A UPPAAL model for timing analysis of atomic execution in component-based multi-mode systems

Yin Hang, Hans Hansson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
{young.hang.yin,hans.hansson}@mdh.se

April 18, 2012

Abstract

This report provides a complete UPPAAL model of the mode switch handling of an Atomic Execution Group (AEG) in a component-based multi-mode system (CBMMS) with pipe-and-filter architecture. The purpose of this model is to analyze the worst-case latency due to the atomic execution of this AEG during a mode switch. This worst-case latency plays a significant role in deriving the global mode switch time of a CBMMS.

1 Introduction

The mode switch handling of component-based multi-mode systems (CBMMSs) is an emerging topic. We previously developed a Mode Switch Logic (MSL) [1] [4] to achieve the composable mode switch of CBMMSs. Our MSL was later extended by the mode mapping mechanism [2] to solve the mode incompatibility problem between different components. Since many CBMMSs are real-time systems, it is important to ensure that a mode switch can be completed within a bounded time. We have performed a preliminary mode switch timing analysis [3] to determine the global mode switch time. However, we used to assume that the execution of each component is immediately aborted due to a mode switch. This assumption is unrealistic because in practice one or a set of components can have atomic execution that must run to completion and cannot be aborted. We consider them as an Atomic Execution Group (AEG), which can consist of one or several components. When a mode switch is triggered, any ongoing execution in an AEG must be completed first, thus the mode switch will be delayed. In this report, we will present a UPPAAL [5] model of the mode switch handling of an AEG to derive its worst-case latency during a mode switch. Figure 1 shows the components and their connections in the AEG as part of a CBMMS. This example will be used throughout this report. The system has a pipe-and-filter architecture as it waits for the input data, processes the data and then generates the output data. Similarly, all its components follow the same repeated

execution pattern: waiting for inputs, processing data and producing outputs. Different components can process different data simultaneously.

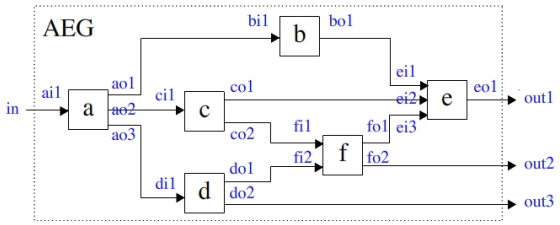

Figure 1: An Atomic Execution Group (AEG)

A component has a non-empty set of input and output ports (see the blue texts in Figure 1). We assume that primitive components have data going through all its input and output ports, i.e. input data has to be available at all input ports before processing can start and output data must be sent via all output ports. Whenever a primitive component receives new data at an input port, the data is first queued in a corresponding input buffer. While a primitive component is processing data, new arriving data must wait in its input buffers and cannot be processed until the component completes its current data processing. As opposed to a primitive component, a composite component does not buffer its input data. Whenever it receives new data, it will simply propagate the data to its subcomponents. Similarly, whenever it receives output data from its subcomponents, it will immediately forward the data via its corresponding output port(s). Figure 1 only shows activated primitive components in the AEG in the current mode because deactivated primitive components or composite components do not contribute to the atomic execution time of this AEG.

The data processing time of each primitive component c_i in the AEG is bounded by a timing interval $[C_{c_i}^{min}, C_{c_i}^{max}]$. The incoming data rate of the AEG is within the interval $[R_{min}, R_{max}]$. When a mode switch is triggered, all the data within the AEG must be completely processed before the AEG can switch mode. AE denotes the worst-case value of this data processing time. To ensure that AE is bounded and that our calculations terminate, we will enforce a maximum number of data elements in the AEG. Depending on the incoming data rate and component processing times, this bound may or may not be reached. In fact, the bound could be used as a modeling artifact, but could also be a mechanism in the real system. The timing parameters of the system are as follows:

- Incoming data rate R=[7,8].
- Data processing time C of components a-f: C_a=[4,5], C_b=[7,8], C_c=[6,7], C_d=[5,6], C_e=5, C_f=[7,8].
- Maximum number of data elements in the AEG N=5.

The complete UPPAAL model includes five parts:

1. *Data source*: generates data at a flexible rate.

