Handling multiple mode switch scenarios 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*—The growing complexity of embedded systems software entails new development techniques. Component-Based Software Engineering is undoubtedly suitable for the development of complex systems thanks to its inherent component reuse. Another approach to reduce software complexity is by partitioning the system behavior into different operational modes. Each mode is associated with a unique behavior and the system can change behavior by switching between modes. When such a multi-mode system is developed by reusable software components, a crucial issue is how to achieve a seamless composition of multi-mode components and also how to handle mode switch properly. As an integrated solution to the challenges of multi-mode component-based software system development we have proposed the Mode Switch Logic (MSL). The current version of MSL assumes independent handling of a single mode switch scenario, i.e. that no other mode switch is triggered until an ongoing mode switch is completed. For a wide class of systems, this is an unrealistic assumption. In this paper we lift this assumption by proposing an extension of MSL to handle multiple mode switch scenarios concurrently triggered by different components.**

***Keywords*-component-based; mode switch; multi-mode**

I. Introduction

The software complexity of embedded systems is growing rapidly, imposing challenges on traditional development techniques. As a consequence, there is a strong demand for new techniques to reduce software complexity. Among these new techniques, Component-Based Software Engineering (CBSE) [1] provides a promising paradigm for the development of complex systems, as it allows a system to be built by reusable components that can be independently developed. The success of CBSE has been evidenced by a number of component models proposed in both industry and academia [2] [3].

Another approach to reduce software complexity is to partition the system behavior into different operational modes. Such a multi-mode system usually runs in one of its supported modes and can switch to another mode under certain circumstances. A representative example is the control software of an airplane, which could run in the modes *taxi* (the initial mode), *taking off*, *flight* and *landing*. Different subsystems are running in different modes. For instance, the navigation subsystem may only run in *flight* mode and the subsystem for controlling the wheels may only run in *taxi* mode.

Combining CBSE and multi-mode systems, we get a Component-Based Multi-Mode System (CBMMS), i.e. a multi-mode system developed in a component-based manner. Fig. 1 illustrates a conceptual CBMMS, with its component hierarchy on the left and its component connections on the right. The system, i.e. Component a, consists of three components: b, c and d. Component c is composed by e and f. According to the terminology of CBSE, we distinguish *primitive components* and *composite components*. A primitive component is directly implemented by code while a composite component is composed by other components. In Fig. 1, b, d, e and f are primitive components while a and c are composite components. Since the component hierarchy has a tree structure, a composite component and its subcomponents have a parent-and-children relationship. For instance, c is the parent of e and f, which in turn are the children of c. Moreover, the system can run in two modes: m_a^1 and m_a^2. When the system is in m_a^1, Component d is deactivated (i.e. not running), shown in the component hierarchy in Fig. 1 by not displaying d in mode m_a^1. In contrast, when the system is in m_a^2, d is activated while f is deactivated. Besides, Component b has different mode-specific behaviors represented by black and grey colors in Fig. 1.

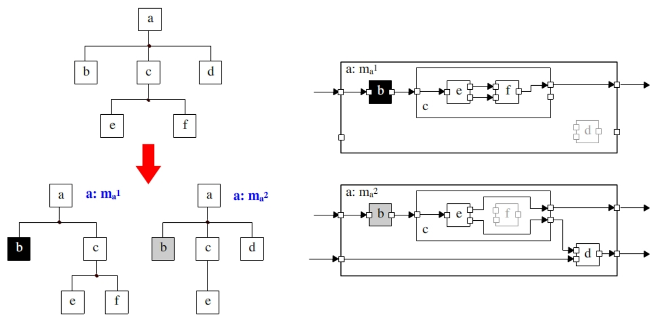

Figure 1. A component-based multi-mode system

The predominant challenge for a CBMMS is its mode switch handling. Fig. 1 implies that the mode switch of a system may amount to the joint mode switches of many different components at various levels. For instance, the mode switch from m_a^1 to m_a^2 requires the activation of d, the deactivation of f, and the change of behavior of b. In order to overcome this challenge, we have developed the Mode Switch Logic (MSL) [4] [5], a systematic approach to the development of CBMMSs and its mode switch handling. In MSL, some components can detect a mode switch event and trigger a mode switch scenario that is propagated to some other components which may also switch mode as a consequence.

Currently, MSL is limited by assuming that only a single mode switch scenario is handled at a time. However, in a real system, it could be possible that multiple mode switch scenarios are triggered simultaneously, thus giving rise to a conflict situation. A vivid example of concurrent triggering of multiple mode switch scenarios is road trains [6] which provide the opportunity for a vehicle to join or leave a platoon of other vehicles led by a professional driver in the front vehicle on a motorway. To be a member of a platoon, a vehicle should approach the platoon and send a request. If the request is approved by the leading driver, the vehicle will become part of the platoon and enter a semi-autonomous control mode in which the vehicle is able to automatically follow the platoon so that the driver can relax himself by doing something more interesting other than maneuvering the car. When a driver wants to leave the platoon, he can simply send a leaving request to inform the leading driver and drive out of the platoon as the car switches to manual mode. Now consider two concurrent events: one is triggered by a driver who wants to leave the platoon while the other is triggered as the leading driver brakes abruptly due to the unexpected obstruction of a wild animal in front. An emergency brake must be immediately performed for all members of the platoon before any vehicle can switch to manual mode and leave the platoon. Similarly, in a CBMMS, different components may concurrently request to switch modes. The contribution of this paper is that it proposes an extension of MSL that can handle such multiple concurrent mode switch scenarios.

The remainder of the paper is organized as follows: Section II briefly introduces MSL. In Section III, a conflict handling mechanism is proposed for MSL to manage concurrent triggering of multiple mode switch scenarios. The correctness-verification of the conflict handling mechanism is presented in Section IV. Related work is reviewed in Section V. Section VI concludes the paper and discusses future work.

II. The Mode Switch Logic

The Mode Switch Logic (MSL) includes three major elements: (1) a mode-aware component model; (2) a mode mapping mechanism; and (3) a mode switch runtime mechanism. The focus of this paper is on the mode switch runtime mechanism which is extended to cope with multiple mode switch scenarios.

The mode-aware component model defines essential features for a component to be mode-aware. A component can support multiple modes, each of which represents a unique configuration. The mode switch of an individual component is realized by reconfiguration, viz. changing its configuration in the current mode to the configuration in the new mode. Furthermore, dedicated mode switch ports are introduced for the exchange of mode-related information with the parent and subcomponents of a component.

