A Mode Switch Logic for component-based multi-mode systems

Yin Hang
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
young.hang.yin@mdh.se

Hans Hansson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
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. A classical approach to reduce embedded systems design and run-time complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems. In addressing this issue, we have developed a Mode Switch Logic (MSL) for component-based multi-mode systems, implementing seamless coordination and synchronization of mode switch in systems composed of independently developed components.

1 Introduction

Traditionally, partitioning system behaviors into different operational modes has been used to reduce complexity and improve resource efficiency. Each mode corresponds to a specific system behavior. The system can start by running in a default mode and switches to another appropriate mode when some condition changes. In this way, the complexity of both system design and verification can be reduced while system execution efficiency is improved. A typical multi-mode system is the control software of an airplane, which e.g. could run in *taxi* mode (the initial mode), *taking off* mode, *flight* mode and *landing* mode.

There are a variety of alternatives to design and develop a multi-mode system. We set our focus on Component-Based Development (CBD), a promising solution for the development of embedded systems. CBD boasts quite a number of appealing features such as complexity management, increased productivity, higher quality, faster developing time, lower maintenance costs and reusability [3]. The most adorable feature of CBD for us is its component reuse idea, which allows us to build a system by reusable components, i.e. a system does not have to be developed from scratch, instead, some of its components or subsystems may be directly obtained from a repository of pre-developed components.

Our focus is component-based multi-mode systems (CBMMSs), i.e. multi-mode systems built by a set of hierarchically organized components. Just as what its name indicates, 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 circumstances. Figure 1 illustrates the hierarchical component structure of a typical CBMMS that will be used throughout this report. 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* is directly made by software codes, thus it cannot be further decomposed into other components; (2) A *composite component* is the composition of other components. Obviously, in Figure 1, a, c, d, e and f are primitive components whereas *Top* and b are composite components. Since the component hierarchy of the system has a tree structure, the subcomponents of a composite component at one level down can be called children and this composite component is called parent. Moreover, the system supports two modes: $m1$ and $m2$. When the system is in $m1$, Component f is deactivated (invisible in $m1$ in Figure 1). By contrast, when the system is in $m2$, f is activated whilst c and e become deactivated. Besides, Component a has different mode-specific behaviors presented by black and grey colors in Figure 1.

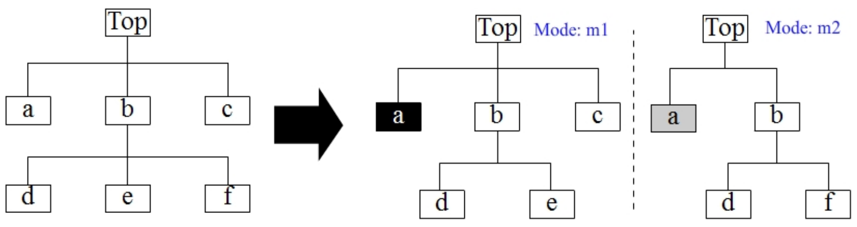

Figure 1: A component-based multi-mode system

Figure 1 presents the component hierarchy of the system, yet without showing how these components are connected. A component must have ports to communicate with another component. Depending on the system software architecture and system functionalities, components can be connected in many different ways. Figure 2 presents the component connection of the system in Figure 1. As a simple example, each component has only one input port and one output port denoted by blue texts and there is only one-to-one connection with single direction. Actually this is a typical pipe-and-filter system, which waits for some input data, processes the data and then generates the output data. Component functionality is presented by the red texts. The input data is an integer i and each component performs simple numerical calculation. For instance, Component a calculates $i*10$ in mode $m1$ and $i*20$ in mode $m2$. This is consistent with Figure 1 as Component a has different mode-specific behaviors in these two modes. The consistence is also reflected from deactivated components, which are not connected to any component in the corresponding mode.