2. *MSI source*: the source that triggers a Mode Switch Instruction (MSI) at any time. Upon receiving an MSI, a primitive component will start its reconfiguration for the new mode and a composite component will refer to its local mode mapping and propagate the MSI to its subcomponents based on the mode mapping result.

3. The *AEG*: receives data from *Data source*, processes it and deposits the results at its output port(s). Furthermore, it ensures that the number of data elements n in the AEG is within the bound N. *Data source* is turned off when $n = N$. If mode switch is not in progress, *Data source* is turned on again when n decreases. When the AEG receives an MSI, *Data source* will also be turned off. AE is the maximal data processing time to reach $n = 0$.

4. *Data forwarder*: forwards data between components without the sender having to know the identity of the receiver. This simplifies the modeling, since when some connections are changed, or a component is removed or added, only component connection definitions referred to by *Data forwarder* need to be updated.

5. *Primitive components*: modeled by a UPPAAL template for each of them. Although the number of components could be arbitrary, we can use a parameterized generic UPPAAL model that applies to all components.

2 The complete UPPAAL model

In this section, the UPPAAL models of all the five parts introduced in the last section will be presented.

2.1 The global declaration

```
//Components
const int top=1;
const int a=2;
const int b=3;
const int c=4;
const int d=5;
const int e=6;
const int f=7;

//Ports
const int pin=51;
const int ai1=52;
const int ao1=53;
const int ao2=54;
const int ao3=55;
const int bi1=56;
const int bo1=57;
const int ci1=58;
const int co1=59;
const int co2=60;
const int di1=61;
const int do1=62;
const int do2=63;
const int ei1=64;
const int ei2=65;
const int ei3=66;
const int eo1=67;
const int fi1=68;
const int fi2=69;
const int fo1=70;
const int fo2=71;
const int pout1=72;
const int pout2=73;
const int pout3=74;

const int pMin=51;
const int pMax=74;
const int cMin=1;
```

```
const int cMax=7;

int[0,10] dataCounter=0;
bool DSstatus=true;

//--------------------The mapping of component connection
const int conPairN=12;
const int[pMin,pMax] top_from[conPairN]={pin,ao1,ao2,ao3,bo1,co1,co2,do1,do2,eo1,fo1,fo2};
const int[pMin,pMax] top_to[conPairN]={ai1,bi1,ci1,di1,ei1,ei2,fi1,fi2,pout3,pout1,ei3,pout2};

//Global input and output data
const int iData=2;  //To simplify the problem, the input value is always 2.
//Its value has nothing to do with the worst-case latency of the AEG
int[pMin,pMax] origin;    //The data sender
int[pMin,pMax] target;    //The data receiver
int[0,150] data;

broadcast chan newData;
chan dataIn,dataOut,MSI;
urgent chan Go;
```

2.2 Data source

The UPPAAL model of *Data source* is presented in Figure 2.

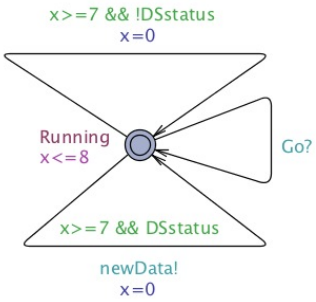

Figure 2: Data source

Local declaration:

```
clock x;
```

2.3 MSI source

The UPPAAL model of the MSI source is presented in Figure 3. It has no local declaration.

Figure 3: MSI source

2.4 The Atomic Execution Group (AEG)

The UPPAAL model of *the AEG* is presented in Figure 4.
Local declaration:

```
clock z;
const int THRESHOLD=5;
const int oNumber=3;
int[pout1,pout3] outputList[oNumber]={pout1,pout2,pout3};
int[0,150] outputDList[oNumber]={0,0,0};
```