Since multi-mode components are independently developed for reuse, we do not assume a system-wide agreement on the naming and number of modes of each component. The mode mapping mechanism describes the correlation between the modes of different components. Given the current mode of a component at runtime, mode mapping can tell the current modes of other components. In addition, it also determines the new modes of different components when a mode switch is taking place. More details of the mode-aware component model and the mode mapping mechanism can be found in [4].

The mode switch runtime mechanism coordinates the mode switches of different components at runtime. It defines the following roles:

- The Mode Switch Source (MSS): a (primitive or composite) component which can detect a mode switch event (e.g. that the value of a sensor reaches a threshold) and actively request to switch mode by triggering a mode switch scenario. We use $c_k : m^i_{c_k} \to m^j_{c_k}$ to denote a mode switch scenario in which an MSS c_k requests to switch mode from mode $m^i_{c_k}$ to $m^j_{c_k}$. We shall hereafter use "scenario" when referring to such a "mode switch scenario".
- The Mode Switch Decision Maker (MSDM): a component which has the authority to approve or reject a scenario. This component is scenario-dependent and must be either directly or indirectly composed by the MSS that triggers the scenario. A mode switch is performed only when the MSDM approves a scenario.
- Type A/B components: For a given scenario, a Type A component must switch mode as a consequence, while a Type B component is unaffected. Type A and Type B components are determined by mode mapping and are scenario-dependent. In this paper, we use $T_{c_i} = A$ or $T_{c_i} = B$ to denote that c_i is a Type A or Type B component, respectively.

The mode switch runtime mechanism consists of a Mode Switch Propagation (MSP) protocol and a mode switch dependency rule. The MSP protocol propagates a scenario to all Type A components without affecting Type B components. The MSDM is also identified by the MSP protocol for a specific scenario. If the MSDM approves a scenario by triggering a mode switch, the mode switch dependency rule guarantees the mode consistency between different components upon each mode switch completion.

For a component c_i, let S_{c_i} denote that the current state of c_i allows a mode switch, and let $\neg S_{c_i}$ denote that the current state of c_i does not allow a mode switch. Also, let P_{c_i} be the parent of c_i and *Top* be the top component in the component hierarchy. Now consider a scenario triggered by an MSS c_i, with c_j as the MSDM and C_M as the set of vertically intermediate components between c_i and c_j in the component hierarchy. We extend the MSP protocol presented in [4] into the following:

Definition 1. ***The Mode Switch Propagation (MSP) protocol:*** *When c_i detects a mode switch event, it will request to switch mode by triggering a scenario. If $c_i \neq Top$, c_i will issue an **MSR** (Mode Switch Request) primitive which is sent to P_{c_i}, eventually reaching c_j through C_M. For each $c_k \in C_M$, identified when $T_{c_k} = A$ and S_{c_k} upon receiving the **MSR**, c_k forwards the **MSR** to P_{c_k}. Upon receiving the **MSR**, the MSDM c_j is identified if one of the three following conditions is satisfied: (1) $T_{c_j} = B$; (2) $T_{c_j} = A$ and $\neg S_{c_j}$; (3) $T_{c_j} = A$ and S_{c_j} and $c_j = Top$. The MSDM c_j makes the following decisions:*

- *In Condition (2), c_j will reject the **MSR** by issuing an **MSD** (Mode Switch Denial) primitive that is propagated back to c_i via C_M. Mode switch propagation is terminated when c_i receives the **MSD**. No component will switch mode for this scenario.*
- *In conditions (1) and (3), c_j will approve the **MSR** by issuing an **MSQ** (Mode Switch Query) primitive that is propagated downstream and stepwise to all Type A components. After receiving an **MSQ**, a component c_l is required to reply to P_{c_l} with either an **MSOK** or **MSNOK** primitive. Component c_l replies with an **MSOK** if S_{c_l} (and if all its Type A subcomponents have replied with an **MSOK** if c_l is composite). Otherwise, if $\neg S_{c_l}$, c_l will directly reply to P_{c_l} with an **MSNOK** (without propagating the **MSQ** downstream further if c_l is composite). If c_l receives at least one **MSNOK** from a subcomponent after its **MSQ** propagation, it will also reply to P_{c_l} with an **MSNOK**.*
- *If all the Type A subcomponents of c_j have replied with an **MSOK**, c_j will trigger a mode switch by issuing an **MSI** (Mode Switch Instruction) primitive that follows the propagation trace of the **MSQ**. Mode switch propagation is completed when all Type A components have received the **MSI**. In contrast, if c_j receives at least one **MSNOK**, it will abort the mode switch plan by issuing an **MSD** that follows the propagation trace of the **MSQ**. Mode switch propagation is terminated without any component switching mode when the **MSD** reaches all components that have received the **MSQ**.*

*If $c_i = Top$, then $c_j = c_i$ and $C_M = \varnothing$. Then, when c_i detects a mode switch event, it will directly issue an **MSQ** to its Type A subcomponents.*

The difference between the MSP protocol in [4] and the extended MSP protocol above is only the way of directly rejecting an **MSR**. In the old version, an MSDM does nothing when it directly rejects an **MSR**, whereas in the extended version here, an **MSD** must be sent back from the MSDM all the way down to the MSS. This extension serves as an initial preparation for the handling of multiple scenarios.

When an MSDM triggers a mode switch by issuing an **MSI**, all Type A components will switch mode after its **MSI** propagation, following the mode switch dependency rule, which specifies the conditions of mode switch completion:

Definition 2. ***The mode switch dependency rule:*** *Let c_j be the MSDM for a scenario and c_j triggers a mode switch by issuing an **MSI** that is propagated downstream and stepwise to all Type A components. Then,*

- *For any primitive component c_i ($T_{c_i} = A$), c_i starts its mode switch by reconfiguring itself upon receiving an **MSI**. The mode switch completion of c_i equals its reconfiguration completion. An **MSC** (Mode Switch Completion) primitive will be sent from c_i to P_{c_i} when c_i completes its mode switch.*
- *For any composite component c_i ($T_{c_i} = A$), c_i starts its mode switch by reconfiguring itself after its **MSI** propagation. Component c_i completes its mode switch when it completes its reconfiguration and has received an **MSC** from all its Type A subcomponents. After that, if $c_i \neq c_j$, an **MSC** will be sent from c_i to P_{c_i} after c_i completes its mode switch.*
- *If $T_{c_j} = A$, the system mode switch is completed after the mode switch of c_j. Otherwise, if $T_{c_j} = B$, the system mode switch is completed after c_j has received an **MSC** from all its Type A subcomponents.*

The mode switch dependency rule guarantees that all Type A components must be running in their new modes after the mode switch completion of a system, which is a key property ensuring mode consistency.