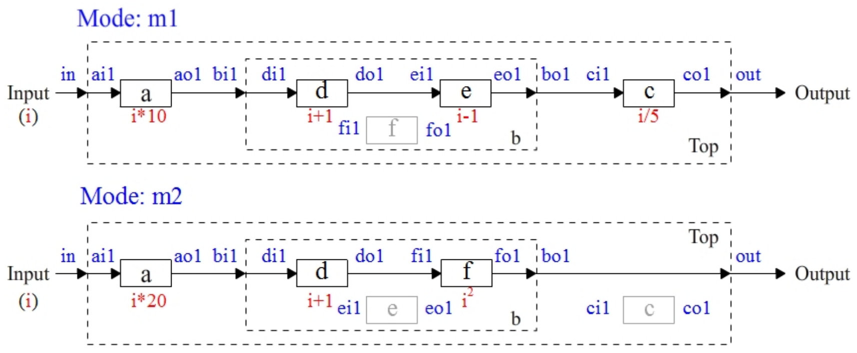

Figure 2: Component connections in different modes

From the example in Figure 2, we can see that the system behavior of a CBMMS is highly dependent on its components in each mode. This dependency also holds during a mode switch. When a system switches from one mode to another mode, its mode switch is essentially achieved by the joint mode switches of certain components. The central issue is that the mode switches of different components must be coordinated and synchronized to achieve a successful and efficient global mode switch. Notwithstanding that there is plenty of research work dealing with mode switch, little attention has been paid to this composable mode switch problem. To this end, we have developed a Mode Switch Logic (MSL) for CBMMSs.

2 Related work

Mode switch problems (sometimes also called ”Mode change”) can be found in a multitude of related ongoing research works 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. One of the earliest publications related to mode switch is by Sha et al. [18], who developed a simple mode switch protocol in a prioritized preemptive scheduling environment guaranteeing short and bounded mode switch latency. Later Real and Crespo [16] conducted a survey of different mode switch protocols and proposed several new protocols along with associated schedulability analysis. Protocols for symmetrical multiprocessor platforms are presented in [11], and extended to uniform multiprocessor platforms in [23]. There are also a number of papers, e.g. [21] and [12], targeting mode switch schedulability analysis, including EDF scheduling for multi-mode real-time systems [1] [19] [10].

In addition, Phan et al. study multi-mode real-time systems from a different perspective. They extend the traditional Real-Time Calculus (RTC) into multi-mode RTC to determine typical system properties [14]. They also present a multi-mode automaton model for modeling multi-mode applications and an interface-based technique for their compositional analysis [15]. Their most recent work presents a semantic framework for mode switch protocols [13].

Several frameworks have been developed for the support of multi-mode systems, such as COMDES-II [9] and MyCCM-HI [2]. Moreover, mode switch can be supported by a few programming languages/-models, such as AADL [4], Giotto [8] and TDL [20] (implemented in the Ptolemy II framework [17]).

Compared with these existing works, the research on composable mode switch of CBMMSs is relatively new and also demanding. In [5], we proposed the original MSL for CBMMSs, which serves as the basis of this report. Based on the original MSL, a preliminary timing analysis of the global mode switch was provided in [7]. One unrealistic assumption that has been made in our original MSL is that all components support the same modes. In [6], this assumption was lifted and a mode mapping mechanism was proposed to handle the mode incompatibility problem. In this report, the original MSL will be updated and explained in detail.

3 The mode-aware component model

Dozens of component models have already been proposed to date. However, most of them do not support multiple modes. The COMDES-II component model considers multiple modes and mode switch, but provides no mechanism guiding different components to realize the composable mode switch. Here we propose a new mode-aware component model tailored to our MSL.

Figure 3 illustrates the mode-aware component model for both primitive and composite components. A few points need to be mentioned:

- A component typically has one or more input and output ports. Each port is externally connected to a neighboring component, its parent or internally connected to one subcomponent.
- For a primitive component, we introduce a mode switch dedicated port p^{MSR}(MSR will be explained in the next section) to communicate with its parent during a mode switch. For a composite component, we introduce two mode switch dedicated ports, during a mode switch, one is p^{MSR} for the communication with its parent and the other is p^{MSR}_{in} for the communication with its subcomponents.
- For both primitive and composite components, the internal MSL defines how a component performs its own mode switch and also controls its own mode switch behavior.
- A component has predefined configuration for each mode. The configuration of a primitive component consists of its running status (*activated* or *deactivated*) and mode-specific behavior. The configuration of a composite component consists of its running status (also *activated* or *deactivated*), activated subcomponents, and active inner component connections. In each mode, only activated

components are running, while deactivated components are temporarily unavailable. Likewise, in each mode, only active component connections are considered. A connection becomes inactive when it is disconnected due to a mode switch. The mode switch of a component corresponds to its reconfiguration.