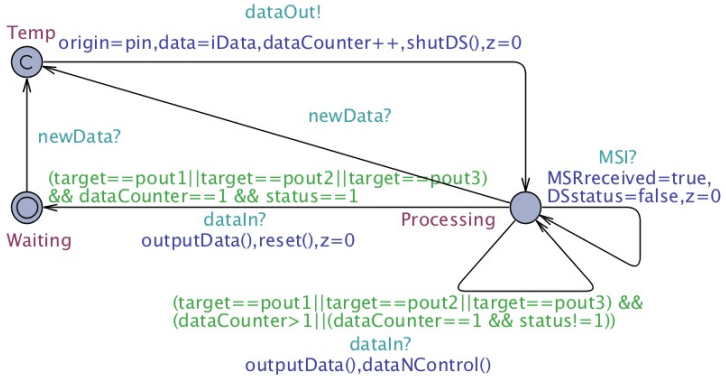

Figure 4: the AEG

```
int[0,THRESHOLD] outputNList[oNumber]={0,0,0};
int[0,2] status=0;
bool MSIreceived=false;

void outputData()
{
    int i;
    for(i=0;i<oNumber;i++)
    {
        if(target==outputList[i])
        {
            outputDList[i]=data;
        }
    }
}

int[0,2] outputNStatus()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<oNumber;i++)
    {
        if(outputNList[i]!=0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==oNumber)
    {
        return 2;  //all segments of a data can be sent out
    }
    else if(numberOfNonZero==oNumber-1)
    {
        return 1;  //Only one branch is pending
    }
    else
    {
        return 0;  //Nothing special
    }
}

void dataNControl()
{
    int i;
    for(i=0;i<oNumber;i++)
    {
        if(target==outputList[i])
        {
            outputNList[i]++;
        }
    }
    status=outputNStatus();
    if(status==2)
    {
        for(i=0;i<oNumber;i++)
        {
            outputNList[i]--;
        }
        dataCounter--;
        status=outputNStatus();
        if(dataCounter<THRESHOLD && !MSIreceived)
        {
            DSstatus=true;
        }
    }
}
```

```
void shutDS()
{
    if(dataCounter>=THRESHOLD)
    {
        DSstatus=false;
    }
}

void resetDS()
{
    if(!DSstatus)
    {
        DSstatus=true;
    }
}

void reset()
{
    int i;
    dataCounter=0;
    for(i=0;i<oNumber;i++)
    {
        outputNList[i]=0;
    }
    status=0;
    resetDS();
    if(MSIreceived=true)
    {
        MSIreceived=false;
    }
}
```

2.5 Data forwarder

The UPPAAL model of *Data forwarder* is presented in Figure 5.

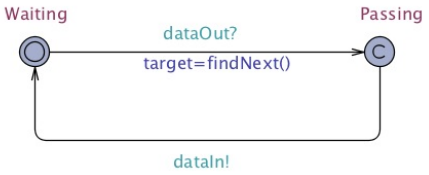

Figure 5: Data forwarder

Local declaration:

```
const int error=-1;

int findNext()
{
    int i;
    for(i=0;i<conPairN;i++)
    {
        if(origin==top_from[i])
        {
            return top_to[i];
        }
    }
    return error;  //Since component connection is well defined, the error never happens
}
```

2.6 Component *f*

The UPPAAL model of Component f is presented in Figure 6. As the most representative component, Component f has multiple inputs and outputs. The model of f is generic in the sense that all the other components in the AEG from Figure 1 can be modeled in the same way. When it is not processing any data, it is in state **nonProcessing**. When it is processing data, it is in state **Processing**. The invariant $x<=8$ and guard $x>7$ define the interval of its data processing time, i.e. $C_f = [7, 8]$. Component f receives data through the channel *dataIn?* and sends output data through the channel *dataOut!*. f recognizes new

data by the guard *target==fi1||target==fi2* where *fi1* and *fi2* are its input ports. When all input buffers are non-empty, the boolean variable *readyToProcess* is set to true and f will switch to location **Processing** by the urgent channel *Go!*. Data is processed by the function *processData()*, thus representing mode-specific behavior of a primitive component. After processing the data, f immediately sends its output data through all its output ports. This is modeled by the sequential and atomic output data generation from its output ports. The two committed states **Temp1** and **Temp2** guarantee its atomicity. *outputCounter* records how many output ports have sent out the data. When the output data is sent through all its output ports, f goes back to state **nonProcessing** and checks its buffer status again.

When modeling another component with different number of inputs and outputs, the model structure remains the same and only some parameters need to be changed. If a component has only one output port, the model can be simplified by removing state **Temp2** and *outputCounter*.

