Timing analysis for mode switch in component-based multi-mode systems

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

***Abstract*—Component-Based Development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable components. Partitioning the behavior into a set of major *operational modes* is a classical approach to reduce complexity of embedded systems design and execution. In supporting system modes in CBD, a key issue is seamless composition of pre-developed multi-mode components into systems. We have previously developed a Mode Switch Logic (MSL) for component-based multi-mode systems implementing such seamless composition.**

In this paper we extend our MSL to cope with atomic transactions, i.e., to handle sets of components that must not be aborted in the middle of the processing of data. This is in contrast with our original MSL, in which components are immediately aborted to perform a mode switch. Based on our extended MSL, we provide analysis of the mode switch timing.

***Keywords*-component-based, mode switch, timing analysis**

I. Introduction

Partitioning system behaviors into different operational modes is a classical approach to reduce complexity and improve efficiency of real-time software. Typically, each mode corresponds to a specific system behavior. The system starts by running in a default initial mode and switches to another appropriate mode when some condition changes. For instance, the control software of an airplane could run in modes: *taxi* (the initial mode), *taking off*, *flight* and *landing*.

Component-Based Development (CBD) is a design paradigm in which systems are built from pre-developed reusable components [1]. Since a component is built for reuse in different systems, no specific context should be assumed. The fundamental idea is to reduce development time by building systems from existing components.

Our focus is component-based development of real-time embedded software. Since operational mode is highly relevant for such software, our research is devoted to exploring efficient and predictable mode switch in component-based multi-mode systems (CBMMSs). A CBMMS has two distinctive features: (1) it is built in a component-based manner; (2) it supports multiple operational modes and can switch between different modes under certain conditions. Fig. 1 illustrates the hierarchical component structure of a typical CBMMS. From the top level, the system consists of three components: *a*, *b* and *c*. Component *b* is composed of three other components: *d*, *e* and *f*. With respect to the terminology of CBD, we can distinguish two basic types of components: (1) a *primitive component* contains executable software codes, and it cannot be further decomposed; (2) a *composite component* is a composition of other components. In Fig. 1, *a*, *c*, *d*, *e* and *f* are primitive components whereas *Top* and *b* are composite. The system supports two modes: $m1$ and $m2$. When the system is in $m1$ (Fig. 1 lower left), Component f is deactivated (suspended). In $m2$ (Fig. 1 lower right), f is activated (executing) while c and e are deactivated. In addition, Component a has different mode-specific behavior in $m1$ and $m2$ indicated by different shadings in Fig. 1.

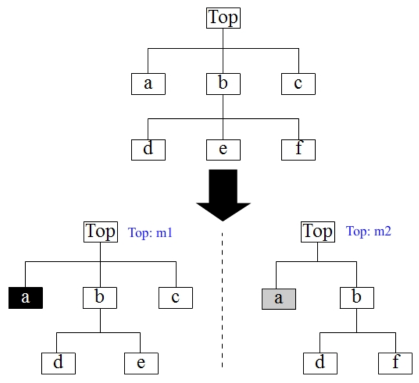

Figure 1. Component hierarchy in different modes

There are several important challenges that need to be addressed in providing a complete handling of modes and mode switches in a CBMMS. Related to our approach, these challenges include the following problems already explored:

- **Mode switch propagation:** A mode switch can be triggered by any component and other components must be made aware of the mode switch. Since a component has no global knowledge of the system component hierarchy, a mode switch event has to be stepwise propagated as provided by the mechanism in [2].
- **The guarantee of a consistent mode switch:** The mode switch of a system typically corresponds to mode switches of several components, which must be synchronized and coordinated to ensure that the system is in a consistent state when it completes a mode switch.

An initial solution for a limited setting is provided in [2], and extended in [3].

- **The mode mapping between different components:** Since components are developed independently, the components in a system may support different modes and the supported modes of different components must be consistently mapped, i.e. it must be unambiguously defined which mode in a particular component corresponds to which mode in another component. In [4], we present such a mode mapping mechanism.

together with the specific contributions of this paper:

- **Atomic component execution handling:** Our original mode switch mechanism (the Mode Switch Logic-MSL) assumes that the execution of each component is immediately aborted when a mode switch is triggered. This assumption is unrealistic because it ignores the case with executions that must be completed before a mode switch can be taken. As the first contribution of this paper, we extend our MSL by adding the support for atomic component execution.
- **Conflict handling for multiple mode switch triggering:** While a CBMMS is in the process of switching mode, a new mode switch triggering will incur conflict. Previously, we assumed that such conflicts never occur. As the second contribution, we provide an initial solution to this problem based on our extended MSL.
- **Mode switch timing analysis:** Many CBMMSs are also real-time systems, thus the mode switch time must be bounded and predictable. A very basic timing analysis is presented in [5]. As the third contribution of this paper we present a more general and thorough timing analysis for the mode switch of a CBMMS either with or without atomic component execution.

The remainder of the paper is organized as follows: Section II introduces some related work. Section III describes the system and component models as well as the assumptions made in this paper. Section IV extends our original MSL by adding the support for atomic component execution. In Section V, the global mode switch time is analyzed for CBMMSs either with or without atomic component execution. Section VI describes how to derive the extra mode switch latency introduced by atomic component execution by using the model checker UPPAAL [6]. Finally, we make our conclusion and discuss some future work in Section VII.

II. Related work

Mode switch (sometimes also called "Mode change") problems can be found in a multitude of related ongoing research on miscellaneous topics, a majority of which delve into multi-mode real-time systems, in particular the study of mode switch protocols and scheduling issues during mode switch (e.g. [7] and [8]). One of the earliest publications related to mode switch is by Sha et al. [9], who developed a simple mode switch protocol in a prioritized preemptive scheduling environment guaranteeing short and bounded mode switch latency. Later Real and Crespo [10] conducted a survey of different mode switch protocols and proposed several new protocols along with associated schedulability analysis. Mode switch protocols are also extended to multiprocessor platforms, e.g., in [11] and [12].

