Handling emergency mode switch for component-based systems

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

***Abstract*—Software reuse is deemed as an effective technique for managing the growing software complexity of large systems. Software complexity can also be reduced by partitioning the system behavior into different modes. Such a multi-mode system is able to dynamically change its behavior by switching between different modes. When 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 handle mode switch properly. This is the motivation for the Mode Switch Logic (MSL), supporting the development of component-based multi-mode systems by providing mechanisms for mode switch handling. In this paper, MSL is extended and adapted to systems with emergency triggering of mode switches that must be handled with minimal delay. We propose an Immediate Handling with Buffering (IHB) approach to enable the responsive handling of such an emergency event in the presence of other concurrent non-emergency mode switch events. We present a model checking based verification of IHB and illustrate its benefits by an example.**

***Keywords*-component; mode switch; emergency**

I. Introduction

Component-Based Software Engineering (CBSE) [1] is a paradigm for reducing mainly design time complexity, characterized by systems being composed of independently developed reusable software components. Partitioning the system behavior into different operational modes is a complementary approach that targets reduction of both design and run-time complexity. Such a multi-mode system usually runs in one mode and can switch to another mode under certain conditions. For instance, the control software of an airplane could run in the modes *taxi* (the initial mode), *taking off*, *flight* and *landing*.

Taking the advantage of both CBSE and multi-mode systems, we aim at building multi-mode systems by reusing multi-mode components. Fig. 1 illustrates a multi-mode system built by multi-mode components. The system, i.e. Component *a*, consists of components *b*, *c* and *d*. Component *c* is composed by *e* and *f*. Among these components, *b*, *d*, *e*, and *f* are *primitive components* directly implemented by code, while *a* and *c* are *composite components* composed by other components. The tree structure of the component hierarchy implies a parent-and-children relationship between each composite component and the components directly composing it. For instance, *a* is the parent of *b*, *c*, *d* which in turn are the subcomponents or children of *a*. Besides, *a* can run in two modes: m_a^1 and m_a^2. When *a* runs in mode m_a^1, *d* is deactivated (represented by the dimmed color); when *a* runs in m_a^2, *d* becomes activated and extra connections are established within *a*. In addition, *b* exhibits different mode-specific behaviors (distinguished by black and grey colors) when *a* is running in different modes. Similar to *a*, the other components may also support multiple modes.

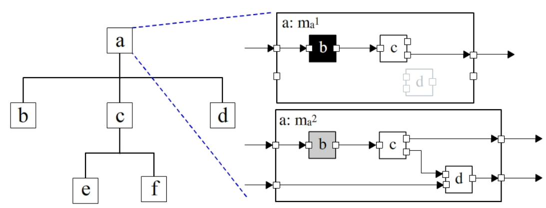

Figure 1. A multi-mode system built by multi-mode components

The key challenge for such a system is its mode switch handling in the sense that a system mode switch could correspond to the mode switches of many different independently developed components. For instance, a system mode switch from m_a^1 to m_a^2 in Fig. 1 requires the activation of *d*, the behavior change of *b*, and possibly the change of *c*, *e* and *f*. The mode switches of different components must be well synchronized and coordinated to guarantee a correct system mode switch. We have developed the *Mode Switch Logic (MSL)* [2] as the corresponding solution.

In MSL, a mode switch is triggered as an event by a single component, e.g. when a sensor value exceeds a predefined threshold. Such an event typically leads to a *mode switch scenario* or simply *scenario*, i.e. a switch of the triggering component from one mode to another mode, potentially leading to the mode switches of some other components. Hence, a scenario must be propagated to those components. MSL allows the concurrent and independent triggering of multiple scenarios by different components. However, MSL currently treats all scenarios equally, without considering their urgency. This makes MSL less suitable for use in time-critical systems where a scenario may be related to an emergency event that must be handled within a short time period. The contribution of this paper is that it extends MSL by an approach called *Immediate Handling with Buffering (IHB)* that distinguishes an emergency scenario from a non-emergency scenario and exerts itself to achieve a responsive handling of an emergency scenario with minimum impact on other (non-emergency) scenarios.

The remainder of the paper is structured as follows: Section II gives a brief introduction of MSL. In Section III, we elaborate on our IHB approach. In Section IV we present the verification of IHB. Related work is reviewed in Section V. Finally, Section VI concludes the paper and discusses some future work.

II. The Mode Switch Logic

The Mode Switch Logic (MSL) allows the hierarchical composition of multi-mode components. Each component has a unique configuration associated with each of its modes. Its mode switch is performed by *reconfiguration*, i.e. by changing its configuration in the current mode to the configuration in the new mode. A component is able to exchange mode information with its parent and subcomponents via dedicated ports. Each component handles a scenario by running a built-in *mode switch run-time mechanism (MSRM)*. We first present how the MSRM handles a single scenario, without the interference of other scenarios. Then we further explain the handling of multiple concurrent scenarios.

