Handling emergency mode switch
for component-based systems—An extended report

Yin Hang, Hans Hansson
Mälardalen Real-Time Research Centre
Mälardalen University
Västerås, Sweden
{young.hang.yin,hans.hansson}@mdh.se
October 7, 2014

Abstract
Component-Based Software Engineering has been introduced as a technique for the development of complex systems. Software complexity can also be reduced by partitioning the system behavior into different modes. Such a multi-mode system is able to change its behavior at runtime by switching between different operational modes. When a multimode 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. 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 report, MSL is extended and adapted to systems with emergency triggering of mode switches that must be handled with minimal delay. We propose an approach, called Immediate Handling with Buffering (IHB), 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 the IHB approach, and illustrate its benefits by a small example.

1 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 set our focus on a Component-Based Multi-Mode System (CBMMS), i.e. a multi-mode system developed in a component-based manner. Fig. 1 illustrates a CBMMS, with its component hierarchy (left) and its component connections (right). The system, i.e. Component a, consists of three 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. In contrast, a and c are composite components composed by other 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: m₁ and m₂. When the system is in m₃, Component d is deactivated (i.e. not running), shown
in the component hierarchy in Fig. 1 by not displaying $d$ in mode $m^1_a$. In contrast, when the system is in $m^2_a$, $d$ 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](image)

The key challenge for a CBMMS is its mode switch handling. The mode switch of such a system could correspond to the mode switches of many different independently developed components. For instance, a system mode switch from $m^1_b$ to $m^2_b$ in Fig. 1 requires the activation of $d$, the deactivation of $f$ and the behavior change of $b$. The mode switches of different components must be well synchronized and coordinated to guarantee a correct system mode switch. The Mode Switch Logic (MSL) [2] has been developed as the corresponding solution.

In MSL, a mode switch is triggered as an event by a single component. For instance, a component may trigger a mode switch 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 from one mode (the current mode) to another mode (the new mode). A scenario may lead to the mode switches of a number of 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. For example, many embedded systems are expected to switch to a safe mode to eliminate or minimize the consequence of an emerging fault. The scenario that requests to switch to the safe mode is typically an emergency scenario. The contribution of this report 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 report is structured as follows: Section 2 gives a brief introduction of MSL. In Section 3, we elaborate on our IHB approach. In Section 4 we present the verification of IHB. Section 5 shares some thoughts on additional improvement of IHB. Related work is reviewed in Section 6. Finally, Section 7 concludes the report and discusses some future work.

## 2 The Mode Switch Logic

The Mode Switch Logic (MSL) not only enables the hierarchical composition of multi-mode components into CBMMSs, but also is able to consistently coordinate the mode
switches of different components, even when multiple mode switch scenarios are concurrently triggered. We shall first present how MSL handles a single scenario, without the interference of other scenarios. Then we further explain the handling of multiple concurrent scenarios.

2.1 The handling of a single scenario

MSL comprises three major elements: a mode-aware component model, a mode mapping mechanism and a mode switch runtime mechanism.

The mode-aware component model allows a component to support multiple modes and have a unique configuration defined for each mode. A mode switch leads to the reconfiguration of a component by changing its configuration in the current mode to a new configuration in the target mode. Each component is able to exchange mode related information with its parent and subcomponents via dedicated mode switch ports.

Each component runs a mode switch runtime mechanism that controls its mode switch behavior during a mode switch. 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 will 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. A Mode Switch Propagation (MSP) protocol [2] has previously been proposed 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 via their dedicated mode switch ports. A scenario leads to a mode switch only if it is approved by a Mode Switch Decision Maker (MSDM) which is a component with higher authority dynamically identified by the MSP protocol based on mode mapping and the current state of the component. The MSDM is usually an ancestor of the MSS in the component hierarchy. The MSP protocol can be presented as follows:

**Definition 1. The Mode Switch Propagation (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 issue an **MSD** (Mode Switch Denial) primitive $msd^k$ that is propagated back to $c_i$ via the same intermediate components. If ready to switch mode, $c_j$ will issue an **MSQ** (Mode Switch Query) primitive (denoted as $msq^k$) 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 (denoted as $msok^k$) if ready to switch mode or with an **MSNOK** (denoted as $msnok^k$) 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 (denoted as $msi^k$) 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 $msq^k$. 

3
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 the mode switch dependency rule which guarantees that a mode switch is always completed in a bottom-up manner: 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 for Scenario $k$ 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 ($T^k_{c_i} = B$).

The MSP protocol and the mode switch dependency rule are demonstrated by the simple example in Fig. 2, where a scenario $k$ is triggered by the MSS $e$. Components in the red dotted loop are Type A components for $k$ and $a$ is the MSDM. Fig. 3 depicts the complete mode switch process for $k$. First an $msr^k$ is propagated from the MSS $e$ 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. 3. Finally, an $msc^k$ is propagated bottom-up to indicate mode switch completion. The white bars in Fig. 3 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 scenario $k$ triggered by the MSS $e$](image)