The mode switch runtime mechanism, chiefly represented by the MSP protocol and the mode switch dependency rule, is demonstrated by the time sequence diagram in Fig. 2, showing a complete mode switch process based on the example in Fig. 1. In Fig. 2, the MSS *c* triggers a scenario by issuing an **MSR**. The **MSR** from *c* is sent to its parent *a*, which is the MSDM of this scenario. Component *a* approves the **MSR** by issuing an **MSQ** that is propagated to all Type A components. It should be noted that *e* is a Type B component, thus not affected by this scenario. After receiving the **MSQ**, each Type A component checks its current state. Here the current state of each Type A component allows a mode switch, therefore an **MSOK** is sent back in response to the **MSQ**. Once the MSDM *a* receives the **MSOK** from its Type A subcomponents *b*, *c* and *d*, it will trigger a mode switch by issuing an **MSI** that follows the propagation trace of the **MSQ**. The propagation of **MSI** results in the reconfiguration of each Type A component, represented by black bars in Fig. 2. Finally, **MSC** primitives are sent bottom-up to indicate mode switch completion. The white bars mean that the mode switch of a composite component is blocked by its subcomponents, i.e. a composite component has completed its reconfiguration but is still waiting for at least one **MSC**. Since the MSDM a is a Type A component, the system mode switch is completed upon the mode switch completion of a.

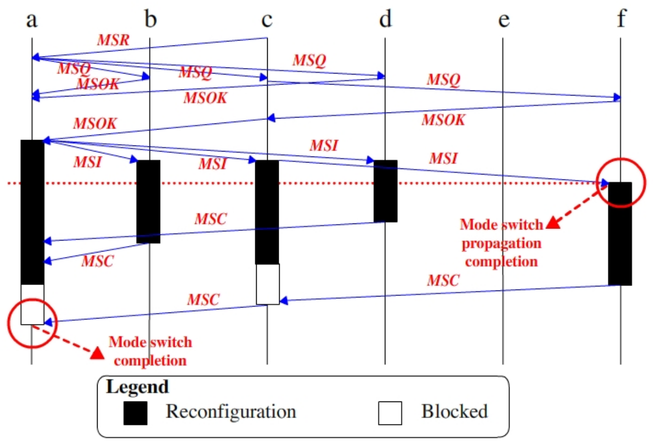

Figure 2. A complete mode switch process

III. The handling of multiple mode switch scenarios

The mode switch runtime mechanism presented in Section II assumes that no new scenarios are triggered until the current scenario is completely handled. However, it is likely that a system has multiple MSSs potentially triggering concurrent scenarios. In this section, we propose a conflict handling mechanism to cope with multiple scenarios based on the following assumptions:

1) A mode switch cannot be aborted or rolled back.
2) An MSS will not trigger a new scenario until the current scenario triggered by it is completely handled.
3) There is no mode switch failure, i.e. all mode switch primitives are correctly communicated and the handling is correctly executed.

Essentially, the conflict handling mechanism introduces separate queues for storing incoming **MSR** and **MSQ** primitives and updates these queues based on a set of pre-defined criteria. In order to preserve component encapsulation, these queues are locally defined for each component instead of being globally defined for the entire system.

A. MSR and MSQ queues

When multiple scenarios are considered, each component must be able to distinguish different scenarios. Hence, we introduce unique identifiers:

Definition 3. *A mode switch scenario ID is a unique ID of a scenario. Any **MSX** primitive (e.g. **MSR** or **MSQ**) must carry a (mode switch) scenario ID k that it is associated with, denoted as msx^k.*

For a system where concurrent scenario triggering is allowed, each component may receive multiple primitives simultaneously or within a short interval, each primitive being associated with a specific scenario. Since a component can only handle a single primitive each time, other primitives must be buffered somewhere to be handled afterwards. Therefore, we introduce MSR and MSQ queues. For a CBMMS, let PC be the set of its primitive components and let CC be the set of its composite components. We also use $\widetilde{CC}$ to denote the set of composite components excluding *Top*. Let SC_{c_i} be the set of subcomponents of a composite component c_i. Furthermore, let $T^k_{c_i} = A$ or $T^k_{c_i} = B$ denote that c_i is a Type A or Type B component for Scenario k and let $SC^A_{c_i}(k)$ denote the set of Type A subcomponents of c_i for k, then:

Definition 4. *An MSR queue of c_i, denoted as $c_i.Q_{msr}$, is a FIFO queue storing any incoming **MSR** from SC_{c_i} (and from c_i itself if c_i is an MSS and $c_i \neq Top$). An **MSR** in this queue is denoted as msr^k, or $msr^k_{c_j}$ where k is the scenario ID and $c_j \in SC_{c_i} \cup \{c_i\}$ is the immediate sender of this **MSR**.*

Whenever c_i receives an **MSR** from a subcomponent or decides to trigger a scenario by issuing an **MSR** ($c_i \neq Top$) as an MSS, the **MSR** will be enqueued in $c_i.Q_{msr}$. Conversely, an **MSR** $msr^k_{c_j}$ is dequeued from $c_i.Q_{msr}$ when any one of the following conditions is satisfied:

1) c_i completes its mode switch based on Scenario k ($T^k_{c_i} = A$).
2) c_i receives a msc^k from all $c_j \in SC^A_{c_i}(k)$ ($T^k_{c_i} = B$).
3) c_i receives a msd^k ($c_i \in PC \vee SC^A_{c_i}(k) = \varnothing \vee c_i \Rightarrow k$).
4) c_i has propagated a msd^k ($SC^A_{c_i}(k) \neq \varnothing$).

Here $c_i \Rightarrow k$ means that c_i is the MSS of k. Conditions (3) and (4) imply two cases. First, if $\exists msr^k \in c_i.Q_{msr}$ that is directly rejected by the MSDM, then c_i should know where k comes from as it receives a msd^k. If c_i is the MSS of k, no further propagation will be needed for the msd^k. If k is from $c_j \in SC_{c_i}$, then c_i should propagate the msd^k to c_j. Second, if $\exists msr^k \in c_i.Q_{msr}$ that is accepted by the MSDM and c_i has propagated a msq^k to $SC^A_{c_i}(k)$, then c_i should also propagate a msd^k to $SC^A_{c_i}(k)$ unless $SC^A_{c_i}(k) = \varnothing$.