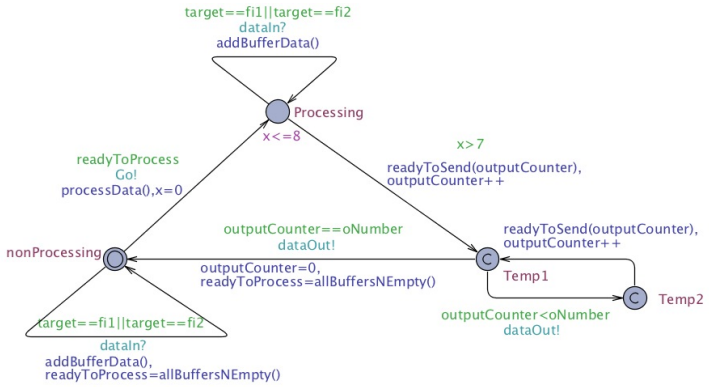

Figure 6: Component f

Local declaration:

```
clock x;
int[0,150] fData;
const int inputN=2;
const int inputList[inputN]={fi1,fi2};
int[-1,150] buffer[inputN][5]={{-1,-1,-1,-1,-1},{-1,-1,-1,-1,-1}};
int[0,5] bufferN[inputN]={0,0};
bool readyToProcess=false;
const int oNumber=2;
const int outputList[2]={fo1,fo2};
int[0,oNumber] outputCounter=0;

bool allBuffersNEmpty()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
    {
        if(bufferN[i]>0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    int i;
```

```
    for(i=0;i<inputN;i++)
    {
        if(target==inputList[i])
        {
            if(bufferN[i]<5)
            {
                buffer[i][bufferN[i]]=data;
                bufferN[i]++;
            }
        }
    }
}

void adjustBuffer()
{
    int i;
    int j;
    for(i=0;i<inputN;i++)
    {
        for(j=0;j<bufferN[i];j++)
        {
            buffer[i][j]=buffer[i][j+1];
        }
        bufferN[i]--;
    }
}

void processData()
{
    int i;
    fData=0;
    for(i=0;i<inputN;i++)
    {
        fData+=buffer[i][0];
    }
    adjustBuffer();
}

void readyToSend(int oCounter)
{
    data=fData;
    origin=outputList[oCounter];
}
```

2.7 Component *a*

The UPPAAL model of Component *a* is presented in Figure 7.

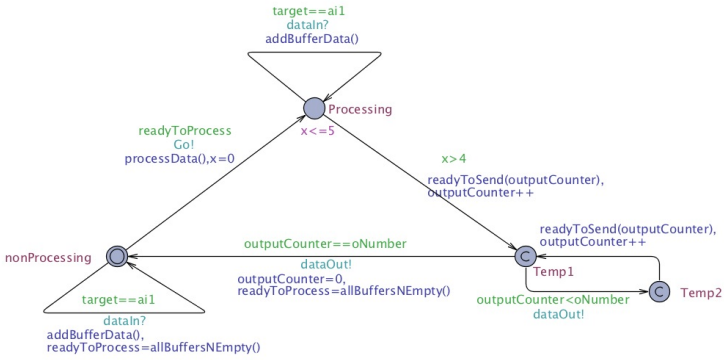

Figure 7: Component *a*

Local declaration:

```
clock x;
int[0,150] aData;
const int inputN=1;
int[-1,150] buffer[5]={-1,-1,-1,-1,-1};  //-1 means no data
int[0,5] bufferN=0;
bool readyToProcess=false;
const int oNumber=3;
const int outputList[oNumber]={ao1,ao2,ao3};
int[0,oNumber] outputCounter=0;

bool allBuffersNEmpty()
{
    int i;
```

```
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
    {
        if(bufferN>0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    if(bufferN<5)
    {
        buffer[bufferN]=data;
        bufferN++;
    }
    /*else
    {
        Buffer overflow
    }*/
}

void adjustBuffer()
{
    int i;
    for(i=0;i<bufferN;i++)
    {
        buffer[i]=buffer[i+1];
    }
    bufferN--;
}

void processData()
{
    aData=buffer[0];
    adjustBuffer();
    aData=aData+10;
}

void readyToSend(int oCounter)
{
    data=aData;
    origin=outputList[oCounter];
}
```

2.8 Component *b*

The UPPAAL model of Component b is presented in Figure 8.