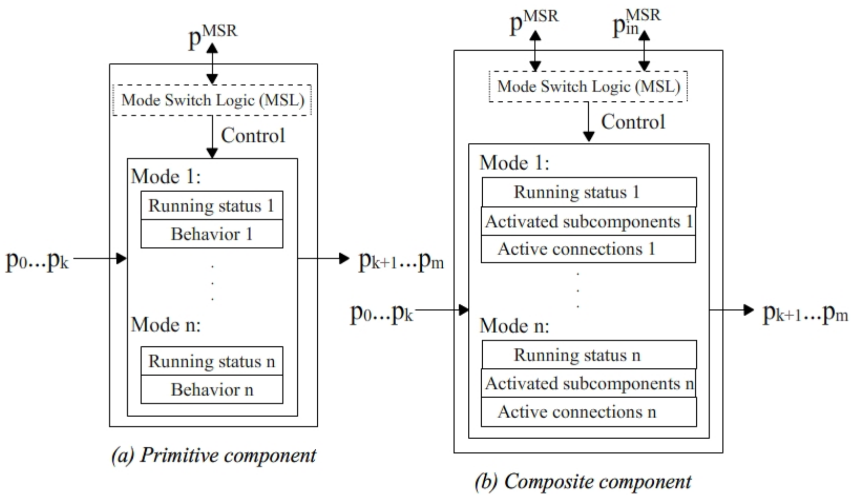

Figure 3: The mode-aware component model

Based on this mode-aware component model, both primitive and composite components can be formally defined. If we define *PC* as primitive component and *CC* as composite component, then each $c \in PC$ is a tuple:

$$< P, M, B, MB, S, MSL >$$

where P is the set of ports (the mode switch dedicated ports are implicit and thus not included) of c partitioned into the disjoint subsets of input ports P_{in} and output ports P_{out}; M is the set of operational modes supported by c; B is the set of mode-specific behaviors of c; the function $MB : M \rightarrow B$ defines the behavior associated with a certain mode; the function $S : M \rightarrow \{Activated, Deactivated\}$ indicates the running status of c in a certain mode; and MSL is the mode switch logic integrated in c, which will be described as algorithms in later sections.

Similarly, each $c \in CC$ is a tuple:

$$< P, SC, Con, M, S, ASC, ACon, MSL >$$

where P is the set of ports of c (defined in the same way as $c \in PC$); SC is the set of subcomponents of c; $Con \subseteq (P_{sub} \cup P) \times (P_{sub} \cup P)$ is the set of connections between the subcomponents in SC and connections between c and SC, where P_{sub} is the set of ports of SC:

$$P_{sub} = \bigcup_{\substack{comp \in SC \\ p \in P_{comp}}} p;$$

M is the set of operational modes supported by c; the function $S : M \rightarrow \{Activated, Deactivated\}$ indicates the running status of c in each mode $m \in M$; the function $ASC : M \rightarrow 2^{SC}$ defines the set of activated subcomponents in each mode; the function $ACon : M \rightarrow 2^{Con}$ defines the set of active connections (connections in use) in each mode; and MSL is the mode switch logic integrated in c. Now let's take Component c and b in Figure 2 as examples. As a primitive component, Component c is defined by the tuple:

$$< P_c, M_c, B_c, MB_c, S_c, MSL_c >$$

where MSL_c can be separately described as an algorithm and:

$$\begin{aligned}
P_c &= \{ci1, co1\} \\
M_c &= \{m1, m2\} \\
B_c &= \{B_c(m1)\} \\
MB_c &= \{m1 \to B_c(m1), m2 \to \emptyset\} \\
S_c &= \{m1 \to \mathit{Activated}, m2 \to \mathit{Deactivated}\}
\end{aligned}$$

Since Component c is deactivated in mode $m2$, it has no mode-specific behavior in this mode, i.e. $m2 \to \emptyset$. By contrast, as a composite component, Component b is defined by the tuple:

$$< P_b, SC_b, Con_b, M_b, S_b, ASC_b, ACon_b, MSL_b >$$

where MSL_b can be separately described as an algorithm and:

$$\begin{aligned}
P_b &= \{bi1, bo1\} \\
SC_b &= \{d, e, f\} \\
Con_b &= \{(bi1, di1), (do1, ei1), (eo1, bo1), (do1, fi1), (fo1, bo1)\} \\
M_b &= \{m1, m2\} \\
S_b &= \{m1 \to \mathit{Activated}, m2 \to \mathit{Activated}\} \\
ASC_b &= \{m1 \to \{d, e\}, m2 \to \{d, f\}\} \\
ACon_b &= \{m1 \to \{(bi1, di1), (do1, ei1), (eo1, bo1)\}, m2 \to \{(bi1, di1), (do1, fi1), (fo1, bo1)\}\}
\end{aligned}$$

Note that, each pair of connections (e.g., $(do1, ei1)$) implies a data flow from first to second element (e.g., $do1$ to $ei1$).