A. The handling of a single scenario

The component that triggers a scenario is called the *Mode Switch Source (MSS)*. After an MSS triggers a scenario, it will assign a unique scenario ID k to this triggering of the scenario, which is then propagated to the components which need to switch mode due to k. We call such components *Type A components* and components not affected by k are called *Type B components*. For each component c_i and a scenario k, $T^k_{c_i} = A$ or $T^k_{c_i} = B$ denotes that c_i is a Type A or Type B component for k. Type A/B components are identified by a mode mapping mechanism included in each composite component. This mechanism relates the modes of the parent to those of the children and vice versa. Since a component only knows the information of itself and its subcomponents, the propagation of a scenario must be stepwise, either one step up to the parent or one step down to the subcomponents. The MSRM of each component includes a *Mode Switch Propagation (MSP) protocol* [2] for the propagation of a scenario triggered by an MSS to all Type A components without disturbing Type B components. In general, the MSP protocol defines a number of primitives transmitted across different components. A scenario leads to a mode switch only if it is approved by a *Mode Switch Decision Maker (MSDM)* (a component which is usually an ancestor of the MSS) dynamically identified by the MSP protocol. The MSP protocol is presented as follows:

Definition 1. ***The MSP protocol:*** *Let c_i be an MSS triggering a scenario k and c_j be the MSDM of k. Component c_i triggers k by issuing an **`MSR`** (Mode Switch Request) primitive (denoted as msr^k) that is propagated to the parent of c_i and stepwise towards c_j. Upon receiving the msr^k, c_j checks if it is ready to switch mode. If not, c_j will reject k by issuing an **`MSD`** (Mode Switch Denial) primitive msd^k that is propagated back to c_i via the same intermediate components. Otherwise, c_j will issue an **`MSQ`** (Mode Switch Query) primitive msq^k that is propagated downstream and stepwise to all Type A components, asking if they are ready to switch mode. Upon receiving the msq^k, each component replies with an **`MSOK`** primitive $msok^k$ if ready to switch mode or with an **`MSNOK`** primitive $msnok^k$ otherwise. If all Type A components are ready to switch mode, c_j will trigger the mode switch for k by issuing an **`MSI`** (Mode Switch Instruction) primitive msi^k that follows the propagation trace of the msq^k. The propagation of k is completed when all Type A components receive the msi^k. Otherwise, if at least one Type A component replies with an $msnok^k$, c_j will abort the propagation of k by issuing an msd^k that follows the propagation trace of the msq^k.*

The formal and complete description of the MSP protocol can be found in [2]. Basically, the MSP protocol first identifies the MSDM of a scenario which then triggers a two-phase propagation. In the first phase, the MSDM asks if all Type A components are ready for the mode switch. In the second phase, the MSDM makes the final decision by either triggering or not triggering the mode switch. Mode switch is triggered when the MSDM issues an **`MSI`**.

After the propagation of an **`MSI`**, a Type A component will start reconfiguration, following a *mode switch dependency rule* which is part of its MSRM and guarantees that a mode switch is always completed bottom-up: A primitive component completes its mode switch after its reconfiguration and sends an **`MSC`** (Mode Switch Completion) primitive msc^k to its parent. A composite component c_i completes its mode switch after it completes its reconfiguration and has received an msc^k from all its Type A subcomponents. If c_i is not the MSDM of k, c_i will send an msc^k to its parent. A system mode switch is completed when: (1) the MSDM c_i completes its mode switch for k ($T^k_{c_i} = A$); or (2) the MSDM c_i has received an msc^k from all its Type A subcomponents ($T^k_{c_i} = B$).

To demonstrate the handling of a single scenario, suppose e in the example in Fig. 1 triggers a scenario k as the MSS, with a identified as the MSDM. Components b and f are Type B components while the others are Type A components. The handling of k is depicted in Fig. 2. First, an msr^k is propagated from e to its parent c, and then to the MSDM a. In Phase 1, an msq^k is propagated stepwise to Type A components, all of which are ready to switch mode. Therefore, in Phase 2, a issues an msi^k that triggers the mode switches of Type A components, whose reconfigurations are represented by the black bars in Fig. 2. Finally, an msc^k is propagated bottom-up to indicate mode switch completion. The white bars in Fig. 2 mean that the mode switch of a composite component cannot be completed after its reconfiguration because it is still waiting for an msc^k from at least one subcomponent. This complies with the mode switch dependency rule.

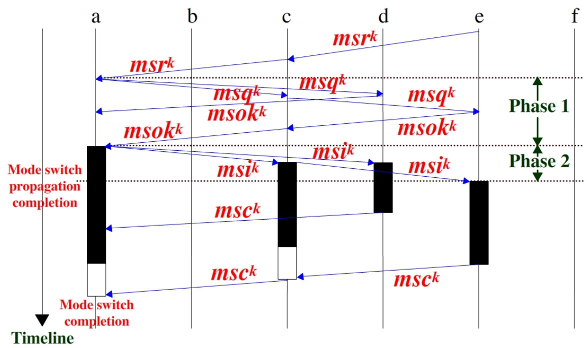

Figure 2. A mode switch based on Scenario k

B. The handling of multiple concurrent scenarios

To handle concurrent scenarios, two FIFO queues, MSR queue and MSQ queue, are introduced for each component. A component stores incoming **MSR/MSQ** primitives in the corresponding queues and handle them one at a time. Let $c_i.Q_{msr}$ and $c_i.Q_{msq}$ denote the MSR/MSQ queue of a component c_i. We use $Q[1]$ to denote the first element in the queue Q, $x \in Q$ to denote that x is one element in Q, and $Q = \varnothing$ or $Q \neq \varnothing$ to denote that Q is empty or non-empty. If c_i receives multiple scenarios simultaneously, e.g. msr^{k_1} and msr^{k_2}, then c_i puts them in $c_i.Q_{msr}$ based on their arrival order. When c_i completely handles a scenario k, if $c_i.Q_{msr}[1] = msr^k$, then c_i will remove the msr^k from $c_i.Q_{msr}$. Similarly, if $c_i.Q_{msq}[1] = msq^k$, then c_i will remove the msq^k from $c_i.Q_{msq}$.