In addition to the MSR queue, MSQ queues are defined in a similar fashion:

Definition 5. *An MSQ queue of c_i, denoted as $c_i.Q_{msq}$, is a FIFO queue storing an incoming **MSQ** from P_{c_i} (or from c_i itself if c_i is an MSS and $c_i = Top$). An **MSQ** in this queue is denoted as msq^k where k is the scenario ID.*

Whenever c_i receives an **MSQ** from P_{c_i} or decides to trigger a scenario by issuing an **MSQ** ($c_i = Top$), the **MSQ** is enqueued in $c_i.Q_{msq}$. The dequeue conditions of an msq^k

in $c_i.Q_{msq}$ are exactly the same as those of the $msr^k_{c_j}$ in $c_i.Q_{msr}$.

Note that MSR and MSQ queues are not needed for all components. For instance, since a primitive component has no subcomponents, it only needs an MSR queue with size 1 if it can be an MSS in some mode(s). Likewise, since *Top* has no parent, it only needs an MSQ queue with size 1 if it can be an MSS in some mode(s). Table I shows the allocation of MSR and MSQ queues to different types of components together with the maximum required size of each queue. The queue sizes in Table I guarantee that no overflow occurs to the MSR and MSQ queues of any component. The reason for such an allocation and queue size is explained in [7].

Table I
THE ALLOCATION OF MSR AND MSQ QUEUES

c_i	MSR queue	MSQ queue
(1) $c_i \in PC, c_i \neq MSS$	No	Yes (Size: 2)
(2) $c_i \in PC, c_i = MSS$	Yes (Size: 1)	Yes (Size: 2)
(3) $c_i \in \widetilde{CC}, c_i \neq MSS$	Yes (Size: $2 * \lvert SC_{c_i} \rvert$)	Yes (Size: 2)
(4) $c_i \in \widetilde{CC}, c_i = MSS$	Yes (Size: $2 * \lvert SC_{c_i} \rvert + 1$)	Yes (Size: 2)
(5) $c_i = Top, c_i \neq MSS$	Yes (Size: $2 * \lvert SC_{c_i} \rvert$)	No
(6) $c_i = Top, c_i = MSS$	Yes (Size: $2 * \lvert SC_{c_i} \rvert$)	Yes (Size: 1)

Based on the definition of MSR/MSQ queues together with their enqueuing and dequeuing conditions, we propose the *MSR/MSQ queue checking rule* and the *MSR/MSQ queue updating rule*, which jointly constitute our conflict handling mechanism.

B. The MSR/MSQ queue checking rule

The MSR/MSQ queue checking rule includes two additional prerequisite terms: *transition state* and *locked* **MSR**:

Definition 6. *A component c_i is in a transition state within the interval $[t_1, t_2]$ for Scenario k, where*

- *If $c_i \in PC$, t_1 is the time when c_i handles a $msq^k \in c_i.Q_{msq}$, while t_2 is the time when the dequeuing conditions (1) or (3) are satisfied for either $c_i.Q_{msr}$ or $c_i.Q_{msq}$ for k.*
- *If $c_i \in \widetilde{CC}$, t_1 is the time when (1) c_i issues a msq^k to $SC^A_{c_i}(k)$ as an MSDM; or (2) c_i handles a $msq^k \in c_i.Q_{msq}$. In addition, t_2 is the time when one of the dequeuing conditions (1)-(4) is satisfied for either $c_i.Q_{msr}$ or $c_i.Q_{msq}$ for k.*
- *If $c_i = Top$, t_1 is the time when c_i issues a msq^k to $SC^A_{c_i}(k)$, while t_2 is the time when the dequeuing conditions (1), (2), or (4) are satisfied for either $c_i.Q_{msr}$ or $c_i.Q_{msq}$ for k.*

Definition 7. *The first **MSR** in $c_i.Q_{msr}$ is locked if it has been forwarded by c_i to P_{c_i} according to the MSP protocol.*

Based on the definitions above, the MSR/MSQ queue checking rule is defined as follows:

Definition 8. ***The MSR/MSQ queue checking rule:*** *If c_i is not in transition state, then: If $c_i.Q_{msq} \neq \varnothing$, the first **MSQ** in $c_i.Q_{msq}$ will be immediately handled, else if $c_i.Q_{msr} \neq \varnothing$ and the first **MSR** in $c_i.Q_{msr}$ is not locked, the first **MSR** will be immediately handled. The handling of the **MSQ** or **MSR** follows the mode switch runtime mechanism.*

The MSR/MSQ queue checking rule enables a component to handle multiple scenarios sequentially and enables a system to handle multiple scenarios concurrently by its components. When a component is in transition state for Scenario k, it is dedicated to the handling of k until it leaves the transition state, i.e. when it has completely handled k. By this means, its handling of k can never be interfered by the arrival of another Scenario k' which is simply enqueued and handled afterwards. The MSR/MSQ queue checking rule can be implemented as algorithms 1-3, where the following notations deserve extra explanation:

- *Wait*(c_i, A, B) and C : *Signal*(c_i, A, B) are used for c_i to receive the primitive B or send the primitive B to C via the dedicated mode switch port A, which is either p^{MSX} (for exchanging primitives with P_{c_i}) or p^{MSX}_{in} (for exchanging primitives with SC_{c_i}).
- *MSC_Collecting* and $valid^k$ are boolean variables which will be explained in the next section.
- $k \leftarrow c_j$: Scenario k is from the immediate sender c_j.
- $locked$ is a boolean variable of c_i set to true if the first **MSR** in $c_i.Q_{msr}$ is locked.
- $enqueue(A, B)$ is a function enqueuing the primitive A (either **MSR** or **MSQ**) in queue B.
- $updateBothQueues(c_i, k)$: If the first **MSQ** in $c_i.Q_{msq}$ is msq^k, then it is removed from $c_i.Q_{msq}$; if the first **MSR** in $c_i.Q_{msr}$ is msr^k, then it is removed from $c_i.Q_{msr}$. Besides, after c_i removes msr^k, if $locked$ is true, c_i will set it to false.
- *MS_event_detected* is a boolean variable set to true when the MSS c_i detects a mode switch event.
- *Derive_new_mode*(c_i) is a function deriving the new mode of an MSS c_i as c_i detects a mode switch event.
- $TransitionS$ is a boolean variable of c_i set to true when c_i is in transition state.
- *HandleMSQ*(c_i) and *HandleMSR*(c_i) are functions for the handling of an **MSQ** and **MSR**, respectively, following the mode switch runtime mechanism presented in Section II.

Algorithm 1 detects any incoming **MSR** or **MSQ** which is then put in the right queue. Furthermore, if c_i has sent a msr^k to P_{c_i} and msr^k is directly rejected by the MSDM, c_i will receive a msd^k from P_{c_i}. Then c_i should dequeue the locked msr^k from $c_i.Q_{msr}$ (and further propagate the msd^k to $c_j \in SC_{c_i}$ if $c_i \in CC$ and msr^k comes from c_j).