4 The MSR propagation mechanism

To simplify the problem, in this report we assume that the system and all its components support the same modes. For instance, when the system is in mode $m1$, all its components are also running in $m1$. Thus the triggering of a mode switch will contribute to a global activity. Theoretically speaking, a mode switch can be triggered by any component of a CBMMS. This event must be propagated to all the other components by some kind of signal, which is called Mode Switch Request (MSR):

Definition 1. *Mode Switch Request (MSR) is a signal telling each component to switch mode. The MSR itself contains information on the current MSR sender and the target mode which the receiver should switch to. It is originally triggered by a particular component and then propagated to all related components.*

From now on, we will use *MSR* to denote that it functions as a primitive. An *MSR* is originally triggered by the Mode Switch Triggering Source (MSTS):

Definition 2. *Mode Switch Triggering Source (MSTS) is the component who triggers a mode switch and initiates an* MSR*. An MSTS could be either a primitive or composite component.*

When an MSTS triggers a mode switch, a key issue is how its *MSR* can be propagated to other components. The simplest way is to broadcast its *MSR* to all other components, however, this is against

CBD. When a component is integrated into a component-based system, it shouldn't be aware of the other components in the system. Instead, it only knows its own subcomponents at one level down. Therefore, an *MSR* must be propagated and forwarded from component to component through the hierarchy. To guarantee that all components can be notified by the *MSR*, an MSR propagation mechanism is required.

Our MSR propagation mechanism works differently for primitive and composite components. For a primitive component:

- If it is an MSTS, it will send an *MSR* to its parent and itself as it triggers a mode switch.
- If it is not an MSTS, it does not propagate the *MSR*, but will start its own mode switch upon arrival of an *MSR*.

For a composite component, if it is an MSTS, when it initiates an *MSR*, it needs to send the *MSR* to itself, all its subcomponents and its parent if it has one. If it is not an MSTS, it is sensitive to where the *MSR* comes from:

- If the *MSR* comes from one of its subcomponents, it propagates the *MSR* to all its other subcomponents and to its own parent if it is not at the top level.
- If the *MSR* comes from its parent, then it propagates the *MSR* to all its subcomponents.

As an illustration, suppose that *d* in Figure 1 is the MSTS. Then the MSR propagation process is as follows:

1. Component *d* sends an *MSR* to its parent *b* and to itself.
2. Component *b* propagates the *MSR* to *e* and *f*. It also sends the *MSR* to the top component.
3. The top component sends the *MSR* to *a* and *c*. When both *a* and *c* receive the *MSR*, MSR propagation is terminated.

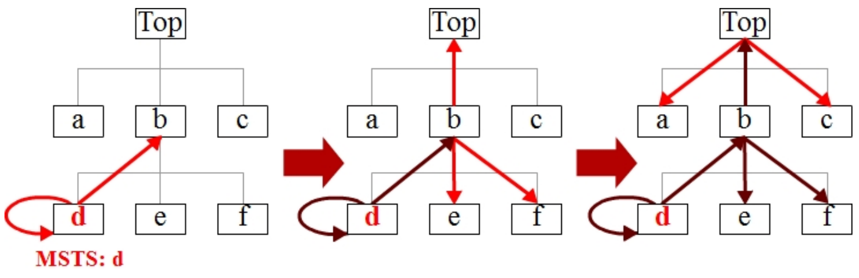

Figure 4: MSR propagation

One problem that we haven't addressed yet is that a system can have multiple MSTSs leading to different mode switch activities, i.e., it is possible that the system receives a second *MSR* during a mode switch. In this report, we assume that the interval between two consecutive *MSR*s is long enough so that the next *MSR* will not be issued until the mode switch of the system triggered by the previous *MSR* is completed.

In order to prove the correctness of our MSR propagation mechanism, a few more concepts need to be explained:

Definition 3. *The Depth Level of a component* x*, i.e.* DL_x*, represents the hierarchical level of a component in a system. The Depth Level of the top component is 0 and the Depth Level is increased by 1 at one level down. The higher Depth Level value, the lower hierarchical level a component is at.*

Definition 4. *If Component* y *is within Component* x *and* $DL_y - DL_x > 1$, *then* x *is the ancestor of* y *and* y *is one descendant of* x. *If* y *is within* x *and* $DL_y - DL_x = 1$, *then* x *is the parent of* y, *and* y *is one child/subcomponent of* x.

The correctness of our MSR propagation mechanism is reflected from two aspects: (1)When an MSTS triggers a mode switch, all components should be notified; (2)When a mode switch is triggered, each component receives one and only one *MSR*. The latter one also indicates its efficiency.