Let $\mathcal{PC}$ and $\mathcal{CC}$ be the set of primitive components and composite components of a system, respectively. For each c_i, let P_{c_i} be the parent of c_i, $\mathcal{SC}_{c_i}$ be the set of subcomponents of c_i, and $\mathcal{SC}^A_{c_i}(k)$ be the set of Type A subcomponents of c_i for Scenario k. Then c_i completely handles k when (1) c_i completes a mode switch for k ($T^k_{c_i} = A$); (2) c_i has received an msc^k from all $c_j \in \mathcal{SC}^A_{c_i}(k)$ ($c_i \in \mathcal{CC} \wedge T^k_{c_i} = B$); (3) c_i propagates an msd^k to $\mathcal{SC}^A_{c_i}(k)$ ($c_i \in \mathcal{CC} \wedge \mathcal{SC}^A_{c_i}(k) \neq \varnothing$); (4) c_i receives an msd^k from P_{c_i} ($c_i \in \mathcal{PC} \vee \mathcal{SC}^A_{c_i}(k) = \varnothing$). A scenario cannot interrupt the ongoing mode switch of a component in a *transition state*:

Definition 2. *A component c_i is in a transition state within the interval $[t_1, t_2]$ for Scenario k, where t_1 is the time when (1) c_i issues an msq^k to $\mathcal{SC}^A_{c_i}(k)$ (when c_i is the MSDM of k); or (2) c_i handles an $msq^k \in c_i.Q_{msq}$. And t_2 is the time when c_i has completely handled k.*

An *MSR/MSQ queue checking rule* is based on Definition 2: If c_i is not in any transition state, then if $c_i.Q_{msq} \neq \varnothing$, c_i will immediately handle $c_i.Q_{msq}[1]$; else if $c_i.Q_{msr} \neq \varnothing$ and $c_i.Q_{msr}[1]$ has not been propagated to P_{c_i}, c_i will immediately handle $c_i.Q_{msr}[1]$.

Note that the handling of one scenario of a component may affect its handling of a subsequent scenario. For instance, in Fig. 2, if a receives another $msr^{k'}$ from d right after the reception of msr^k from c, then a will handle k first. Since $T^k_d = A$, d will switch mode due to k. However, d triggers k' in the old mode, implying that $msr^{k'}$ becomes invalid. Therefore, both a and d should remove the $msr^{k'}$ from their MSR queues after the mode switch for k. This is achieved by an *MSR/MSQ queue updating rule* which is referred to [2] due to limited space.

III. Emergency mode switch handling

In time-critical systems, a scenario may be triggered by an emergency event which requires a responsive and exclusive handling compared with non-emergency scenarios. To support this, and as the contribution of this paper, we extend the MSRM of each component by the Immediate Handling with Buffering (IHB) approach while assuming:

1) A system has at most one emergency scenario, which can be recognized by all components.
2) From each mode a direct switch to the emergency mode is possible.
3) Primitives sent between components are received in the same order they are sent.
4) Component reconfiguration cannot be interrupted.

Assumptions 1 and 2 can be statically checked at design time, and Assumption 3 can be assured by the inter-component communication infrastructure. Assumption 4 is a precondition for IHB.

A. The handling of an emergency scenario

An emergency scenario k can be propagated by an **EMS** (Emergency Mode Switch) primitive ems^k. Once the ems^k is triggered, it should never be rejected and a mode switch must be performed in time. Let Top be the component at the top of the component hierarchy. An emergency scenario can be propagated by following the *Emergency Mode Switch Propagation (EMSP) protocol*:

Definition 3. ***The EMSP protocol:*** *Let c_i be the MSS of an emergency scenario k. Then, (1) If $c_i \in \mathcal{PC}$, it will send an ems^k to P_{c_i}; (2) If $c_i \in \mathcal{CC} \setminus \{Top\}$, it will send an ems^k to P_{c_i} and $\mathcal{SC}^A_{c_i}(k)$; (3) If $c_i = Top$, it will send an ems^k to $\mathcal{SC}^A_{c_i}(k)$.*

For each c_j that receives the ems^k, (1) If $c_j \in \mathcal{PC}$, no further propagation is needed; (2) If $c_j \in \mathcal{CC} \setminus \{Top\}$, it propagates the ems^k depending on the sender c_n and $T^k_{c_j}$: If $c_n = P_{c_j}$, c_j will propagate the ems^k to $\mathcal{SC}^A_{c_j}(k)$; if $c_n \in \mathcal{SC}_{c_j}$ and $T^k_{c_j} = A$, then c_j will propagate the ems^k to $\{P_{c_j}\} \cup \mathcal{SC}^A_{c_j}(k) \setminus \{c_n\}$; if $c_n \in \mathcal{SC}_{c_j}$ and $T^k_{c_j} = B$, then c_j will propagate the ems^k to $\mathcal{SC}^A_{c_j}(k) \setminus \{c_n\}$ as the MSDM of k; (3) If $c_j = Top$, then c_j will propagate the ems^k to $\mathcal{SC}^A_{c_i}(k) \setminus \{c_n\}$, where $c_n \in \mathcal{SC}_{c_j}$.

Unlike a non-emergency scenario, the immediate handling of an emergency scenario is a critical issue that must be guaranteed even at the sacrifice of enforcing a component to switch mode. After the propagation of ems^k, a Type A component will start its reconfiguration following the original mode switch dependency rule.

To demonstrate the EMSP protocol, k in Fig. 2 is handled as an emergency scenario in Fig. 3. Compared with Fig. 2, it is self-evident that the propagation of an emergency scenario is faster than that of a non-emergency scenario.