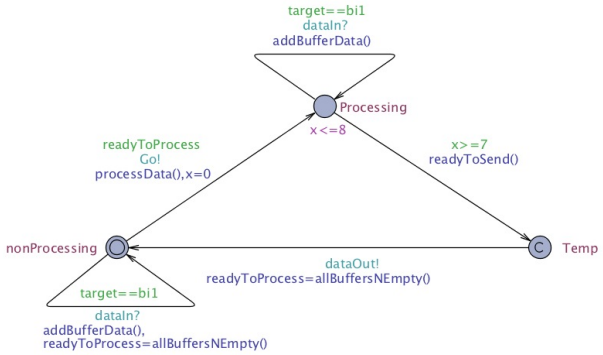

Figure 8: Component b

Local declaration:

```
clock x;
int[0,150] bData;
const int inputN=1;
int[-1,150] buffer[5]={-1,-1,-1,-1,-1};  //-1 means no data
```

```
int[0,5] bufferN=0;
bool readyToProcess=false;

bool allBuffersNEmpty()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
    {
        if(bufferN>0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    if(bufferN<5)
    {
        buffer[bufferN]=data;
        bufferN++;
    }
    /*else
    {
        Buffer overflow
    }*/
}

void adjustBuffer()
{
    int i;
    for(i=0;i<bufferN;i++)
    {
        buffer[i]=buffer[i+1];
    }
    bufferN--;
}

void processData()
{
    bData=buffer[0];
    adjustBuffer();
    bData=bData*2;
}

void readyToSend()
{
    data=bData;
    origin=bo1;
}
```

2.9 Component *c*

The UPPAAL model of Component *c* is presented in Figure 9.

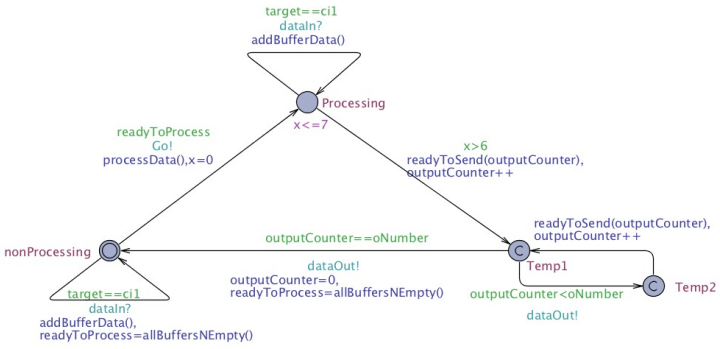

Figure 9: Component *c*

Local declaration:

```
clock x;
int[0,150] cData;
const int inputN=1;
int[-1,150] buffer[5]={-1,-1,-1,-1,-1};  //-1 means no data
int[0,5] bufferN=0;
bool readyToProcess=false;
const int oNumber=2;
const int outputList[2]={co1,co2};
int[0,oNumber] outputCounter=0;

bool allBuffersNEmpty()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
    {
        if(bufferN>0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    if(bufferN<5)
    {
        buffer[bufferN]=data;
        bufferN++;
    }
    /*else
    {
        Buffer overflow
    }*/
}

void adjustBuffer()
{
    int i;
    for(i=0;i<bufferN;i++)
    {
        buffer[i]=buffer[i+1];
    }
    bufferN--;
}

void processData()
{
    cData=buffer[0];
    adjustBuffer();
    cData=cData+10;
}

void readyToSend(int oCounter)
{
    data=cData;
    origin=outputList[oCounter];
}
```

2.10 Component *d*

The UPPAAL model of Component d is presented in Figure 10.

Local declaration:

```
clock x;
int[0,150] dData;
const int inputN=1;
int[-1,150] buffer[5]={-1,-1,-1,-1,-1};  //-1 means no data
int[0,5] bufferN=0;
bool readyToProcess=false;
const int oNumber=2;
const int outputList[2]={do1,do2};
int[0,oNumber] outputCounter=0;

bool allBuffersNEmpty()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
    {
        if(bufferN>0)
        {
            numberOfNonZero++;
        }
    }
```