In addition to Algorithm 1, if c_i is an MSS in at least one of its modes, Algorithm 2 is also applied to enqueue self-triggered **MSR** or **MSQ** of c_i.

Algorithm 1 *MSR_MSQ_enqueue*(c_i)

```
loop
  $Wait(c_i, p^{MSX} \vee p_{in}^{MSX}, primitive)$;
  if $primitive = msr_{c_j}^{k}$ && $c_j \in SC_{c_i}$ then
    if $c_i \in CC$ && MSC_Collecting then
      $valid^k := true$;
    end if
    $enqueue(msr_{c_j}^{k}, c_i.Q_{msr})$;
  else if $primitive = msq^k$ then
    $enqueue(msq^k, c_i.Q_{msq})$;
  else{$primitive = msd^k$}
    if $c_i \in CC$ && $k \leftarrow c_j \in SC_{c_i}$ then
      $c_j : Signal(c_i, p_{in}^{MSX}, msd^k)$;
    end if
    $updateBothQueues(c_i, k)$;
  end if
end loop
```

Algorithm 2 *MS_detection*(c_i)

```
loop
  if MS_event_detected then
    Derive_new_mode$(c_i)$;
    if $c_i \neq Top$ then
      $enqueue(msr_{c_i}^{k}, c_i.Q_{msr})$;
    else{$c_i = Top$}
      $enqueue(msq^k, c_i.Q_{msq})$;
    end if
  end if
end loop
```

Algorithm 3 implements the MSR/MSQ queue checking rule. The MSR and MSQ queues of c_i are checked only when c_i is not in any transition state. It calls functions *HandleMSQ*(c_i) and *HandleMSR*(c_i). Due to limited space, the algorithms for these two functions (available in [7]) will not be presented here.

Algorithm 3 *CheckQueue*(c_i)

```
loop
  if $\neg TransitionS$ then
    if $c_i.Q_{msq} \neq \varnothing$ then
      HandleMSQ$(c_i)$;
    else
      if $c_i.Q_{msr} \neq \varnothing$ && $\neg locked$ then
        HandleMSR$(c_i)$;
      end if
    end if
  end if
end loop
```

C. The MSR/MSQ queue updating rule

The MSR/MSQ queue checking rule alone is still insufficient to handle multiple scenarios correctly, as it is unaware of the impact of a scenario upon other pending scenarios, implicitly assuming that different scenarios are independent of each other. Nevertheless, after a component completes its mode switch for Scenario k, if $\exists msr_{c_j}^{k'}$ in $c_i.Q_{msr}$ ($c_j \in SC_{c_i} \cup \{c_i\}$), then $msr_{c_j}^{k'}$ may not be valid any more. This problem can be illustrated by a small example. Let's consider a system with its component hierarchy presented in Fig. 1. Tables II and III give the basic mode mappings of the two composite components *a* and *c*. In each table, modes in the same column are mapped. For example, according to Table II, when *a* is running in mode m_a^1, *b* must be running in m_b^1, *c* can run in either m_c^1 or m_c^2, and *d* is deactivated. Fig. 3 depicts three scenarios: (1) $S_1 = (b : m_b^1 \rightarrow m_b^2)$; (2) $S_2 = (e : m_e^1 \rightarrow m_e^2)$; (3) $S_3 = (f : m_f^1 \rightarrow m_f^2)$. For S_1 and S_3, all components are Type A components, while for S_2, only *c* and *e* are Type A components. For each scenario, the current possible mode and the target mode of each Type A component are also defined in Fig. 3. For instance, for S_1, $m_f^1 \rightarrow m_f^2$ means that S_1 will imply the mode switch of *f* from m_f^1 to m_f^2. The given mode mappings imply that when *b* is running in m_b^1, *e* can be in either m_e^1 or m_e^2 while *f* must be running in m_f^1. Therefore, all three scenarios can be simultaneously triggered.

Table II
THE MODE MAPPING TABLE OF *a*

Component	Supported modes		
a	m_a^1		m_a^2
b	m_b^1		m_b^2
c	m_c^1	m_c^2	m_c^3
d	Deactivated		m_d^1

Table III
THE MODE MAPPING TABLE OF *c*

Component	Supported modes		
c	m_c^1	m_c^2	m_c^3
e	m_e^1	m_e^2	Deactivated
f	m_f^1		m_f^2

Suppose S_1 and S_3 are triggered at the same time. Then *b* and *f* will issue two different **MSR** primitives (say msr^{S_1} and msr^{S_3}) simultaneously. The MSP protocol indicates that *a* is the MSDM for both scenarios. After some time, a possible outcome is that msr^{S_1} arrives at $a.Q_{msr}$ earlier than msr^{S_3}. Applying the MSR/MSQ queue checking rule, *a* will first handle msr^{S_1}. Suppose a system mode switch is successfully performed based on S_1. Upon mode switch completion, msr^{S_1} is dequeued from $a.Q_{msr}$ and *a* is supposed to handle msr^{S_3}. However, all components are Type A components for S_1, including the MSS of S_3, *f*, whose current mode has become m_f^2 rather than m_f^1. As a consequence, S_3 is no longer valid because it can only be triggered when *f* is in m_f^1. A reasonable action regarding such an invalid scenario would be to remove all the pending

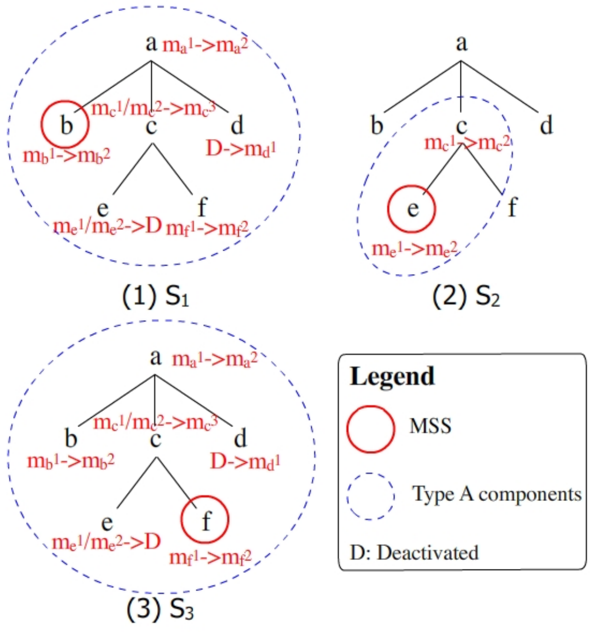

Figure 3. Mode switch scenarios

MSR primitives associated with S_3, including $msr_c^{S_3}$ in $a.Q_{msr}$, $msr_f^{S_3}$ in $c.Q_{msr}$, and $msr_f^{S_3}$ in $f.Q_{msr}$.