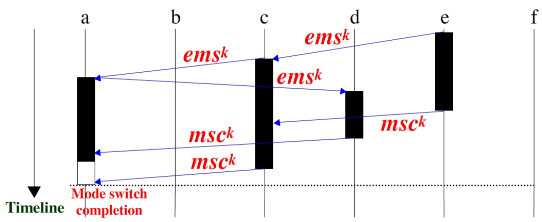

Figure 3. Demonstration of the EMSP protocol

Since we assume that component reconfiguration cannot be interrupted, an ongoing reconfiguration of a component may delay its handling of an **EMS**. Hence we introduce an EMS queue for each component to store an incoming **EMS**. The EMS queue of c_i is denoted as $c_i.Q_{ems}$ and is of size 1, since we assume that only one emergency scenario is specified for each system. When c_i triggers or receives an ems^k, it will put the ems^k in $c_i.Q_{ems}$. Component c_i removes the ems^k from $c_i.Q_{ems}$ when it completes the handling of the ems^k, i.e. when (1) c_i has completed its mode switch for k ($T^k_{c_i} = A$); or (2) c_i has received an msc^k from all $c_j \in \mathcal{SC}^A_{c_i}(k)$ ($T^k_{c_i} = B$). During the handling of the ems^k, c_i is in an *Emergency Transition State (ETS)*:

Definition 4. *A component c_i is in an Emergency Transition State (ETS) within the interval $[t_1, t_2]$ for an emergency scenario k, where t_1 is the time when c_i starts to handle the ems^k in $c_i.Q_{ems}$ and t_2 is the time when c_i has completed the handling of k.*

Hereafter we use Normal Transition State (NTS) to indicate a transition state (Definition 2). An MSS should not trigger a scenario in an NTS or ETS.

Reading from the EMS queue of a component has higher priority than reading from its MSR and MSQ queues. We replace the MSR/MSQ queue checking rule with the following *pending scenario checking rule*:

Definition 5. ***The pending scenario checking rule:*** *If c_i is not in an NTS or ETS, it periodically checks its EMS/MSQ/MSR queues until it identifies a primitive x that is immediately handled by c_i, where*

- *If $c_i.Q_{ems} \neq \emptyset$, then $x = c_i.Q_{ems}[1]$.*
- *If $c_i.Q_{ems} = \emptyset \wedge c_i.Q_{msq} \neq \emptyset$, then $x = c_i.Q_{msq}[1]$.*
- *If $c_i.Q_{ems} = \emptyset \wedge c_i.Q_{msq} = \emptyset \wedge c_i.Q_{msr} \neq \emptyset$ and $c_i.Q_{msr}[1]$ has not been propagated to P_{c_i}, then $x = c_i.Q_{msr}[1]$.*

As c_i leaves an ETS for k, it can apply the same MSR/MSQ queue updating rule as in [2] to remove elements in its MSQ/MSR queues which become invalid due to k.

B. Issues due to concurrent triggering of emergency and non-emergency scenarios

A component c_i may receive a downstream **EMS** from the parent or an upstream **EMS** from a subcomponent. The **EMS** triggered by c_i is also considered as an upstream **EMS** for c_i. After a comprehensive analysis of all the possible cases where an upstream/downstream emergency scenario interleaves with a non-emergency scenario, we have identified three major issues related to the concurrent triggering of both emergency and non-emergency scenarios.

Issue 1: When a component c_i switches mode due to an upstream **EMS** (ems^{k_2}), c_i may have already sent an **MSR** (msr^{k_1}) to P_{c_i}, with k_2 invalidating the msr^{k_1}.

Issue 1 is illustrated by Fig. 4(a), where *b* receives an ems^{k_2} ($T^{k_2}_b = A$) from *d* after sending an msr^{k_1} to *a*. Since *b* is not in the NTS for k_1, according to Definition 5, *b* will handle the ems^{k_2} and switch to the new mode, making the msr^{k_1} previously sent to *a* invalid. Hence, *b* must abort the handling of the msr^{k_1} and notify *a* as well.

Issue 2: An upstream emergency scenario may make an **MSQ** in the MSQ queue invalid.

Issue 2 is illustrated by Fig. 4(b) where *b* receives an msq^{k_1} from *a* and an ems^{k_2} from *d* at the same time. Scenario k_1 is triggered by *e* while k_2 is triggered by *d*. Component *b* will put the msq^{k_1} in $b.Q_{msq}$ and put the ems^{k_2} in $b.Q_{ems}$. According to Definition 5, *b* handles the ems^{k_2} first. If $T^{k_2}_b = A$, *b* will switch mode based on k_2. However, *a* sends the msq^{k_1} to *b* assuming that *b* is in its old mode. Therefore, k_2 makes the msq^{k_1} invalid.

Issue 3: When an ems^{k_2} arrives at a component c_i which is in an NTS for k_1, the handling of ems^{k_2} could be unnecessarily delayed by k_1.

Issue 3 is illustrated by Fig. 4(c). Component *b* receives an msq^{k_1} from its parent *a* and then propagates the msq^{k_1} to its subcomponents *c* and *d* at t_0. Meanwhile, *c* has sent an ems^{k_2} to *b* before *c* receives the msq^{k_1}. Since *b* has entered the NTS for k_1 at t_0, it will complete the handling of k_1 before it can handle ems^{k_2}. However, since no component has started its reconfiguration for k_1, it is possible to abort the handling of k_1 to facilitate the handling of k_2.