Theorem 1. *Our MSR propagation mechanism guarantees that when a mode switch is triggered, all the components will receive the* MSR *initiated from the MSTS no matter where the MSTS is.*

Proof. Without losing generality, let's define Component x ($x \in CC$ and $DL_x = n$) as the MSTS. First, x can send the *MSR* to itself, thus x can surely receive the *MSR*. When x initiates an *MSR*, we need to prove that any other component y_k with $DL_{y_k} = k$ and $0 \leq k \leq DL_{MAX}$ (DL_{MAX} is the maximal depth level in a system) will receive the *MSR*. With respect to the position of another component in the hierarchy, we distinguish three different cases:

- y_k is one descendant of x. In this case, $n < k \leq DL_{MAX}$. Suppose y_k does not receive the *MSR* from x. Then $\exists y_{k-1}$, such that $y_k \in SC_{y_{k-1}}$ and $DL_{y_{k-1}} = k-1$, and y_{k-1} does not receive the *MSR* (If y_{k-1} has received the *MSR*, then according to our MSR propagation mechanism, it will send the *MSR* to all its subcomponents including y_k). By this structural induction, $\exists y_{n+1}$, such that $y_{n+1} \in SC_x$ and $DL_{y_{n+1}} = n+1$, and y_{n+1} does not receive the *MSR*. This is in contradiction with our MSR propagation mechanism because x must send the *MSR* to all its subcomponents including y_{n+1} at the very beginning of the MSR propagation process. Therefore, y_k will surely receive the *MSR* from x.

- y_k is the parent or one ancestor of x. In this case, $0 \leq k < n$, thus the top component is also included. Suppose y_k does not receive the *MSR* from x. If y_k is the parent of x, it is obviously in contradiction of the MSR propagation mechanism, as x will send the *MSR* to its own parent at the very beginning. If y_k is one ancestor of x, then $\exists y_{k+1}$ which is also one ancestor of x, such that $y_{k+1} \in SC_{y_k}$ and $DL_{y_{k+1}} = k+1$, and y_{k+1} does not receive the *MSR* (According to our MSR propagation mechanism, each ancestor of the MSTS will receive the *MSR* from a subcomponent and must propagate it further to its own parent if there is one. Therefore, if y_{k+1} has received the *MSR*, then according to our MSR propagation mechanism, it will send the *MSR* to y_k upwards). By this structural induction, $\exists y_{n-1}$, such that $x \in SC_{y_{n-1}}$ (i.e. y_{n-1} is the parent of x) and $DL_{y_{n-1}} = n-1$, and y_{n-1} does not receive the *MSR*. This is in contradiction with our MSR propagation mechanism because x must send the *MSR* to its own parent at the very beginning of the MSR propagation process. Therefore, y_k will surely receive the *MSR* from x.

- y_k is neither the descendant nor the ancestor of x. In this case, $0 < k \leq DL_{MAX}$. Suppose y_k does not receive the *MSR* from x. Then $\exists y_{k-1}$, such that $y_k \in SC_{y_{k-1}}$ and $DL_{y_{k-1}} = k-1$, and y_{k-1} does not receive the *MSR*(The same reason as the first case). If y_{k-1} is the parent or one ancestor of x, it is in contradiction with the second case which has been proved (The parent and any ancestor of x can surely receive the *MSR* from x). If y_{k-1} is neither the parent nor one ancestor of x, by this structural induction, $\exists y_m$ such that $DL_{y_m} = m$ and $0 < m < k$, and y_m does not receive the *MSR*. In order not to violate the fact already proved in the second case, y_m still cannot be the parent or any ancestor of x. However, during the induction, as m decreases, y_m will be finally the top component when $m = 0$. And the top component is the ancestor of all components including x. Therefore, y_m will inevitably be either the parent or one ancestor of x. And it will be in contradiction with the fact proved in the second case anyway. Therefore, y_k will surely receive the *MSR* from x.

Since, both the MSTS x and any other component y_k (y_k must belong to one of the three cases above) will surely receive the *MSR* from x no matter where the MSTS is, Theorem 1 is proved. □

Theorem 2. *Our MSR propagation mechanism guarantees that there is no redundant MSR transmission.*

Proof. We have proved that an MSTS guarantees each other component will receive one *MSR*. Now we can assume that one component receives two identical *MSR*s. If we can prove that this is in contradiction with our MSR propagation mechanism, we will know a component cannot receive the same *MSR* twice with regard to the same MSTS. More generally speaking, we will also conclude that a component cannot receive the same *MSR* more than twice because it has to receive two before it can receive more *MSR*s. Then this will imply that each component should receive one and only one *MSR* from the MSTS and Theorem 2 will be proved.