Sometimes a pending scenario may still remain valid in spite of the mode switch completion of another scenario. Suppose S_2 and S_3 are simultaneously triggered. A possible outcome is that msr^{S_2} arrives at $c.Q_{msr}$ earlier than msr^{S_3}. Evidently, c will first handle msr^{S_2}. If a mode switch is performed and completed based on S_2, the pending Scenario S_3 is still valid because f (the MSS of S_3) is a Type B component for S_2 which does not affect f. In this case, the pending **MSR** primitives $msr_f^{S_3}$ in both $c.Q_{msr}$ and $f.Q_{msr}$ should not be removed without handling.

Unfortunately, after a mode switch, it is impossible for each component to tell if a pending scenario is valid or not. The reason is that only an MSS itself knows that it is the MSS of a scenario (in order not to break component encapsulation). However, it is viable for each component to tell whether a pending **MSR/MSQ** in its MSR/MSQ queue is valid or not. A valid **MSR/MSQ** is specified as follows:

Definition 9. *Let c_i be a component which has completed its mode switch based on Scenario k ($T_{c_i}^k = A$) or has received an* **MSC** *from all $c_j \in SC_{c_i}^A(k)$ ($c_i \in CC, T_{c_i}^k = B$). If $c_i.Q_{msr} \neq \emptyset$, then c_i will identify the validity of each $msr^{k'} \in c_i.Q_{msr}$ ($k' \neq k$):*

- *If $c_i \in PC$, then c_i must be the MSS of k'. Hence $msr^{k'}$ is invalid.*
- *If $c_i \in CC$, then $msr^{k'}$ can come from either c_i or $c_j \in SC_{c_i}$. If $msr^{k'}$ is from c_i itself, then $msr^{k'}$ is valid when $T_{c_i}^k = B$ and invalid when $T_{c_i}^k = A$. If $msr^{k'}$ is from $c_j \in SC_{c_i}$, then $msr^{k'}$ is valid under two conditions: (1) $T_{c_j}^k = B$; (2) $T_{c_j}^k = A$ and c_j sends $msr^{k'}$ to c_i after sending msc^k to c_i while c_i is waiting for msc^k from $SC_{c_i}^A(k)$. Otherwise, $msr^{k'}$ is invalid.*

If c_i = Top, $T_{c_i}^k = A$, and $\exists msq^{k'} \in c_i.Q_{msq}$, then $msq^{k'}$ is invalid. Otherwise, $msq^{k'}$ is always valid.

Definition 9 explains *MSC_Collecting* and $valid^k$ in Algorithm 1. When *MSC_Collecting* is true for a composite component c_i which is in transition state for k, c_i must be waiting for msc^k from $SC_{c_i}^A(k)$. In the meantime, an incoming $msr_{c_j}^{k'}$ from $c_j \in SC_{c_i}$ should be considered to be valid, denoted as $valid^{k'}$ in Algorithm 1. If $T_{c_j}^k = B$, then $msr_{c_j}^{k'}$ is obviously valid because c_j is even unaffected by k. If $T_{c_j}^k = A$, then c_j must be sending $msr_{c_j}^{k'}$ to c_i after its mode switch for k. The reason is ascribed to the mode switch dependency rule which ensures that c_j must complete mode switch before c_i. Hence $msr_{c_j}^{k'}$ is also valid for c_i.

Once the validity of each pending **MSR/MSQ** is identified, each component will remove each invalid pending **MSR/MSQ** from its MSR/MSQ queues, following the *MSR/MSQ queue updating rule*:

Definition 10. ***The MSR/MSQ queue updating rule:*** *Let c_i be a component which has completed its mode switch based on Scenario k ($T_{c_i}^k = A$) or has received an* **MSC** *from all $c_j \in SC_{c_i}^A(k)$ ($c_i \in CC, T_{c_i}^k = B$). If $c_i.Q_{msr} \neq \emptyset$, then c_i will remove each invalid* **MSR** *from $c_i.Q_{msr}$. Similarly, if $c_i.Q_{msq} \neq \emptyset$, then c_i will remove each invalid* **MSQ** *from $c_i.Q_{msq}$.*

The MSR/MSQ queue updating rule is implemented as Algorithm 4, where $dequeue(A, B)$ is a function dequeuing a primitive A from queue B.

Algorithm 4 *UpdateQueue*(c_i, k)

```
if $c_i \in PC$ then
    if $\exists msr_{c_i}^{k'} \in c_i.Q_{msr}$ then
        $dequeue(msr_{c_i}^{k'}, c_i.Q_{msr})$;
    end if
else{$c_i \in CC$}
    if $(\exists msr_{c_i}^{k'} \in c_i.Q_{msr})$ && $(T_{c_i}^k = A)$ then
        $dequeue(msr_{c_i}^{k'}, c_i.Q_{msr})$;
    end if
    if $(\exists msr_{c_j}^{k'} \in c_i.Q_{msr})$ && $(c_j \in SC_{c_i})$ then
        if $T_{c_j}^k = A$ && $\neg valid^{k'}$ then
            $dequeue(msr_{c_j}^{k'}, c_i.Q_{msr})$;
        end if
    end if
    if $(c_i$ = Top$)$ && $(T_{c_i}^k = A)$ && $(\exists msq^{k'} \in c_i.Q_{msq})$ then
        $dequeue(msq^{k'}, c_i.Q_{msq})$;
    end if
    $valid^{k'} := false$;
end if
$locked := false$;
```

The essence of the MSR/MSQ queue updating rule is to remove a pending **MSR/MSQ** which becomes invalid due to the mode switch of a previous scenario. It should be noted that the MSR/MSQ queue updating rule does not remove any **MSR** or **MSQ** associated with the currently handled scenario because this is already covered by the dequeuing conditions introduced in Section III-A.

One may wonder why $c_i \in \widetilde{CC}$ does not remove a pending **MSQ**. It can be inferred that an incoming **MSQ** is pending in $c_i.Q_{msq}$ only when c_i is in transition state. Otherwise, the **MSQ** will be immediately handled by c_i. Suppose c_i is in transition state for Scenario k, with a pending $msq^{k'}$ in $c_i.Q_{msq}$. Then c_i must be the MSDM for k. Otherwise, if c_i is not the MSDM for k, P_{c_i} must also be in transition state for k and should not send $msq^{k'}$ until it leaves this transition state. Now that c_i is the MSDM for k, components out of c_i must all be Type B components for k. Therefore, k' must be valid and should not be removed.