Phan et al. extend the traditional Real-Time Calculus to handle multi-mode [13], and present a multi-mode automaton model for modeling multi-mode applications, together with an interface-based technique for compositional analysis [14] and a semantic framework for mode switch protocols [15].

Several frameworks have been developed for the support of multi-mode systems, such as COMDES-II [16] and MyCCM-HI [17], and mode switch support has been provided for a few programming languages/models, such as AADL [18], Giotto [19] and TDL [20].

In general, the mode switch problem in CBD has been rarely explored, not to mention the consideration of atomic component execution in the multi-mode context. Atomic execution has mostly been considered as a type of real-time task whose current execution must be completed before a mode switch (e.g. in the mode change model in [8]).

III. System model

In this section, we first introduce the system and component models that we base our work on. We assume pipe-and-filter (data/control flow) type of executions, which are common in multimedia and process control systems, although we believe that our results have wider applicability. We will first introduce some notations, after which we present a model for component-based multi-mode pipe-and-filter systems that do not require atomic component execution. Then we extend the model to also cover atomic component execution.

A. Notations

A CBMMS consists of a set of hierarchically organized components. Let PC denote the set of primitive components and CC denote the set of composite components: $PC \cap CC = \emptyset$. The top component is denoted by Top. Let $\widetilde{CC}$ denote $CC \backslash \{Top\}$. For $c_i \in \widetilde{CC}$, P_{c_i} denotes the parent of c_i. As the set of subcomponents of c_i, SC_{c_i} is divided into two disjoint parts: the set of subcomponents ASC_{c_i} activated in the current mode and the set of subcomponents DSC_{c_i} deactivated in the current mode. $SC_{c_i} = ASC_{c_i} \cup DSC_{c_i}$.

B. The general system and component model

In [2], we have designed component models that support multi-mode and composable mode switch for both primitive and composite components. Each component has input and output ports through which it communicates with its neighboring components. In addition, components have dedicated ports used for mode switch handling. For each supported mode, there is a specific configuration of the component. The configuration is a collection of the mode-specific properties of a component. Some properties are application dependent while some others are general. For instance, the configuration of a primitive component usually includes its mode-specific behavior and the configuration of a composite component usually includes the running status (either activated or deactivated) of its subcomponents and active inner component connections. If a component is deactivated, it is not running in any mode but only responsive to mode related events. A component is able to reconfigure itself when switching mode. All components follow the same repeated execution pattern: waiting for inputs, processing data and producing outputs.

In this paper, we target pipe-and-filter systems. Just like the execution pattern of each component, a pipe-and-filter system waits for its input data, processes data and generates output data. Different components can process different data simultaneously. Fig. 2 depicts the component connection of the example given by Fig. 1. The component connection here is quite simple, i.e. there are no diverging or converging branches, or feedback loops.

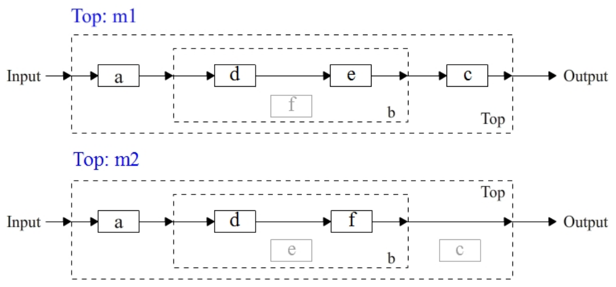

Figure 2. A component-based multi-mode pipe-and-filter system

A component has a non-empty set of input and output ports. 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).

C. The system model with atomic component execution

Atomic components cannot be interrupted by a mode switch at any point in time; they have to complete any ongoing execution before reconfiguration for the new mode. In addition to single atomic component, we will consider groups of components that are atomic, termed an Atomic Execution Group (AEG). All ongoing executions in the AEG have to be completed before reconfiguration, including processing of any data existing or produced by computations within the group. It should be noted that AEGs are mode specific, i.e., components that belong to the same AEG in one mode may belong to different AEGs in another mode; they may not even belong to any AEG in another mode. The following are properties/requirements of AEGs:

- If $c_i \in CC$ is in an AEG in one mode, then $\forall c_k \in ASC_{c_i}$ must be in the same AEG as well. For instance, in Fig. 2, if Component b is in an AEG when the top component is in $m1$, then d and e must be in the same AEG as well.
- An AEG can consist of a set of components. To simplify the handling mechanism and presentation, we assume that each AEG is represented by a single component c_i that is either primitive or composite.
- A system may have multiple AEGs at the same time, but there should be no overlapping between any two AEGs. It is allowed to have one AEG included by a bigger AEG, which then absorbs the smaller one.

In this paper, we make the following additional restrictions (assumptions) on the AEG:

- An AEG should have only one input port. The reason is that an AEG must freeze its input(s) (i.e. accept no more data) when it is told to switch mode. If an AEG has multiple input ports, it may receive data from different input ports at different times. Consequently, input freezing becomes a complex issue, which is out of the scope of this paper.
- There is no cyclic connection, i.e. feedback loop, in an AEG. Feedback loops complicate the data flow within an AEG as well as the mode switch timing analysis.
- Data transmission between different components within an AEG is instantaneous. An AEG is most likely to reside in the same physical (sub)system, therefore, compared with component execution time, data transmission time can be considered negligible (or included in the component execution time).