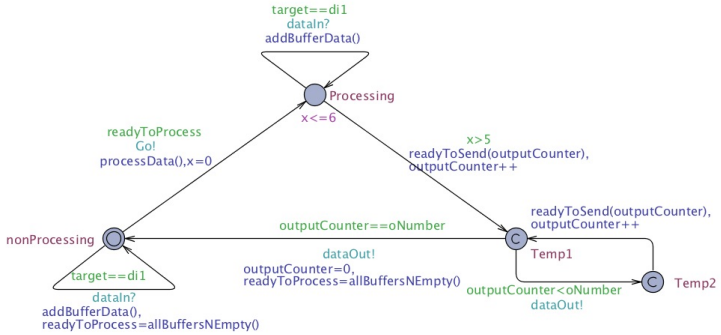

Figure 10: Component d

```
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    if(bufferN<5)
    {
        buffer[bufferN]=data;
        bufferN++;
    }
    /*else
    {
        Buffer overflow
    }*/
}

void adjustBuffer()
{
    int i;
    for(i=0;i<bufferN;i++)
    {
        buffer[i]=buffer[i+1];
    }
    bufferN--;
}

void processData()
{
    dData=buffer[0];
    adjustBuffer();
    dData=dData+10;
}

void readyToSend(int oCounter)
{
    data=dData;
    origin=outputList[oCounter];
}
```

2.11 Component *e*

The UPPAAL model of Component e is presented in Figure 11.

Local declaration:

```
clock x;
int[0,150] eData;
const int inputN=3;
const int inputList[inputN]={ei1,ei2,ei3};
int[-1,150] buffer[inputN][5]={{-1,-1,-1,-1,-1},{-1,-1,-1,-1,-1},{-1,-1,-1,-1,-1}};
int[0,5] bufferN[inputN]={0,0,0};
bool readyToProcess=false;

bool allBuffersNEmpty()
{
    int i;
    int numberOfNonZero=0;
    for(i=0;i<inputN;i++)
```

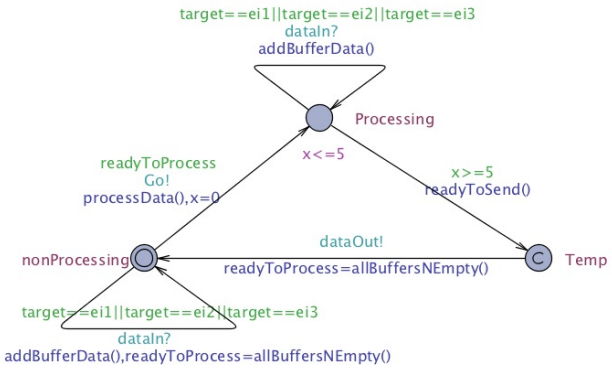

Figure 11: Component e

```
    {
        if(bufferN[i]>0)
        {
            numberOfNonZero++;
        }
    }
    if(numberOfNonZero==inputN)
    {
        return true;
    }
    else
    {
        return false;
    }
}

void addBufferData()
{
    int i;
    for(i=0;i<inputN;i++)
    {
        if(target==inputList[i])
        {
            if(bufferN[i]<5)
            {
                buffer[i][bufferN[i]]=data;
                bufferN[i]++;
            }
        }
    }
}

void adjustBuffer()
{
    int i;
    int j;
    for(i=0;i<inputN;i++)
    {
        for(j=0;j<bufferN[i];j++)
        {
            buffer[i][j]=buffer[i][j+1];
        }
        bufferN[i]--;
    }
}

void processData()
{
    int i;
    eData=0;
    for(i=0;i<inputN;i++)
    {
        eData+=buffer[i][0];
    }
    adjustBuffer();
}

void readyToSend()
{
    data=eData;
    origin=eo1;
}
```

3 Verification

Some interesting results including AE can be obtained by verifying the following properties of the UPPAAL model:

- *A[] not deadlock*: no deadlock will occur in the model.
- *sup{AEG.Processing}: AEG.z*: returns the maximal value of the clock z of AEG in state **Processing**. This equals AE.
- *E<> AEG.Processing && AEG.z==AE*: there is a scenario in which the clock z reaches AE when AEG is in state **Processing**. Once AE is derived, this property searches the worst-case scenario, and using the "Diagnostic Trace" function of UPPAAL, the worst-case scenario can be displayed as an execution trace.
- *sup: dataCounter*: returns the maximal number of data items that can be simultaneously processed in the AEG. If N is only a modeling artifact, then for the validity of the calculated AE, this value must be less than N. In other cases, validity requires a mechanism in the deployed system that keeps n within the bound N.
- *sup: Component.bufferN[Index]*: returns the maximal number of elements in one buffer of a component.

