crtNode" select="/xmi:XMI" />
          <xsl:apply-templates select="$crtNode" mode="findWCET">
              <xsl:with-param name="actorId" select="$actId"/>
              <xsl:with-param name="actorName" select="$actName"/>
          </xsl:apply-templates>
    </xsl:if>
</xsl:template>


<xsl:template match="*" mode="findWCET">
    <xsl:param name="actorId"/>
    <xsl:param name="actorName"/>

    <xsl:variable name="exeTime" select="Time:TimedProcessing[@base_Action=$actorId]/
        duration/@value"/>
    <actorProperties actor="{$actorName}">
      <processor type="p0" default="true">
      <xsl:choose>
         <xsl:when test="$exeTime␣!=␣''">
            <executionTime time="{$exeTime}"/>
         </xsl:when>
         <xsl:otherwise>
            <executionTime time="0"/>
         </xsl:otherwise>
      </xsl:choose>
      </processor>
    </actorProperties>
</xsl:template>

</xsl:stylesheet>
```

References

[1] C. Glitia, J. Deantoni, and F. Mallet. Logical time at work: capturing data dependencies and platform constraints. In *Forum for Design Languages (FDL)*, Proceedings of the 2010 Forum on specification & Design Languages, pages 240–246, Southampton, United Kingdom, Sept. 2010. Electronic Chips & Systems design Initiative (ECSI).

[2] A. Lanusse, Y. Tanguy, H. Espinoza, C. Mraidha, S. Gerard, P. Tessier, R. Schnekenburger, H. Dubois, and F. Terrier. Papyrus uml: an open source toolset for mda. In *ECMDA-FA 09: Model driven architecture - foundations and applications: 5th European conference, ECMDA-FA 2009, Enschede, the Netherlands, June 23-26, 2009 ; proceedings*, page 14. Springer, 2009.

[3] F. Mallet, J. Deantoni, C. André, and R. De Simone. The Clock Constraint Specification Language for building timed causality models. *Innovations in Systems and Software Engineering*, pages 99–106, Mar. 2010.

[4] OMG. *Unified Modeling Language: Superstructure*. Object Modeling Group, 2000.

[5] OMG. Uml profile for marte: Modeling and analysis of real-time embedded systems, 2009.

[6] S. Stuijk, M. Geilen, and T. Basten. Sdf3: Sdf for free. In *ACSD*, pages 276–278. IEEE Computer Society, 2006.