Considering one MSTS, a component rather than the MSTS can receive the same *MSR* twice only due to the following possible reasons:

1. The MSTS sends the same *MSR* twice to the same target(s), which will thus propagate it twice.
2. There is a loop in the component hierarchical structure so that the same *MSR* can be propagated back to a component which has already propagated it before.
3. There is at least one component who propagates the same *MSR* twice to at least one target even after receiving the *MSR* only once.
4. There is at least one component who propagates the *MSR* back to the current sender after receiving the *MSR* from that sender.

Now let's analyze these four reasons above:

- The first reason makes no sense because our MSR propagation mechanism only allows the MSTS to send one *MSR* to the same target, who could be its parent or one of its subcomponents.
- The second reason makes no sense either. The component hierarchy has a typical tree structure as it roots at the top and extends itself to lower levels. No overlapping or loop can be found in the component hierarchy.
- The third reason is actually in contradiction with our MSR propagation mechanism. During MSR propagation, no component will send the same *MSR* twice to another component.
- According to our MSR propagation mechanism, when a component receives the *MSR* from its parent, it will propagate the *MSR* to all its subcomponents but not its parent. When a component receives the *MSR* from one subcomponent, it will propagate the *MSR* to its parent (if it has one) and all its subcomponents rather than the sender. This means that the *MSR* is never propagated back from a receiver to a sender. Therefore, such a component mentioned in the fourth reason never exists.

Among the four possible reasons, the first two makes no sense and the last two are in contradiction with our MSR propagation mechanism. As a result, it is impossible for any component to receive the same *MSR* twice with regard to one MSTS. Then based on the reasoning at the beginning of this proof, Theorem 2 is proved. □

5 Mode switch dependency rules

The MSR propagation mechanism is only applied during MSR propagation, which is just the initial stage of the global mode switch process. After MSR propagation, each component will start its own mode switch. The mode switches of different components must be properly synchronized. Here we make a mode switch dependency rule to achieve this goal:

Rule 1. *Each component starts its reconfiguration after its MSR propagation. There is no dependency on the reconfigurations of different components. A primitive component completes its mode switch after its reconfiguration and then it must send an ms_done signal to its parent. A composite component is supposed to collect ms_done from all its subcomponents. A composite component completes its mode switch when its reconfiguration is completed and it has received ms_done from all its subcomponents. Upon mode switch completion, a composite component will send ms_done to its own parent if it is not at the top level. When the top component completes its mode switch, the global mode switch is completed.*

Based on our mode switch dependency rule, the mode switch of a CBMMS is completed in a bottom-up manner as the mode switch of a composite component is dependent on the mode switches of its sub-components. One advantage of our dependency rule is that it is independent of component connection. What it concerns is only the system component hierarchy.

Figure 5 demonstrates the global mode switch process of the system in Figure 1. Component *d* is the MSTS. The global mode switch starts by MSR propagation which has already been described in the previous section. After receiving the *MSR*, each component will start its reconfiguration, presented by black bars in Figure 5. For primitive components such as *a*, *c*, *d*, *e* and *f*, an *ms_done* signal is sent right after the reconfiguration which equals mode switch completion for them. The top component has a short reconfiguration time, yet it still needs to wait for the *ms_done* signals from its subcomponents *a*, *b* and *c*, thus it is temporarily blocked before its mode switch completion as indicated by the white bar in the figure.

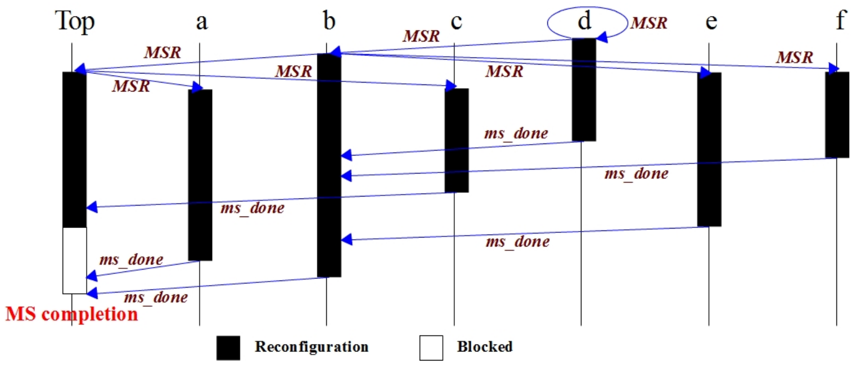

Figure 5: The global mode switch process

The correctness of our mode switch dependency rule can be reflected from the following theorem:

Theorem 3. *Our mode switch dependency rule is deadlock-free.*

Proof. Our mode switch dependency rule is applied after MSR propagation. Since then, there are only two kinds of activities: component reconfiguration and *ms_done* transmission. For a system supporting fully parallel execution, there is no dependency between the reconfiguration of any two components. If no parallel execution is supported, the reconfigurations of different components may be affected by the scheduling policy, but all their reconfigurations can be completed. Thus component reconfiguration will not lead to any deadlock. Besides, since *ms_done* is always sent from a subcomponent to its parent in a bottom-up manner, a component never needs to wait for anything from the parent after its reconfiguration. This will not give rise to any deadlock either. The only one possible deadlock situation is that a composite component is waiting for *ms_done* from its subcomponents and at least one subcomponent will never send *ms_done* to it. Suppose one composite component x is waiting for one *ms_done* from one subcomponent y that never comes. If y is primitive, since the reconfiguration time of any component is bounded, based on our dependency rule, y must send *ms_done* to x as soon as it completes its reconfiguration. This deadlock situation will be in contradiction with the dependency rule. If y is composite, this must imply that y is also waiting for *ms_done* from at least one of its subcomponents. By this structural induction to lower levels, we will unavoidably find that this deadlock is caused by a primitive component z who never sends *ms_done* to its parent after reconfiguration. This is obviously in contradiction with the dependency rule. Therefore, our mode switch dependency rule is deadlock-free. Theorem 3 is proved. $\square$

6 Algorithms for the MSL

In this section, the MSR propagation mechanism and the mode switch dependency rule are combined together as the MSL and they can be implemented in both primitive and components as algorithms. Since the mode switch behaviors of primitive and composite components are different, they will be discussed separately. Algorithm 1 and 2 describe the MSL that control the mode switch processes of primitive and composite components respectively. Before presenting the algorithms, we introduce the following notations:

- m_i denotes the current mode of a component.
- *Wait* and *Signal* are primitives for receiving and sending *MSR* and *ms_done* via the mode switch dedicated ports, i.e. p^{MSR} or p^{MSR}_{in}.
- $MSR(x,y)$ is the *MSR* carrying the new target mode x and the identity of the sending component y.
- $ms\_done(x)$ is the *ms_done* signal carrying the identity of the sender.
- $Reconfiguration(m_{old}, m_{new})$ changes running status and mode-specific behavior for a primitive component. For a composite component, it changes running status and its inner component connections. It may also include some cleaning up in the old mode and preparation for the new mode.
- For a composite component, top is a boolean variable only set to $true$ if it is the top component.
- $exec(MB_x(m_{new}))$ means that x starts to execute its behavior defined in mode m_{new} after mode switch.
- $N^{ms\_done}_x$ denotes the number of *ms_done* signals already received from the subcomponents of a composite component.

7 Current work and future work

Our original MSL assumes all components support the same modes and this unrealistic assumption has been lifted in [6]. Another unrealistic assumption is that the execution of a component is aborted immediately when an *MSR* arrives. However, in a practical system, it is fairly common that one component or even a group of components have atomic execution, which must run to completion before a mode

Algorithm 1 $AlgPC.mode\_switch(x \in PC, m_i \in M_x)$

```
$current\_mode := m_i$;
loop
    $Wait(p^{MSR}, MSR(m_{new}, origin))$;
    $Reconfiguration(current\_mode, m_{new})$;
    $Signal(p^{MSR}, ms\_done(x))$;
    $current\_mode := m_{new}$;
    if $S_x(m_{new}) = Activated$ then
        $exec(MB_x(m_{new}))$;
    end if
end loop
```

Algorithm 2 $AlgCC.mode\_switch(x \in CC, m_i \in M_x, top)$

```
$current\_mode := m_i$;
loop
    $Wait(p^{MSR} \wedge p_{in}^{MSR}, MSR(m_{new}, origin))$;
    if $origin = parent$ then
        $\forall c \in SC_x : Signal(p_{in}^{MSR}, MSR(m_{new}, x))$;
    else
        $\forall c \in (SC_x \setminus \{origin\}) : Signal(p_{in}^{MSR}, MSR(m_{new}, x))$;
        if $\neg top$ then
            $Signal(p^{MSR}, MSR(m_{new}, x))$;
        end if
    end if
    $Reconfiguration(current\_mode, m_{new})$;
    while $N_x^{ms\_done} < |SC_x|$ do
        $Wait$;
    end while
    if $\neg top$ then
        $Signal(p^{MSR}, ms\_done(x))$;
    end if
    $current\_mode := m_{new}$;
end loop
```