With respect to the mode switch timing analysis, we make two more assumptions:

- For each component, the response time of the mode switch handling is known and bounded, such that the timing of each step can be represented by a known constant maximal value, i.e., we do not here deal with issues related to scheduling or execution time estimation. This could also correspond to a fully parallel system with known execution times.
- When a mode switch is triggered, all components will switch mode. This implies that a mode switch is always

a global activity. In [4], it has been pointed out that some components may be unaffected by a mode switch event, and a mode switch can also be a local activity, i.e. within a non-top composite component. Yet the timing analysis for a local mode switch is left for future work.

IV. The extended Mode Switch Logic

According to our original MSL, the mode switches of different components are synchronized by the Mode Switch Request (MSR) propagation mechanism and mode switch dependency rule. An MSR is a signal telling a component to switch mode. A mode switch is triggered by a Mode Switch Source (MSS) which is the component that initiates the MSR. An MSS can be either a primitive or composite component, and a system can have multiple MSSs in each mode. The MSR from an MSS must be propagated to all other components of the system. This has been handled by the original MSR propagation mechanism [2]. If a system has no AEG, after receiving the MSR, each component will abort its current execution immediately and start its reconfiguration. However, components within an AEG must run to completion before reconfiguration. Here we propose a new mode switch (MS) propagation mechanism, which compared with the original MSR propagation mechanism, can cope with atomic component execution as well as resolve the conflict of multiple overlapping MSRs.

A. The extended MSL without atomic component execution

Let c_i denote an MSS with $c_j = P_{c_i}$. The MS propagation mechanism without considering atomic component execution works as follows:

MS propagation mechanism: *When c_i triggers a mode switch, it sends an MSR to c_j. As a composite component, c_j refers to the local mode mapping and makes a decision upon receiving the MSR:*

- *If the MSR does not imply any mode switch of c_j, then c_j approves the MSR by sending a Mode Switch Instruction (MSI) to SC_{c_j} based on its mode mapping.*
- *If the MSR implies the mode switch of c_j whose condition does not allow such a mode switch, then c_j rejects the MSR by doing nothing.*
- *If the MSR implies the mode switch of c_j whose condition allows such a mode switch and $c_j \neq Top$, then c_j will forward the MSR to its parent P_{c_j} and let P_{c_j} make further decisions. If this MSR is finally approved, c_j will receive an MSI from P_{c_j} and propagate the MSI to SC_{c_j} based on its mode mapping.*

$\forall c_k \in CC$, c_k handles an incoming MSR or MSI exactly in the same way as c_j. $\forall c_l \in PC$, c_l can only receive an MSI but does not propagate the MSI. An MSS is not blocked after issuing an MSR. If Top is an MSS, it can directly issue an MSI to SC_{Top}. The mode switch propagation is terminated when all components have received an MSI associated with the same MSR from an MSS.

The MS propagation mechanism divides the mode switch propagation process into two independent phases: the upstream MSR propagation (if the MSS is not Top) and the downstream MSI propagation (if the MSR is approved). Since we in this paper assume that a mode switch is always a global activity making all components switch mode, a mode switch decision of an MSR is always made by Top which issues the corresponding MSI. Since an MSI is propagated from a parent to children and the component hierarchy has a tree structure, the MSI is propagated to all components (each MSI from Top is sent to each component only once).

Each component starts its reconfiguration after its MSI propagation. The synchronization between the reconfiguration of different components is guided by our mode switch dependency rule [3]:

Mode switch dependency rule: *Each component starts its reconfiguration after its MSI propagation. $c_i \in PC$ sends a Mode Switch Completion (MSC) signal to P_{c_i} upon completion of its reconfiguration to indicate mode switch completion. $c_j \in CC$ completes its mode switch when its reconfiguration is completed and it has received the MSC(s) from all $c_k \in SC_{c_j}$. Thereafter, c_j sends an MSC to P_{c_j} if $c_j \neq Top$. A global mode switch is completed when Top completes its mode switch.*

The mode switch of the system in Fig. 1 is demonstrated in the left part of Fig. 3 (the timeline will be explained in the timing analysis later on), where *d* is the MSS and component reconfiguration is illustrated by black bars. Without atomic component execution, the global mode switch process is independent of component connections.

B. Atomic component execution handling

The basic idea of handling atomic component execution in a pipe-and-filter system is to guarantee that an AEG can complete processing all the data within the AEG before mode switch takes place. When a component is developed, it can be pre-defined that it is an AEG in certain mode(s) without considering the context during composition. The AEG component is able to monitor its Data Processing Status (DPS), which is either "Processing" or "Not processing". When some data is being processed within the AEG, its DPS is "Processing". Otherwise its DPS is "Not processing".

Our MS propagation mechanism and mode switch dependency rule still work with atomic component execution for most components. Only the AEG component needs a slight modification in its MSI propagation. Let $c_i \in CC$ denote an AEG component which has just started its MSI propagation. Essentially, $\forall c_k \in DSC_{c_i}$, the mode switch of c_k is not affected by atomic component execution because it is not running. The MSI propagation of c_i works as follows:

MSI propagation of an AEG component: *c_i refers to its local mode mapping and immediately propagates the MSI to DSC_{c_i}. If DPS_{c_i} ="Not processing", then there is no unfinished atomic component execution and c_i*