C. Solutions to the identified issues

The issues pinpointed in Section III-B pose extra challenge to the handling of concurrent emergency and non-emergency scenarios.

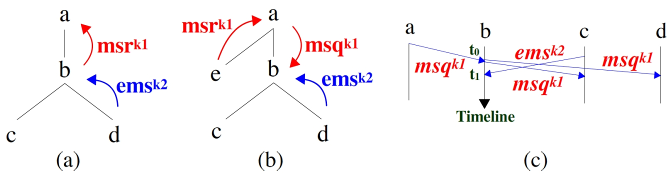

Figure 4. Issues due to the concurrent triggering of both emergency and non-emergency scenarios

Concerning Issue 1, let's first observe the behavior of the MSS c_i of an emergency scenario. Suppose c_i just sends an msr^{k_1} to P_{c_i} at t_1 and receives an msq^{k_1} from P_{c_i} at t_2. Since c_i is not in the NTS for k_1 at the interval $[t_1, t_2]$, c_i may trigger an emergency scenario k_2 within this interval. Before issuing an ems^{k_2}, c_i should realize that the msr^{k_1} previously sent to P_{c_i} becomes invalid due to k_2. Hence c_i should abort the handling of k_1 and notify P_{c_i} and $\mathcal{SC}_{c_i}$. An msd^{k_1} can be sent from c_i to $c_j \in \mathcal{SC}_{c_i}$ which aborts the handling of k_1. Similarly, we introduce an upstream **MSA** (Mode Switch Abort) primitive so that P_{c_i} can abort the handling of k_1 while receiving an msa^{k_1} from c_i.

Upon receiving an msa^k, a component can abort the handling of k by applying the *MSA handling rule*:

Definition 6. ***The MSA handling rule:*** *Let $c_i \in \mathcal{CC}$ be a component that receives an msa^k from $c_j \in \mathcal{SC}_{c_i}$.*

- *If there is one $msr^{k_1}_{c_j} \in c_i.Q_{msr}$, then $k = k_1$ and c_i will remove it from $c_i.Q_{msr}$.*
- *If there are two **MSR** primitives[1] from c_j in $c_i.Q_{msr}$, let the first be $msr^{k_1}_{c_j}$ and the second be $msr^{k_2}_{c_j}$ ($k_1 \neq k_2$). Then $k = k_2$. If c_i is in the NTS for k_1, then c_i will only remove $msr^{k_2}_{c_j}$ from $c_i.Q_{msr}$. Otherwise, c_i will remove both $msr^{k_1}_{c_j}$ and $msr^{k_2}_{c_j}$ from $c_i.Q_{msr}$.*

If $c_i.Q_{msr}[1] = msr^k_{c_j}$ that has been propagated to P_{c_i}, then c_i will propagate the msa^k further up to P_{c_i}. If $\exists msq^k \in c_i.Q_{msq}$, c_i will remove the msq^k from $c_i.Q_{msq}$.

In addition, if c_i has propagated an msq^k to $\mathcal{SC}^A_{c_i}(k)$ without receiving all the replies, then c_i will leave the NTS for k by sending an msd^k to $\mathcal{SC}^A_{c_i}(k) \setminus \{c_j\}$.

The purpose of the rule above is to make all components, which have received the propagation of k, abort the handling of k. The MSA handling rule is demonstrated by Fig. 5. A composite component b, with a as its parent and c and d as its subcomponents, receives an msa^k from c at t_0, right after propagating an msq^k to c and d. By Definition 6, b first removes the msr^k and msq^k from $c_i.Q_{msr}$ and $c_i.Q_{msq}$ respectively. Since b has sent an msr^k to a, an msa^k is sent from b to a, which will also apply the MSA handling rule. Moreover, since b has propagated an msq^k to c and d while d still does not know that the msq^k has become invalid, b also sends an msd^k to d. Component c will ignore the msq^k from b after sending the msa^k to b.

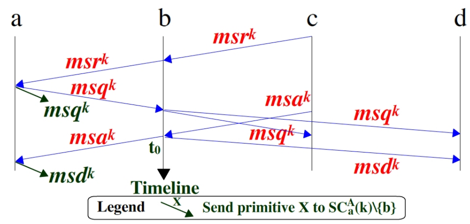

Figure 5. Demonstration of the MSA handling rule

The MSA handling rule only solves Issue 1 identified in Section III-B. Concerning Issue 2, we propose a *preliminary EMS handling rule* which is applied before each component propagates an upstream **EMS**. This rule consists of two parts:

Definition 7. ***The preliminary EMS handling rule (Part 1):*** *Suppose c_i is about to handle an upstream ems^{k_2}. Let $c_i.Q_{msr}[1] = msr^{k_1}_{c_l}$ ($c_l \in \mathcal{SC}_{c_i} \cup \{c_i\}$) if $c_i.Q_{msr} \neq \varnothing$. If $c_i \neq Top$ and $msr^{k_1}_{c_l}$ has been propagated to P_{c_i}, then c_i will send an msa^{k_1} to P_{c_i} when one of the following two conditions is satisfied: (1) $T^{k_2}_{c_l} = A$; (2) $T^{k_2}_{c_l} = B$ and $T^{k_2}_{c_i} = A$. After sending the msa^{k_1}, if $\exists msq^{k_1} \in c_i.Q_{msq}$, then c_i will remove the msq^{k_1} from $c_i.Q_{msq}$.*

The purpose of Definition 7 is to check if c_i has sent an msr^{k_1} to P_{c_i} which becomes invalid due to an upstream ems^{k_2}, where k_1 and k_2 come from different components. If yes, c_i should abort the handling of k_1 and notifies P_{c_i} further. Definition 7 is followed by Part 2 of this rule:

Definition 8. ***The preliminary EMS handling rule (Part 2):*** *After c_i applies Part 1,*

- *If $c_i = Top$, then if $\exists msq^{k_1} \in c_i.Q_{msq}$ and $T^{k_2}_{c_i} = A$, c_i will remove the msq^{k_1} from $c_i.Q_{msq}$.*
- *If $c_i \neq Top$, then if $\exists msq^{k_1} \in c_i.Q_{msq}$ and $T^{k_2}_{c_i} = A$, c_i will send an $msnok^{k_1}$ to P_{c_i} and waits for an msd^{k_1} from P_{c_i}. After receiving the msd^{k_1}, c_i removes the msq^{k_1} from $c_i.Q_{msq}$.*

The purpose of Part 2 is for c_i to abort the handling of an msq^{k_1} while k_1 comes from c_i itself or P_{c_i}. If k_1 comes from P_{c_i}, c_i cannot send an msa^{k_1} to P_{c_i} in that the msa^{k_1} is only sent if c_i has sent an msr^{k_1} to P_{c_i}. Instead, c_i can abort the handling of k_1 by sending an $msnok^{k_1}$ to P_{c_i}, according to the MSP protocol. However, the $msnok^{k_1}$ is only sent when $T^{k_2}_{c_i} = A$ (i.e. when the emergency mode switch of c_i makes the msq^{k_1} invalid).

The preliminary EMS handling rule is demonstrated by Fig. 6 with the same example as in Fig. 5. In Fig. 6(a), b has sent an msr^{k_1} to its parent a and then receives an upstream ems^{k_2} from a subcomponent c. Since $T^{k_2}_b = A$, b sends an msa^{k_1} to a following Part 1 before propagating the ems^{k_2}. In Fig. 6(b), b simultaneously receives an msq^{k_1} and

[1] In [3] we explain that at most two **MSR** primitives from the same subcomponent can co-exist in $c_i.Q_{msr}$.

an upstream ems^{k_2}. Following Part 2, b sends an $msnok^{k_1}$ to a to abort the handling of k_1 since $T_b^{k_2} = A$.

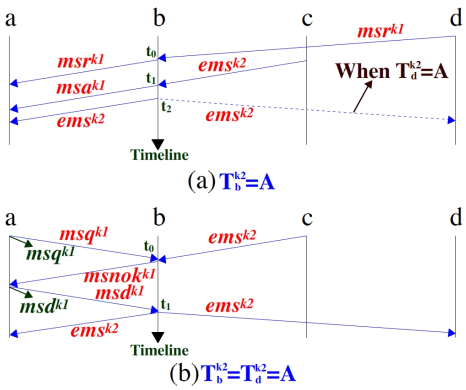

Figure 6. Demonstration of the preliminary EMS handling rule

Issue 3 can be resolved by making each composite component apply the following *EMS receiving rule* as it puts an **EMS** in the EMS queue:

Definition 9. ***The EMS receiving rule****: Let $c_i \in \mathcal{CC}$ be a component that propagates an msq^{k_1} to $\mathcal{SC}_{c_i}^A(k_1)$ and then receives an ems^{k_2} from $c_j \in \{P_{c_i}\} \cup \mathcal{SC}_{c_i}$ before c_i receives all the expected $msok^{k_1}$ or $msnok^{k_1}$ from $\mathcal{SC}_{c_i}^A(k_1)$. As c_i puts the ems^{k_2} in $c_i.Q_{ems}$, c_i will abort the handling of k_1 by propagating an msd^{k_1} to $\mathcal{SC}_{c_i}^A(k_1)$. If $c_i \neq Top$, $T_{c_i}^{k_1} = A$ and $c_j \in \mathcal{SC}_{c_i}$, c_i will send an msa^{k_1} to P_{c_i}.*

Fig. 7 demonstrates the EMS receiving rule with the same example in Fig. 6. Component b receives an ems^{k_2} from c right after propagating an msq^{k_1} to $\mathcal{SC}_b^A(k_1)$ with $T_b^{k_1} = A$. To abort the handling of k_1, b sends an msa^{k_1} to a and sends an msd^{k_1} to c and d. Then b can immediately handle k_2.

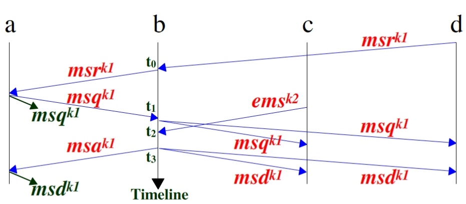

Figure 7. Demonstration of the EMS receiving rule

D. Summary of the IHB approach

As an increment of the MSRM of MSL, our IHB approach brings the following new elements:

1) The EMSP protocol
2) The MSA handling rule
3) The preliminary EMS handling rule
4) The EMS receiving rule

In addition, IHB replaces the MSR/MSQ queue checking rule introduced in [2] with the pending scenario checking rule (Definition 5). The other elements of the MSRM, including the MSP protocol, the mode switch dependency rule and the MSR/MSQ queue updating rule, remain unchanged.

The workflow of IHB is depicted in Fig. 8, where its essential elements are marked in red. The MSA handling rule and the EMS receiving rule are not visible in the figure. Instead, they can be implemented in a separate module which is triggered when an **MSA** or **EMS** arrives.