D. Discussion

The MSR/MSQ queue checking rule always checks the MSQ queue before the MSR queue. A potential problem is the bias towards scenarios triggered by a component closer to *Top*. For instance, consider two scenarios k_1 and k_2, concurrently triggered by c_1 and c_2 respectively. *Top* is the MSDM for both scenarios and c_1 is much closer to *Top*. Since it takes more steps for msr^{k_2} to reach *Top* compared with msr^{k_1}, k_1 is more likely to be handled by *Top* before k_2 despite their simultaneous triggering. When a component c_i, with msq^{k_1} in $c_i.Q_{msq}$ and msr^{k_2} in $c_i.Q_{msr}$, checks its MSR/MSQ queues, msq^{k_1} will be first handled while msr^{k_2} may even be removed afterwards due to the MSR/MSQ queue updating rule.

However, $\forall c_i \in \widetilde{CC}$, since a pending **MSQ** will eventually be handled without being removed by the MSR/MSQ queue updating rule, it is better to assign higher priority to the MSQ queue than the other way round. For instance, if $\exists msq^k \in c_i.Q_{msq}$ and $\exists msr^{k'} \in c_i.Q_{msr}$, the mode switch completion of c_i based on k may skip the subsequent handling of $msr^{k'}$ due to the MSR/MSQ queue updating rule. Conversely, if $msr^{k'}$ is first handled, msq^k will be handled later anyway. Therefore, assigning higher priority to the MSQ queue can benefit more from the MSR/MSQ queue updating rule.

IV. Verification of the Conflict Handling Mechanism

The conflict handling mechanism, represented by the MSR/MSQ queue checking rule and the MSR/MSQ queue updating rule, extends the mode switch runtime mechanism of MSL with the support for handling multiple concurrent scenarios. The correctness of the conflict handling mechanism can be proved by the satisfaction of a number of properties, among which the two most important are:

1) The conflict handling mechanism is deadlock-free.
2) A triggered scenario will eventually be handled.

For Property 2, a scenario is also considered to be handled if its associated primitives are removed by the MSR/MSQ queue updating rule.

We resort to model checking for the verification of the conflict handling mechanism. The conflict handling mechanism is modeled in the model checker UPPAAL [8] with regard to the six cases listed in Table I. In each case, the conflict handling mechanism is implemented in a target component. Shown in Fig. 4, the target component can have a parent stub or two child stubs for some cases which simulate the behaviors of its parent and subcomponents by running the same conflict handling mechanism. Both the parent stub and child stub can trigger scenarios at any time, as long as they do not have any self-triggered pending scenarios. In [7], we manually prove the model equivalence between a stub and a real component. Property 1 can be easily verified by UPPAAL. Table IV shows the verification time[1] excluding Case (4).

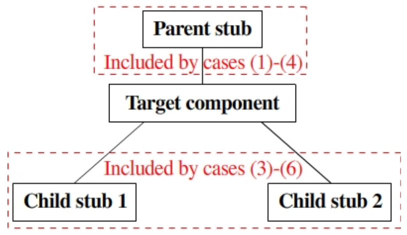

Figure 4. The modeling structure in UPPAAL

Table IV
Verification time of deadlock-freeness, excluding Case (4)

Case	Verification time
(1) $c_i \in PC, c_i \neq MSS$	0.002s
(2) $c_i \in PC, c_i = MSS$	0.013s
(3) $c_i \in \widetilde{CC}, c_i \neq MSS$	33.539s
(5) $c_i = Top, c_i \neq MSS$	0.268s
(6) $c_i = Top, c_i = MSS$	2.56s

All the five cases in Table IV are proven to be deadlock-free with reasonable verification time. The verification for Case (4) demands for more efforts in that UPPAAL ends up with memory exhaustion due to the growing model complexity. This problem is remedied by two means. The first is to use *Compact Data Structure* in UPPAAL for state space representation instead of the default Difference Bound Matrices representation. A direct consequence of this is longer verification time but much less memory consumption. In addition, we divide the model for Case (4) into four

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

simpler models that can be verified separately. Case (4) considers $c_i \in \widetilde{CC}$ which is an MSS. Each scenario k triggered by c_i can affect $SC^A_{c_i}(k)$. Since c_i has two subcomponents (e.g. c_1 and c_2) in the model, k can lead to four sub-cases concerning $SC^A_{c_i}(k)$: (a) $T^k_{c_1} = T^k_{c_2} = A$; (b) $T^k_{c_1} = A, T^k_{c_2} = B$; (c) $T^k_{c_1} = B, T^k_{c_2} = A$; (d) $T^k_{c_1} = T^k_{c_2} = B$. Cases (b) and (c) are symmetrical, hence it is sufficient to only consider three cases, e.g. (a), (b) and (d). Using Compact Data Structure and taking (a), (b) and (d) as three sub-cases of Case (4), we successfully verified the deadlock-freeness for Case (4), with the verification time summarized in Table V.

Table V
VERIFICATION TIME OF DEADLOCK-FREENESS: CASE (4) (USING COMPACT DATA STRUCTURE)

Sub-case	Verification time
(a) $T^k_{c_1} = T^k_{c_2} = A$	475.1s
(b) $T^k_{c_1} = A, T^k_{c_2} = B$	523.855s
(d) $T^k_{c_1} = T^k_{c_2} = B$	286.052s

Property 2 cannot be directly verified by the UPPAAL models made for the verification of Property 1. The reason is that these models allow the triggering of a scenario at any time if it is possible. Since the conflict handling mechanism assigns higher priority to the MSQ queue, if the parent stub keeps sending a msq^k (not associated with the **MSR** sent from the child stubs) to the target component, a pending $msr^{k'}$ from a child stub of Type B for k may never be handled. This problem should not exist in a real-world system because the triggering of a scenario is usually not a frequent event. In order to make Property 2 verifiable, we slightly modify the parent stub such that for every two consecutive **MSQ** primitives (msq^k and $msq^{k'}$) sent by the parent stub, at least either k or k' originates from a child stub. Since this modification does not alter the conflict handling mechanism, the modified models can be used to verify both properties. Property 1 and Property 2 are both satisfied for cases (1)-(6). The detail verification results can be found in [7], including proofs that the parent and child stubs faithfully model relevant aspects of an arbitrary component structure above or below the target component.