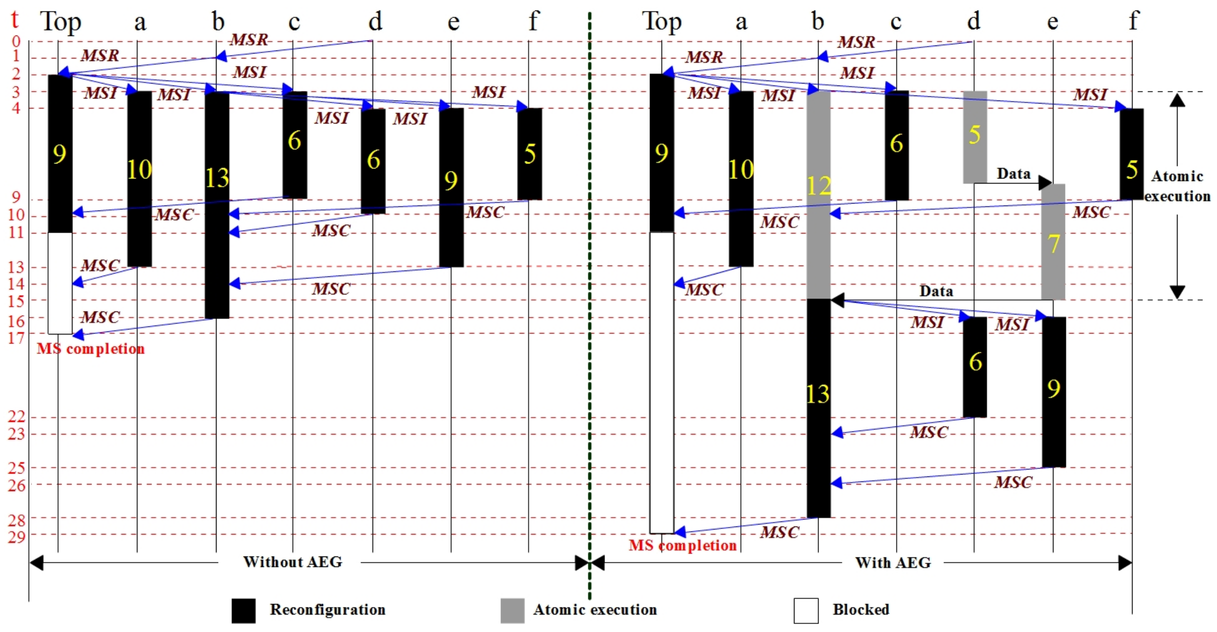

Figure 3. The global mode switch process and timing analysis

can immediately propagate the MSI to ASC_{c_i} as well. If DPS_{c_i} ="Processing", c_i will freeze its input so that no more data will enter the AEG. Then it lets its atomic execution complete while monitoring its DPS. When DPS_{c_i} turns to "Not processing", the atomic component execution in c_i has been completed and c_i can propagate the MSI to ASC_{c_i}. c_i must unfreeze its input to accept new input data upon its mode switch completion.

Our MS propagation mechanism does not exclude the case when an AEG is included in a bigger AEG. Since the outer AEG does not propagate the MSI until all data within it have been processed, the DPS of the enclosed AEG is always "Not processing" when it receives an MSI.

We can use the example in figures 1 and 2 to demonstrate the mode switch of a system with an AEG. Suppose the system is in $m1$ and Component b is an AEG. Then $ASC_b = \{d, e\}$ and $DSC_b = \{f\}$. Component d is still assumed to be the MSS. The right part of Fig. 3 illustrates the scenario when b receives the MSI in a situation when some data has just entered the AEG via the input port of b and there is no other data in the AEG. In the figure, atomic component execution is represented by grey bars. The global mode switch process can be described as follows:

1) d sends an MSR to b, who forwards the MSR to Top.
2) Top approves the MSR and propagates an MSI to its subcomponents a, b, and c.
3) Top, a and c start to reconfigure themselves. As a composite component and an AEG, b first sends an MSI to its deactivated subcomponent f which will initiate its reconfiguration. Then b checks its DPS. Since some data just entered b, DPS$_b$ ="Processing". Therefore, b freezes its input port and monitors DPS$_b$.
4) b finds that DPS$_b$ becomes "Not processing" as the data is completely processed by the AEG. b propagates the MSI to its two activated subcomponents d and e, who will start their reconfigurations.
5) All the components follow the dependency rule introduced in Section IV-A. When b completes its mode switch, it will unfreeze its input.

By comparing the left and right parts of Fig. 3, we can see that atomic component execution changes the global mode switch process only by adding a bounded delay during the MSI propagation of an AEG component. Alg. 1 implements the MS propagation mechanism of $c_i \in \widetilde{CC}$ with or without AEGs. The special case when c_i is an MSS, a primitive component or Top can be easily deduced. This algorithm can be considered as a separate task dedicated to handling the mode switch of c_i, thus an ongoing atomic execution will not be interrupted by an incoming MSR or MSI. A few points deserve further explanation in the algorithm:

- p^{MSX} and p^{MSX}_{in} are the mode switch dedicated ports of c_i, the former for the communication with P_{c_i} and the latter for the communication with SC_{c_i}. MSX stands for either MSR or MSI.
- *Wait* and *Signal* are primitives for receiving and sending MSR or MSI via p^{MSX} or p^{MSX}_{in}. MSR is sent via p^{MSX} and received via p^{MSX}_{in} while MSI is sent via p^{MSX}_{in} and received via p^{MSX}.
- *MSX(y,z)* is the MSR or MSI carrying the new target mode y of the receiving component and the identity z of the sending component.
- After reconfiguration, c_i follows the mode switch de-

pendency rule which is not affected by AEGs and therefore omitted in the algorithm.