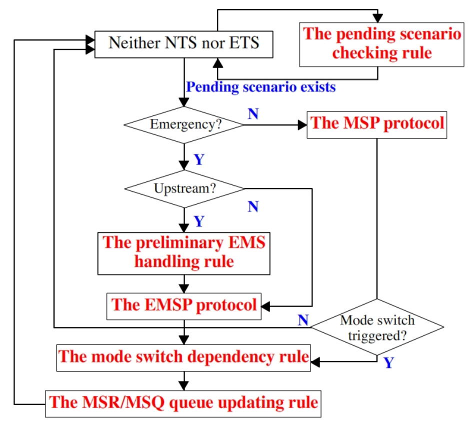

Figure 8. The workflow of IHB

We have described the complete set of algorithms for IHB by pseudocode which are excluded from this paper due to space limitation but can be found in the technical report [3].

E. Improvement by IHB

We use an example to demonstrate how the handling of an emergency scenario is improved by IHB. Depicted in Fig. 9, three scenarios: k_0, k_1 and k_2 are concurrently triggered, marked in different colors. For each scenario, the Type A components are enclosed in the corresponding dotted loop. Two mode switch processes are compared, one on the left (when k_2 triggered by e is a non-emergency scenario) and the other on the right (when k_2 is an emergency scenario). Component b receives an msq^{k_0} from a and an msr^{k_1} from d at the same time. After that, b receives either an msr^{k_2} or ems^{k_2} from e. The handling of the msr^{k_2} by b is delayed first by k_0 and then by k_1, while b handles the ems^{k_2} immediately. As b receives the ems^{k_2}, it aborts the handling of k_0 by sending an msa^{k_0} to a and an msd^{k_0} to c, driven by the EMS receiving rule. After that, b immediately propagates the ems^{k_2} and starts its reconfiguration. Apparently, IHB brings substantial improvement to the mode switch time of the emergency scenario k_2.

IV. Verification

The major concern of our verification is to prove that our IHB approach satisfies the following two key properties:

1) Deadlock freeness: IHB is deadlock-free.
2) Completeness: a component completes the handling of each scenario within bounded time.

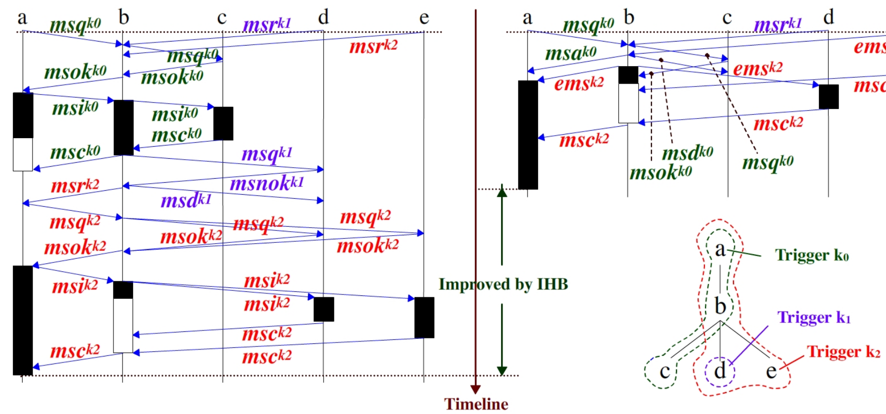

Figure 9. Improvement by IHB when a non-emergency scenario k_2 (diagram to the left) becomes an emergency scenario (diagram to the right)

We resort to model checking for the verification of IHB, using the model checker UPPAAL [4]. However, since model checking requires that a specific model instance is provided, we divide our verification into two steps:

1) Building an abstract UPPAAL model that implements IHB and satisfies the specified properties.
2) Proving that the UPPAAL model faithfully captures the relevant behavior of an arbitrary complex finite system of components.

A. Verification of the abstract model

Inspired by [5], we construct an abstract system model in UPPAAL by using stubs. IHB is implemented on a single target component while the rest of the system is simulated by a parent stub and a number of child stubs. Illustrated in Fig. 10, the modeled system consists of four components: a target component b together with a parent stub a and two child stubs c and d. Non-emergency scenarios can arrive at b from all stubs, e.g. an msq^{k_0} from a, an msr^{k_1} from c, and an msr^{k_2} from d. Note that k_0 here could be equal to k_1 or k_2 according to the MSP protocol. Besides, b can receive either a downstream ems^{k_3} from a or an upstream ems^{k_3} from c. All scenarios can be recurrently triggered whenever possible. We do not consider the case when b triggers a scenario itself because this can be simulated by adding a virtual child stub of b which triggers the scenario instead.

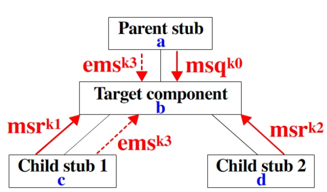

Figure 10. The modeling structure in UPPAAL

Our UPPAAL modeling includes three cases: (1) $b = Top$; (2) $b \in \mathcal{CC} \setminus \{Top\}$; and (3) $b \in \mathcal{PC}$. The system has no parent stub for Case (1) and has no child stubs for Case (3). Case (2) yields the most complex UPPAAL model, especially in the presence of an upstream ems^{k_3}. To reduce verification time, we assume that an upstream ems^{k_3} is triggered only once for Case (2). Since an emergency scenario is a rare event, even if it can be triggered multiple times, the interval between two such events must be long enough for a component to complete each emergency mode switch. Then each triggering of the ems^{k_3} is independent and it is sufficient to trigger k_3 only once in the model.