Now that the correctness of the conflict handling mechanism has been verified by model checking assuming that each composite component has two subcomponents, we can further prove its correctness for a more general system. Since the conflict handling mechanism is not dependent on the number of subcomponents of any composite component, the conflict handling mechanism works for a CBMMS where each composite component has arbitrary number of subcomponents.

V. RELATED WORK

The extended MECHATRONICUML (EUML) [9] by Heinzemann et al. is currently the most closely related work to our MSL. In EUML, each component has reconfiguration ports resembling the dedicated mode switch ports of our mode-aware component model. Each composite EUML component internally has a manager and executor which play the same role as our mode switch runtime mechanism. Components can propagate messages for requesting/executing reconfiguration. The propagation of these messages can be compared with our MSP protocol. However, the MSP protocol relies on mode mapping whereas the message propagation in EUML is controlled by some reconfiguration rules. EUML requires that reconfiguration should be performed bottom-up; this is similar to our mode switch dependency rule which in addition allows concurrent reconfigurations of different components. The major difference between EUML and MSL is that EUML focuses more on component reconfiguration without addressing mode. In general, MSL is more mature since EUML is more recently developed. EUML is also aware of the transmission of multiple request messages, which is comparable to the triggering of multiple scenarios described in this paper, however, no concrete solutions have been reported yet.

Another recent work related to MSL is the oracle-based approach by Pop et al. [10]. The basic idea is to abstract component behaviors into a property network spread throughout the component hierarchy. The mode of each component is modeled as a property and mapped from a set of properties to their valuations. A single property change can be propagated throughout the property network, potentially leading to the valuation change of other properties. And then the new mode of each component can be derived after the update of the property network. A finite-state machine called Oracle is offline constructed to guarantee predictable update time of the property network. The construction of Oracle implies that the mode switch handling requires global information of the property network. In contrast, MSL is fully distributed, requiring no global information.

Mode switch has been addressed in a number of component models, e.g. SaveCCM [11], Koala [12], Rubus [13], and MyCCM-HI [14], to name a few. In Koala and SaveCCM, a special *switch* connector is introduced to achieve the structural diversity of a component. Depending on the input data, *switch* can select one of multiple outgoing connections. In Rubus, mode is treated as a system property. A system-wide static configuration of components is defined for each mode. MyCCM-HI provides a more advanced mechanism for handling mode switch. Each MyCCM-HI component is mode-aware and is associated with a mode automaton which implements its mode switch mechanism. In addition, mode switch is also addressed by languages such as the Architecture Analysis & Design Language [15], where a state machine is used to represent the mode switch behavior of a component. Each state machine consists of a number of states (modes), transitions between these states (mode switches) and input/output event ports used for mode switch triggering. Compared with MSL, none of these works provide any systematic strategy to coordinate the mode switches of different components, due to the common assumption of independent mode switches between components.

VI. Conclusion

In this paper, we have proposed a conflict handling mechanism as a supplement to the Mode Switch Logic (MSL) dedicated to the development of Component-Based Multi-Mode Systems as well as their mode switch handling. The conflict handling mechanism enables MSL to deal with concurrent triggering of multiple mode switch scenarios. The correctness of this mechanism has been verified by model checking.

The current conflict handling mechanism can be enhanced in future by prioritizing scenarios so that scenarios with higher priorities can be handled earlier. Another potential improvement is the consideration of timeout. After triggering a scenario, a component may expect it to be handled within a specified time interval. If the pending scenario cannot be handled in time, a timeout event may be issued for further actions. We also intend to adapt MSL to safety-critical systems. According to the mode switch runtime mechanism of MSL, a scenario is rejected even if a single Type A component is not ready to switch mode. In a safety-critical system, some mode switch can be rather urgent and should not be rejected. Such a mode switch would require special treatment.

The verification of this work can be further complemented by extensive simulation. Since model checking is subject to state explosion, it is less suitable for verification of very large systems. A better alternative could be to simulate MSL and analyze its performance in different complex systems that are automatically generated. Moreover, as far as we know, no existing multi-mode systems are yet built by multi-mode components, hence an important future direction is to explore the usability of MSL in real-world systems.

Acknowledgment

This work is supported by the Swedish Research Council via the ARROWS project at Mälardalen University.

References

[1] I. Crnković and M. Larsson, *Building reliable component-based software systems.* Artech House, 2002.

[2] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A classification framework for software component models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, 2011.

[3] T. Pop, P. Hnětynka, P. Hošek, M. Malohlava, and T. Bureš, "Comparison of component frameworks for real-time embedded systems," *Knowledge and Information Systems*, pp. 1–44, 2013.

[4] Y. Hang, "Mode switch for component-based multi-mode systems," Licentiate Thesis, Mälardalen University, Sweden, December 2012.

[5] Y. Hang, J. Carlson, and H. Hansson, "Towards mode switch handling in component-based multi-mode systems," in *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, 2012.

[6] C. Bergenhem, H. Pettersson, E. Coelingh, C. Englund, S. Shladover, and S. Tsugawa, "Overview of platooning systems," in *Proceedings of 19th ITS World Congress*, October 2012.

[7] Y. Hang and H. Hansson, "Handling multiple mode switch scenarios in component-based multi-mode systems," MRTC, Mälardalen University, Tech. Rep. ISSN 1404-3041 ISRN MDH-MRTC-274/2013-1-SE, June 2013.

[8] K. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," *STTT-International Journal on Software Tools for Technology Transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[9] C. Heinzemann and S. Becker, "Executing reconfigurations in hierarchical component architectures," in *Proceedings of 16th International ACM SIGSOFT Symposium on Component Based Software Engineering*, 2013.

[10] T. Pop, F. Plasil, M. Outly, M. Malohlava, and T. Bureš, "Property networks allowing oracle-based mode-change propagation in hierarchical components," in *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, 2012.

[11] H. Hansson, M. Åkerholm, I. Crnković, and M. Törngren, "SaveCCM - a component model for safety-critical real-time systems," in *Proceedings of Euromicro Conference, Special Session on Component Models for Dependable Systems*, 2004.

[12] R. V. Ommering, F. V. D. Linden, J. Kramer, and J. Magee, "The Koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, 2000.

[13] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K. Lundbäck, "The Rubus component model for resource constrained real-time systems," in *Proceedings of 3rd International Symposium on Industrial Embedded Systems*, 2008.

[14] E. Borde, G. Haïk, and L. Pautet, "Mode-based reconfiguration of critical software component architectures," in *Proceedings of Conference on Design, Automation and Test in Europe*, 2009.

[15] 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.