Algorithm 1 $MS\_Propagation\_AEG(c_i)$

```
loop
  repeat
    $Wait(p^{MSX} \vee p_{in}^{MSX}, MSX(m_{origin}^{new}, origin))$;
    if $MSR$ then
      $Mode\_Mapping$;
      $Signal(p^{MSX}, MSR(m_{c_i}^{new}, c_i))$;
    end if
  until $MSI$
  $Mode\_Mapping$;
  if $AEG$ then
    $\forall c_k \in DSC_{c_i} : Signal(p_{in}^{MSX}, MSI(m_{c_k}^{new}, c_i))$;
    if $DPS_{c_i} = Processing$ then
      FreezeInput;
      waituntil $DPS_{c_i} = \neg Processing$;
    end if
    $\forall c_k \in ASC_{c_i} : Signal(p_{in}^{MSX}, MSI(m_{c_k}^{new}, c_i))$;
  else
    $\forall c_k \in SC_{c_i} : Signal(p_{in}^{MSX}, MSI(m_{c_k}^{new}, c_i))$;
  end if
  Reconfiguration;
  $\cdots\cdots$;
  UnfreezeInput;
end loop
```

No matter whether a component-based multi-mode pipe-and-filter system has an AEG or not, the data being processed by the system during a mode switch must be properly handled, though it is not critical for the global mode switch process. Just like an AEG, Top should always freeze its input as it propagates an MSI, and unfreeze its input upon mode switch completion, in that the system is in an unstable state during a mode switch and the processing of new incoming data should be avoided. For the same reason, data communication between components should be disabled until the global mode switch completion. Reconfiguration can change component connection and running status, thus making data communication unpredictable. One simple solution is to delay component execution in the new mode until Top propagates a *Start_to_execute* signal after the global mode switch. Moreover, after an AEG freezes its input, the component directly connected to its input may still send some data. Depending on the system requirements, it may or may not be permissible to discard the data, hence a temporary input buffer of the AEG might be required to store such data.

C. Conflict handling for multiple mode switch triggering

Our original MSL assumes that the interval between two different MSXs is long enough so that a new MSX will not be issued before mode switch completion. However, if a CBMMS has multiple MSSs in one mode, their MSXs may interfere with each other. Actually, even the same MSS may initiate two consecutive MSXs such that the second one interferes with the first one.

Our MS propagation mechanism can resolve the conflicts due to overlapping MSXs. For a global mode switch, since a mode switch decision is always made by Top, an arbitration mechanism can be applied in Top when it receives multiple overlapping MSRs or an incoming MSR is in conflict with its MSI if it is one MSS. Based on the arbitration mechanism, Top can refrain from issuing a new MSI before the completion of an ongoing mode switch. Depending on the desired behavior of a particular CBMMS, different arbitration mechanisms can be used. For instance, during a mode switch, a new MSR may be discarded by Top, or the new MSR may be temporarily delayed until completion of the current mode switch. Top can even set priorities to different MSRs and make arbitration decisions accordingly.

V. Mode switch timing analysis

Since many CBMMSs are also real-time systems, it is important that the system timing constraints can be verified. In this section we present mode switch timing analysis for CBMMSs. We are interested in the global mode switch time, which starts when an MSS initiates an MSR and ends when Top completes its mode switch. We start by analyzing the global mode switch time without considering atomic component execution and then extend this timing analysis to also cover atomic component execution. We still follow the assumption that an MSI always originates from Top and is eventually propagated to all components.

A. The mode switch timing analysis without AEG

For a CBMMS without any AEG, the global mode switch time can be derived from the reconfiguration times of individual components. We first list some key timing factors and corresponding notations:

- t_{MSR}, t_{MSI} and t_{MSC}: The transmission time of an MSR, MSI or MSC, respectively.
- RCT_{c_i}: The reconfiguration time of Component c_i. We assume all scheduling effects are included in RCT_{c_i}.
- MS_{c_i}: The mode switch time of Component c_i. If $c_i \in PC$, then $MS_{c_i} = RCT_{c_i}$. However, if $c_i \in CC$, its reconfiguration completion does not have to result in mode switch completion, ergo $MS_{c_i} \geq RCT_{c_i}$.

We will approximate the possibly varying transmission times (t_{MSR}, t_{MSI} and t_{MSC}) with constants corresponding to their maximum values. In calculating the global mode switch time, the timing analysis of MSR propagation can be performed separately. The total MSR propagation time T_{MSR} is proportional to the hierarchical distance L between the MSS and Top, with $L = 0$ if the MSS is Top. Therefore, T_{MSR} can be easily calculated as

$$T_{MSR} = t_{MSR} * L \quad (1)$$

Fig. 4 depicts a simple MSI propagation scenario within a composite component a, where a broadcasts an MSI to its subcomponents b and c. The black bar represents the reconfiguration time of a, while the linear gradient bars represent the mode switch times of b and c respectively. Since a is the MSI sender, the mode switch starting times of its subcomponents b and c are both delayed for the transmission time of one MSI, i.e. t_{MSI}. After MSI propagation, the reconfigurations of these components can be taken in parallel. According to the mode switch dependency rule, two conditions must be satisfied for the mode switch completion of a: (1) a must complete its own reconfiguration; (2) a must receive the MSCs from both b and c. The calculation of MS_a boils down to selecting the maximum of three timing points: (1) the reconfiguration completion time of a; (2) the time when a receives an MSC from b; (3) the time when a receives an MSC from c. Hence,

$$MS_a = max\{RCT_a, t_{MSI} + MS_b + t_{MSC}, \\ t_{MSI} + MS_c + t_{MSC}\} \tag{2}$$

In Fig. 4, a finally completes its mode switch as it receives an MSC from c. Thus a is blocked for a while after its reconfiguration. In general, for $c_i \in CC$,

$$MS_{c_i} = max\{RCT_{c_i}, max_{c_k \in SC_{c_i}}\{t_{MSI} + MS_{c_k} + t_{MSC}\}\} \tag{3}$$

MS_{c_k} in Eq. 3 actually reveals the recursive nature of the global mode switch time calculation. Since $\forall c_k \in SC_{c_i}$, the MSI propagation of c_k is the same as c_i, MS_{c_k} can be calculated exactly in the same way as MS_{c_i}. This recursion is terminated by $c_j \in PC$ with $MS_{c_j} = RCT_{c_j}$. As a result, starting from primitive components, the global mode switch time is calculated from lower levels to higher levels.