switch can be taken. As one of our current works, we are trying to extend the original MSL by adding the support for atomic component execution. Apart from this, we are also looking into the analysis of the global mode switch time while atomic component execution is considered. In addition, we need to come up with a mechanism to resolve the conflict of multiple *MSR*s. The conflict may occur in two conditions: (1) The same MSTS triggers two *MSR*s with so short interval that some other components receive the second *MSR* before mode switch completion; (2) One MSTS x triggers an *MSR* without knowing that another *MSR* has already been triggered by another MSTS y and that *MSR* has not been propagated to x yet, or x and y trigger different mode switches simultaneously. These two conditions may be treated differently. Furthermore, when our MSL is mature enough, it is our ambition to implement it in the ProCom framework [22] that embodies the feature of component reuse very well.

8 Acknowledgments

This work is supported by the Swedish Research Council.

References

[1] B. Andersson. Uniprocessor edf scheduling with mode change. In *12th International Conference on Principles of Distributed Systems*, pages 572–577, 2008.

[2] E. Borde, G. Haïk, and L. Pautet. Mode-based reconfiguration of critical software component architectures. In *Conference on Design, Automation and Test in Europe*, pages 1160–1165, 2009.

[3] Ivica Crnkovic and Magnus Larsson. *Building reliable component-based software systems*. Artech House, 2002.

[4] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The architecture analysis & design language (AADL): An introduction. Technical Report CMU/SEI-2006-TN-011, Software engineering institute, MA, February 2006.

[5] 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.

[6] 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.

[7] 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.

[8] T. A. Henzinger, B. Horowitz, and C. Meyer Kirsch. Giotto: A time-triggered language for embedded programming. In *PROCEEDINGS OF THE IEEE*, pages 166–184, 2001.

[9] X. Ke, K. Sierszecki, and C. Angelov. COMDES-II: A component-based framework for generative development of distributed real-time control systems. In *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2007.

[10] V. Nélis, B. Andersson, J. Marinho, and S. M. Petters. Global-edf scheduling of multimode real-time systems considering mode independent tasks. In *23rd Euromicro Conference on Real-Time Systems*, pages 205–214, 2011.

[11] V. Nélis, J. Goossens, and B. Andersson. Two protocols for scheduling multi-mode real-time systems upon identical multiprocessor platforms. In *21st Euromicro Conference on Real-Time Systems*, pages 151–160, 2009.

[12] Paulo Pedro and Alan Burns. Schedulability analysis for mode changes in flexible real-time systems. In *10th Euromicro Conference on Real-Time Systems*, pages 172–179, 1998.

[13] L. T. X. Phan, I. Lee, and O. Sokolsky. A semantic framework for mode change protocols. In *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 91–100, 2011.

[14] Linh T. X. Phan, Samarjit Chakraborty, and P S Thiagarajan. A multi-mode real-time calculus. In *Real-Time Systems Symposium*, pages 59–69, 2008.

[15] Linh T. X. Phan, Insup Lee, and Oleg Sokolsky. Compositional analysis of multi-mode systems. In *22nd Euromicro Conference on Real-Time Systems*, pages 197–206, 2010.

[16] J. Real and A. Crespo. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems*, 26(2):161–197, 2004.

[17] P. D. Stefan Resmerita and W. Pree. Timing definition language (TDL) modeling in ptolemy II. Technical report, Department of Computer Science, University of Salzburg, June 2008.

[18] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority-driven preemptive scheduling. *Real-Time Systems*, 1:243–264, 1989.

[19] N. Stoimenov, S. Perathoner, and L. Thiele. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *Conference on Design, Automation and Test in Europe*, pages 99–104, 2009.

[20] Josef Templ. TDL specification and report. Technical report, Department of Computer Science, University of Salzburg, November 2003.

[21] K. W. Tindell, A. Burns, and A. J. Wellings. Mode changes in priority pre-emptively scheduled systems. In *Real Time Systems Symposium*, pages 100–109, 1992.

[22] A. Vulgarakis, J. Suryadevara, J. Carlson, C. Seceleanu, and P. Pettersson. Formal semantics of the ProCom real-time component model. In *35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 478–485, 2009.

[23] P. Meumeu Yomsi, V. Nelis, and J. Goossens. Scheduling multi-mode real-time systems upon uniform multiprocessor platforms. In *15th IEEE International Conference on Emerging Technologies and Factory Automation*, 2010.