The verification of both properties (deadlock freeness and completeness) was repeated for all cases. Property 1 is always satisfied whereas Property 2 is not satisfied under certain conditions. For instance, in Case (2), Property 2 is not satisfied for the msr^{k_1} from c and the msr^{k_2} from d. This result does not reflect any error of IHB or our model. On the contrary, it is expected because a can send an msq^{k_0} to b whenever possible. Since $b.Q_{msq}$ has a higher priority than $b.Q_{msr}$, b may never handle the **MSR** from c or d if a keeps sending the msq^{k_0} to b. We allow a to send the msq^{k_0} to b unlimited number of times, however, Property 2 is only guaranteed when the constant arrival of the msq^{k_0} from a to b is bounded. Therefore, we duplicated our model into two versions. In the second version, we slightly changed the behavior of a such that for every two consecutive **MSQ** primitives[2](msq^k and $msq^{k'}$) from a, at least either k or k' is from c or d. This does not alter the nature of IHB, yet satisfying Property 2. Actually, it is only theoretically possible that b keeps receiving scenarios from a without breaks since mode switch should not be a frequent event.

B. Generalization of the UPPAAL verification results

In order to generalize our UPPAAL verification results, we need to prove that our UPPAAL model faithfully represents an arbitrary complex finite system of components. This boils down to proving the following three assertions:

[2]More generally, for every n consecutive **MSQ** primitives from a, there must exist an msq^k such that k originates from c or d. Here $n = 2$ for simplifying reasons.

1) The parent stub faithfully represents an arbitrary finite structure of components above the target component.
2) A child stub faithfully represents a subcomponent with an arbitrary finite structure of enclosed components.
3) Two child stubs faithfully represent an arbitrary number of child stubs.

The detail description of our UPPAAL model, the complete verification results, and the proof of these assertions are omitted due to limited space, but provided in [3].

V. Related work

In extended MechatronicUML (EUML) [5] by Heinzemann et al., component reconfiguration can be propagated and executed at different hierarchical levels. Reconfiguration rules can be specified for each component at design time. So far EUML has not provided any concrete solution to the handling of concurrent multiple reconfiguration requests.

Pop et al. [6] abstract component behaviors into a global property network. The value change of a property of one component can be propagated throughout the property network, potentially changing the values of some properties of the other components. Mode switch is handled by a global manager using a finite-state machine to guarantee predictable update time of the property network. In contrast, the mode switch handling of MSL is fully distributed.

Mode switch has been addressed in a number of component models, e.g. SaveCCM [7], Koala [8], Rubus [9], and MyCCM-HI [10]. 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 and a system-wide configuration of components is defined for each mode. In MyCCM-HI, each component has a mode automaton implementing its mode switch mechanism. Mode switch is also addressed by languages such as AADL [11], where a state machine is used to represent the mode switch behavior of a component. Compared with MSL, none of these works provide any systematic strategy to coordinate the mode switches of different components. To the best of our knowledge, no work is found on emergency mode-switch handling.

VI. Conclusion and future work

Software complexity can be effectively reduced and managed by reusing software components and introducing modes. We have proposed the Mode Switch Logic (MSL) for developing multi-mode systems by multi-mode components and handling their mode switch. In this paper, MSL is extended with handling of both emergency and non-emergency concurrently triggered mode switch scenarios. we have proposed an Immediate Handling with Buffering (IHB) approach that is able to handle an emergency scenario swiftly in spite of triggering of concurrent non-emergency scenarios. Using model checking based verification, IHB is proven to satisfy the desired properties such as deadlock freeness and completeness.

Future work includes to extend IHB by supporting the triggering of multiple emergency scenarios with different criticality levels. It is also our intention to provide the mode switch timing analysis for IHB for calculating the worst-case mode switch times of both emergency and non-emergency scenarios. This can be achieved by extending our previous analysis for a single non-emergency scenario [12]. Additionally, IHB does not allow an emergency scenario to abort an ongoing component reconfiguration, thus incurring an unacceptable delay to the handling of an emergency scenario if some component has extremely long reconfiguration time. We shall investigate how an emergency scenario can be immediately handled without delay, even at the sacrifice of aborting an ongoing reconfiguration. We also plan to evaluate IHB in a real-world system.

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] Y. Hang and H. Hansson, “Handling multiple mode switch scenarios in component-based multi-mode systems,” in *Proc. APSEC’13*, 2013, pp. 404–413.

[3] ——, “Handling emergency mode switch for component-based systems—an extended report,” Mälardalen University, Tech. Rep. MDH-MRTC-283/2014-1-SE, June 2014.

[4] 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.

[5] C. Heinzemann and S. Becker, “Executing reconfigurations in hierarchical component architectures,” in *Proc. CBSE’13*, 2013.

[6] T. Pop, F. Plasil, M. Outly, M. Malohlava, and T. Bureš, “Property networks allowing oracle-based mode-change propagation in hierarchical components,” in *Proc. CBSE’12*, 2012.

[7] H. Hansson, M. Åkerholm, I. Crnković, and M. Törngren, “SaveCCM - a component model for safety-critical real-time systems,” in *Proc. Euromicro Conference*, 2004.

[8] 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.

[9] 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 *Proc. SIES’08*, 2008.

[10] E. Borde, G. Haïk, and L. Pautet, “Mode-based reconfiguration of critical software component architectures,” in *Proc. DATE’09*, 2009.

[11] 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, February 2006.

[12] Y. Hang and H. Hansson, “Mode switch timing analysis for component-based multi-mode systems,” *Journal of Systems Architecture*, vol. 59, no. 10, Part D, pp. 1299–1318, 2013.