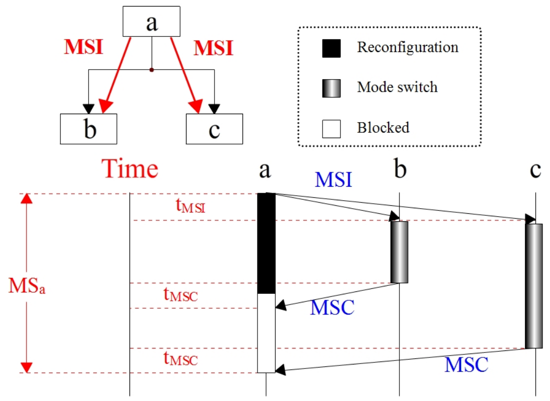

Figure 4. Mode switch timing analysis within a composite component

Since the global mode switch is completed upon the mode switch completion of Top, the global mode switch time MS is simply the sum of the MSR propagation time T_{MSR} and the mode switch time of Top, which is MS_{Top}:

$$MS = T_{MSR} + MS_{Top} \tag{4}$$

Now let's try to analyze the MS of the example from Fig. 1, where Component d is assumed to be the MSS. We assume that $t_{MSR} = t_{MSI} = t_{MSC} = 1$. The left part of Fig. 3 shows the entire mode switch process of the system without AEG. The reconfiguration time of each component is shown on each black reconfiguration bar. Obviously, b and Top are the only two composite components in charge of the MSI propagation described by Fig. 4. The first step is to calculate T_{MSR}:

$$T_{MSR} = t_{MSR} * L = 1 * 2 = 2 \tag{5}$$

The next step is to calculate MS_b:

$$\begin{aligned} MS_b = max\{&RCT_b, max\{t_{MSI} + MS_d + t_{MSC}, \\ &t_{MSI} + MS_e + t_{MSC}, t_{MSI} + MS_f + t_{MSR}\}\} \\ &= max\{13, max\{1+6+1, 1+9+1, 1+5+1\}\} \\ &= 13 \end{aligned} \tag{6}$$

Based on MS_b, MS_{Top} is calculated in the same way:

$$\begin{aligned} MS_{Top} = max\{&RCT_{Top}, max\{t_{MSI} + MS_a + t_{MSC}, \\ &t_{MSI} + MS_b + t_{MSC}, t_{MSI} + MS_c + t_{MSC}\}\} \\ &= max\{9, max\{1+10+1, 1+13+1, 1+6+1\}\} \\ &= 15 \end{aligned} \tag{7}$$

Thus, just as Fig. 3 indicates, the global mode switch time

$$MS = T_{MSR} + MS_{Top} = 2 + 15 = 17 \tag{8}$$

B. The mode switch timing analysis with AEG(s)

For an AEG component c_i. If $c_i \in PC$, its reconfiguration can be delayed by its atomic execution AE_{c_i}. If $c_i \in CC$, the MSI propagation time can be delayed by AE_{c_i}. In the best case $AE_{c_i} = 0$, but we are more interested in the worst case. We shall focus on the case when $c_i \in CC$.

The global mode switch time without considering atomic component execution is defined by equations 1, 3 and 4 in Section V-A. To cater for atomic component execution, we extend Eq. 3 into the following

$$\begin{cases} c_i \neq \text{AEG} \Rightarrow max\{RCT_{c_i}, max_{c_k \in SC_{c_i}}\{t_{MSI} + \\ \quad MS_{c_k} + t_{MSC}\}\} \\ c_i = \text{AEG} \Rightarrow max\{RCT_{c_i} + AE_{c_i}, \\ \quad max_{c_k \in ASC_{c_i}}\{t_{MSI} + MS_{c_k} + t_{MSC} + AE_{c_i}\}, \\ \quad max_{c_k \in DSC_{c_i}}\{t_{MSI} + MS_{c_k} + t_{MSC}\}\} \end{cases} \tag{9}$$

Eq. 9 is consistent with Alg. 1, where $\forall c_k \in DSC_{c_i}$, c_k can receive an MSI immediately from c_i. This is why AE_{c_i} only contributes to the timing of c_i and ASC_{c_i} in Eq. 9. We consider AE_{c_i} as the worst-case atomic component execution time in the AEG c_i, thus components at lower levels in the same AEG do not need to consider AE_{c_i} again. If a system has multiple AEGs, then each AEG has its own AE that needs to be calculated independently. In the next section we will show how to obtain AE.

But first we will analyze the global mode switch time demonstrated in the right part of Fig. 3. We assume that the reconfiguration and transmission times are the same as the left part of Fig. 3. Then T_{MSR} is still 2. If the worst-case data processing times of d and e are 5 and 7, respectively, then in the scenario above, $AE_b = 12$, i.e. the sum of both.

Now let's analyze MS using Eq. 9, starting with MS_b:

$$\begin{aligned} MS_b = max\{&RCT_b + AE_b, max\{t_{MSI} + MS_d + \\ &t_{MSC} + AE_b, t_{MSI} + MS_e + t_{MSC} + AE_b, \\ &t_{MSI} + MS_f + t_{MSC}\}\} \\ &= max\{13 + 12, max\{1 + 6 + 1 + 12, 1 + 9 + \\ &1 + 12, 1 + 5 + 1\}\} = 25 \end{aligned} \tag{10}$$

Then MS_b is used to calculate MS_{Top}:

$$\begin{aligned} MS_{Top} = max\{&RCT_{Top}, max\{t_{MSI} + MS_a + t_{MSC}, \\ &t_{MSI} + MS_b + t_{MSC}, t_{MSI} + MS_c + t_{MSC}\}\} \\ &= max\{9, max\{1 + 10 + 1, 1 + 25 + 1, 1 + 6 + 1\}\} \\ &= 27 \end{aligned} \tag{11}$$

Finally, MS is obtained by

$$MS = T_{MSR} + MS_{Top} = 2 + 27 = 29 \tag{12}$$

Hence, the global mode switch time is 29, which is consistent with Fig. 3. The extra delay due to atomic component execution is substantial, which indicates that AEGs should be kept as small as possible.