Using UPPAAL, all properties have been verified and satisfied. Figure 12 shows the verification result. When properties with the "sup" operator are successfully verified, the corresponding result will be obtained in a pop-up window. When R=[7,8], the maximal number (i.e. n) of data items in the AEG is 5, meaning that the threshold N can be reached. In the worst-case, $AE = 40$.

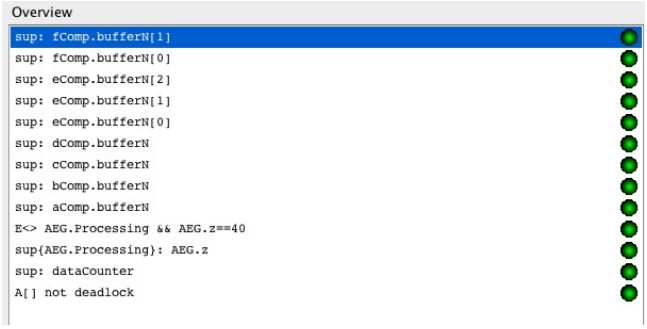

Figure 12: Verification result for R=[7,8]

Moreover, the verification results show that the data rate and component data processing time have substantial influence on the property verification time. The differences are related to variations in the number and length of executions leading up to the worst-case scenario. We repeated the verification of the same set of properties for different data rates and summarized the most important results in Table 1[1].

[1]Verification is performed on MacBook Pro, with 2.66GHz Intel Core 2 Duo CPU and 8GB 1067 MHz DDR3 memory.

An interesting side effect of our modeling is that we can use the last property above to obtain the maximal buffer usage (i.e. required buffer sizes) for the component input buffers. These values are for the considered data rates presented in Table 2.

Apart from R, verification time also depends on the number of components in the AEG, the number of connections and output ports of the AEG, the threshold N and the data processing time of each component. Regardless of the verification time, the way that we model the system does not change.

Property/Value	$R = [6, 8]$	$R = [7, 8]$	$R = [8, 10]$	$R = [10, 12]$
No deadlock	28.64s	5.617s	0.139s	0.108s
Maximal n	5	5	4	3
Deriving AE	45.667s	4.36s	0.1s	0.069s
AE	40	40	25	25
Worst-case scenario	36.716s	4.576s	0.013s	0.016s

Table 1: Property verification results for different data rates

Buffer Index	$R = [6, 8]$	$R = [7, 8]$	$R = [8, 10]$	$R = [10, 12]$
$ai1$	1	1	1	1
$bi1$	3	3	1	1
$ci1$	2	1	1	1
$di1$	1	1	1	1
$ei1$	3	3	2	1
$ei2$	4	4	2	1
$ei3$	3	3	1	1
$fi1$	3	3	1	1
$fi2$	3	3	1	1

Table 2: Maximal buffer usage for different data rates

4 Conclusion

In this report, a UPPAAL model is described and explained in detail for obtaining the worst-case latency due to the atomic execution of an Atomic Execution Group (AEG) in a component-based multi-mode system (CBMMS) during a mode switch. Although this model is only based on a simple example, our UPPAAL models are generic. We conjecture that for any AEG that is in line with our system and component models, we are able to make transformation rules, based on which corresponding UPPAAL models can be automatically generated and verified.

References

[1] Y. Hang, E. Borde, and H. Hansson. Composable mode switch for component-based systems. In *APRES '11: Third International Workshop on Adaptive and Reconfigurable Embedded Systems*, pages 19–22, 2011.

[2] Y. Hang and H. Hansson. A mode mapping mechanism for component-based multi-mode systems. In *4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pages 38–45, 2011.

[3] Y. Hang and H. Hansson. Timing analysis for a composable mode switch. In *The Work-in-Progress session of the 23rd Euromicro Conference on Real-Time Systems*, pages 15–18, 2011.

[4] Y. Hang and H. Hansson. A mode switch logic for component-based multi-mode systems. Technical Report 261/2012, Mälardalen Real-Time Research Centre Mälardalen University, Jan 2012.

[5] Kim Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *STTT-International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.