VI. Using UPPAAL to determine the worst-case latency of an AEG during mode switch

In this section, we demonstrate how to determine the worst-case atomic execution time of an AEG (denoted AE) during a mode switch by using the model-checking tool UPPAAL [6]. AE is then used in calculating the mode switch time in Eq. 9.

A. The worst-case latency of an AEG during mode switch

AE_b of the AEG (Component b) in Fig. 3 can be easily derived. However, AE of an AEG component in general can be affected by many contributing factors such as the component hierarchy, component connections, and data flow.

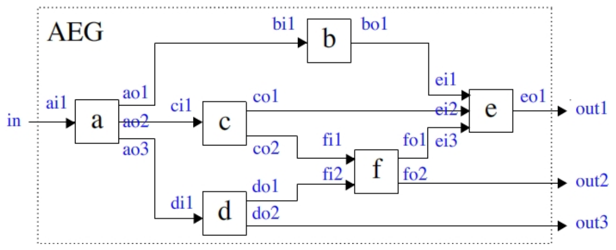

Figure 5. A complex AEG

Fig. 5 depicts a more complex AEG consisting of primitive components *a-f*. Composite components or deactivated components in the AEG are not shown because they do not affect AE. We assume that the data processing time of each component $c_i \in PC$ in the AEG is bounded by a timing interval [$C_{c_i}^{min}$,$C_{c_i}^{max}$], and that the incoming data rate of the AEG is within the interval [R_{min},R_{max}]. When the AEG receives an MSI, all the data within the AEG must be completely processed before the AEG can start its MSI propagation. The worst-case value of this data processing time equals AE. 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 (further details in Section VI-C), 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.

To calculate AE, we propose a model-checking approach based on UPPAAL.

UPPAAL is a model-checking tool which is widely used to model, simulate and verify real-time systems. In particular, since version UPPAAL 4.1.3, there is a "sup" operator able to find out the maximal value of a variable or clock. If an AEG is properly modeled by UPPAAL, we can use "sup" to obtain AE. In this way, the focus is moved from AE calculation to the UPPAAL modeling of a given AEG.

B. UPPAAL modeling

By using UPPAAL, we first model the mode switch behavior of an AEG and then derive AE via its property verification. No matter how complex an AEG is, we can divide its UPPAAL model into four parts:

1) *Data source*: generates data at a flexible rate.
2) 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 $n = 0$.
3) *Data forwarder*: forwards data between components without the sender knowing 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.

4) *Primitive components*: modeled by a UPPAAL template for each of them. Though the number of components can be arbitrary, a parameterized generic model applying to all components can be used.

The full UPPAAL model of the mode switch handling in the AEG component in Fig. 5 is available in [21]. Here we will only focus on the AEG model (Item 2 above) and its enclosed primitive components.

Fig. 6 models the AEG. When a new data enters the AEG, the AEG will propagate this data to its subcomponents through the *dataOut!* channel. *dataCounter* counts the number of data within the AEG. In the function *shutDS()*, the AEG compares *dataCounter* with the threshold N. If the threshold is reached, the data source will be turned off in *shutDS()*. When *dataCounter* decreases and no MSI arrives, the data source is turned on in function *dataNControl()*. The model has two major states: **Waiting** and **Processing**. In state **Waiting**, no data is within the AEG, i.e. if an MSI arrives, the mode switch will not be delayed by atomic component execution, hence AE=0. In state **Processing**, at least one data is in the AEG. The MSI is modeled by the channel *MSI?*. Clock z plays a significant role in that the maximal possible value of z in its State **Processing** corresponds to AE.

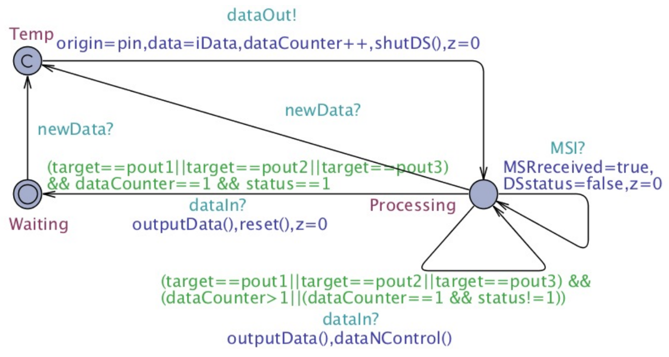
dataOut!
Temp
origin=pin,data=iData,dataCounter++,shutDS(),z=0
C
newData?
newData?
MSI?
MSReceived=true,
DSstatus=false,z=0
(target==pout1||target==pout2||target==pout3)
&& dataCounter==1 && status==1
dataIn?
outputData(),reset(),z=0
Processing
Waiting
(target==pout1||target==pout2||target==pout3) &&
(dataCounter>1||(dataCounter==1 && status!=1))
dataIn?
outputData(),dataNControl()

Figure 6. UPPAAL model: AEG

Fig. 7 illustrates the model of Component *f*, which has multiple inputs and outputs. Actually the model of f is generic in the sense that all the other components in the AEG from Fig. 5 can be modeled in the same way. When not processing any data, *f* is in state **nonProcessing**. When 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 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*.

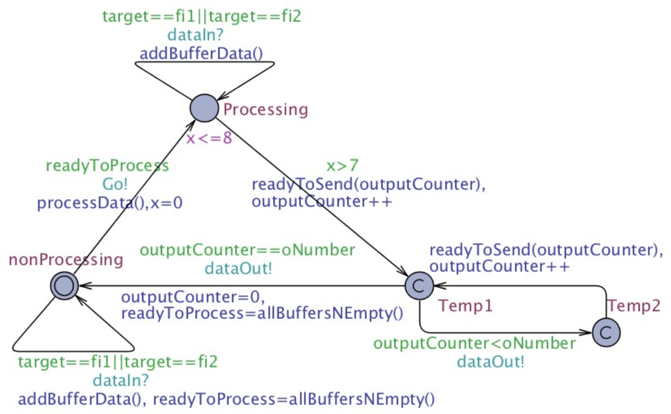

Figure 7. UPPAAL model: Component *f*

C. Verification and evaluation

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. In addition, the verification results show that for 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, $AE = 40$.

Moreover, the verification results show that R and C_{c_i} ($c_i \in PC$ is activated in the AEG) 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. The most important results are summarized in Table I[1].

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. For instance, the maximal usage of the buffer associated with port $ei2$ in Fig. 5. is 1 when R=[10,12], 2 when R=[8,10], and 4 when R=[6,8].

D. Generalization

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. Although we have only demonstrated how to derive AE for 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 the corresponding UPPAAL models can be automatically generated. Since the UPPAAL verification is based on generating and exploring the global state space, it is subject to state explosion while modeling a too complex system. However, we do not expect this to be a limitation in practice, since the complexity of an AEG typically is rather low.

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 I
PROPERTY VERIFICATION RESULTS FOR DIFFERENT DATA RATES

VII. CONCLUSION AND FUTURE WORK

In this paper we extend our original Mode Switch Logic (MSL) by adding the support for atomic component execution and providing an associated global mode switch timing analysis. We introduce a real-time model-checking solution to derive the worst-case data processing time of an Atomic Execution Group (AEG). The behavior of an AEG can be modeled with ease, regardless of its complexity, and its worst-case data processing time can be calculated.

Future work includes to implement support for automatic generation and verification of UPPAAL timing models, lift the assumption that an AEG is represented by a single component, and investigate the handling of feedback loops. Furthermore, we intend to explore and evaluate different approaches to conflict handling for multiple mode switch triggering together with its mode switch timing analysis.

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

ACKNOWLEDGMENT

This work is supported by the Swedish Research Council.

REFERENCES

[1] I. Crnkovic and M. Larsson, *Building reliable component-based software systems*. Artech House, 2002.
[2] Y. Hang, E. Borde, and H. Hansson, “Composable mode switch for component-based systems,” in *APRES ’11*, pp. 19–22.
[3] Y. Hang and H. Hansson, “A mode switch logic for component-based multi-mode systems,” MRTC, Mälardalen University, Tech. Rep. 261/2012, Jan 2012.
[4] ——, “A mode mapping mechanism for component-based multi-mode systems,” in *CRTS’11*, pp. 38–45.
[5] ——, “Timing analysis for a composable mode switch,” in *The Work-in-Progress session of ECRTS’11*, pp. 15–18.
[6] K. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a nutshell,” *STTT-Intl. J. on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.
[7] K. W. Tindell, A. Burns, and A. J. Wellings, “Mode changes in priority pre-emptively scheduled systems,” in *RTSS’92*, pp. 100–109.
[8] P. Pedro and A. Burns, “Schedulability analysis for mode changes in flexible real-time systems,” in *ECRTS’98*, pp. 172–179.
[9] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham, “Mode change protocols for priority-driven preemptive scheduling,” *Real-Time Systems*, vol. 1, pp. 243–264, 1989.
[10] J. Real and A. Crespo, “Mode change protocols for real-time systems: A survey and a new proposal,” *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.
[11] V. Nélis, J. Goossens, and B. Andersson, “Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms,” in *ECRTS’09*, pp. 151–160.
[12] P. M. Yomsi, V. Nelis, and J. Goossens, “Scheduling multi-mode real-time systems upon uniform multiprocessor platforms,” in *ETFA’10*.
[13] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan, “A multi-mode real-time calculus,” in *RTSS’08*, pp. 59–69.
[14] L. T. X. Phan, I. Lee, and O. Sokolsky, “Compositional analysis of multi-mode systems,” in *ECRTS’10*, pp. 197–206.
[15] ——, “A semantic framework for mode change protocols,” in *RTAS’11*, pp. 91–100.
[16] X. Ke, K. Sierszecki, and C. Angelov, “COMDES-II: A component-based framework for generative development of distributed real-time control systems,” in *RTCSA’07*.
[17] E. Borde, G. Haïk, and L. Pautet, “Mode-based reconfiguration of critical software component architectures,” in *DATE’09*, pp. 1160–1165.
[18] P. H. Feiler, D. P. Gluch, and J. J. Hudak, “The architecture analysis & design language (AADL): An introduction,” Software engineering institute, MA, Tech. Rep. CMU/SEI-2006-TN-011, Feb. 2006.
[19] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming,” in *PROCEEDINGS OF THE IEEE*, 2001, pp. 166–184.
[20] J. Templ, “TDL specification and report,” University of Salzburg, Tech. Rep., Nov. 2003.
[21] Y. Hang and H. Hansson, “A UPPAAL model for timing analysis of atomic execution in component-based multi-mode systems,” MRTC, Mälardalen University, Tech. Rep., Feb 2012.