### 2.2 The handling of multiple concurrent scenarios

MSL also supports the handling of multiple scenarios concurrently triggered by different MSSs. The basic idea is to let each component store incoming MSR or MSQ primitives in its MSR and MSQ queues (both are FIFO queues) and handle them one at a time. The MSR/MSQ queue of a component $c_i$ is denoted as $c_i.Q_{msr}$ and $c_i.Q_{msq}$ respectively. We shall use $Q[1]$ to denote the first element in the queue $Q$, $x \in Q$ to denote that $x$ is one of the elements in $Q$, and $Q = \emptyset$ and $Q \neq \emptyset$ to denote that $Q$ is empty and non-empty, respectively. 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 CBMMS, respectively. For each \( c_i \), let \( P_{c_i} \) be the parent of \( c_i \), \( SC_{c_i} \) be the set of subcomponents of \( c_i \), and \( SC_{c_i}^A(k) \) 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 \( c_j \in SC^A_{c_i}(k) \) \( (c_i \in \mathcal{CC} \land T^k_{c_i} = B) \); (3) \( c_i \) propagates an \( msd^k \) to \( SC^A_{c_i}(k) \) \( (c_i \in \mathcal{CC} \land SC^A_{c_i}(k) \neq \emptyset) \); (4) \( c_i \) receives an \( msd^k \) from \( P_{c_i} \) \( (c_i \in \mathcal{PC} \lor SC^A_{c_i}(k) = \emptyset) \). A component can be in a transition state which prevents its ongoing mode switch from being interrupted by a new scenario:

**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 \( SC^A_{c_i}(k) \) (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 this definition: If \( c_i \) is not in any transition state, then if \( c_i.Q_{msq} \neq \emptyset \), \( c_i \) will handle \( c_i.Q_{msq}[1] \), else if \( c_i.Q_{msr} \neq \emptyset \) and \( c_i.Q_{msr}[1] \) has not been propagated to \( P_{c_i} \), \( c_i \) will handle \( c_i.Q_{msr}[1] \).

When a component \( c_i \) completely handles a scenario \( k \), some MSR/MSQ primitives in \( c_i.Q_{msr} \) and \( c_i.Q_{msq} \) may become invalid due to \( k \). 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 \( k \). 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 applied before a component leaves a transition state. We refer to [3] for the complete description of the handling of multiple scenarios.

### 3 Emergency mode switch handling

In time-critical systems, a mode switch 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 report, we extend the mode switch runtime mechanism of MSL by the Immediate Handling with Buffering (IHB) approach, with the following assumptions:
1. At most one emergency scenario is defined for a system and this emergency scenario 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. The reconfiguration of a component cannot be aborted.

5. There is no mode switch failure, i.e. all mode switch primitives are correctly communicated and the handling is correctly executed.

Assumptions 1 and 5 are simplifying assumptions that we plan to lift in the future. 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.

3.1 The handling of an emergency scenario

In order to make all components recognize an emergency scenario, we introduce an EMS (Emergency Mode Switch) primitive. When a component receives an ems\(k\), it will be aware of the emergency scenario \(k\). Once an emergency scenario is triggered, it should never be rejected and a mode switch must be performed in time. This simplifies its propagation in the sense that there is no need to check if all Type A components are ready to switch mode. 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 Emergency Mode Switch Propagation (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 \(c_i \in \mathcal{CC} \setminus \{\text{Top}\}\), it will send an ems\(k\) to \(P_{c_i}\) and \(\mathcal{SC}_A^{c_i}(k)\); (3) If \(c_i = \text{Top}\), 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 \(c_j \in \mathcal{CC} \setminus \{\text{Top}\}\), it propagates the ems\(k\) depending on the sender \(c_n\) and \(T^{c_j}_k\): 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 \(T^{c_j}_k = A\), 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 \(T^{c_j}_k = B\), 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 \(c_j = \text{Top}\), then \(c_j\) will propagate the ems\(k\) to \(\mathcal{SC}_{c_j}(k) \setminus \{c_n\}\), where \(c_n \in \mathcal{SC}_{c_j}\).

The EMSP protocol enforces each component to switch mode no matter whether it is ready or not. This may not be desirable if some components are not ready to switch mode after receiving an EMS. However, 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 an EMS, a Type A component will start its reconfiguration for the emergency mode switch, following the same mode switch dependency rule that is originally applied to a non-emergency mode switch. The EMSP protocol is demonstrated in Fig. 4, assuming the component hierarchy in Fig. 2 and that \(k\) is an emergency scenario. Comparing the two mode switch processes in figures 3 and 4, it is self-evident that the propagation of an emergency scenario is faster than that of a non-emergency scenario. An emergency mode switch is immediately triggered by the MSS. Unlike the other primitives, an EMS can be simultaneously propagated both to a parent and to a subcomponent, thus accelerating its propagation.

The EMSP protocol is implemented in Algorithm 1. Here \(k \leftarrow c_j\) means that scenario \(k\) or a primitive associated with \(k\) is from the immediate sender \(c_j\). Also,
Figure 4: Demonstration of the EMSP protocol

$C : Signal(c_i, A, B)$ and $Wait(c_i, A, B)$ are used for $c_i$ to send a primitive $B$ to $C$ or receive a primitive $B$ via the dedicated mode switch port $A$, which is either $p^{MSX}$ (for exchanging primitives with $P_{ci}$) or $p^{MSX}$ (for exchanging primitives with $SC_{ci}$).

**Algorithm 1** $EMSP(c_i, k)$

```plaintext
if $ems^k ← c_i$ then
  if $c_i ∈ PC$ then
    $Signal(c_i, p^{MSX}_{in}, ems^k)$;
  else
    $∀c_j ∈ SC^{A}_{ci}(k) : Signal(c_i, p^{MSX}_{in}, ems^k)$;
    if $c_i ≠ Top$ then
      $Signal(c_i, p^{MSX}_{in}, ems^k)$;
    end if
  end if
else if $ems^k ← P_{ci}$ then
  if $c_i ∈ PC$ then
    return ;
  else
    $∀c_j ∈ CC \{ Top \}$
    $∀c_j ∈ SC^{A}_{ci}(k) : Signal(c_i, p^{MSX}_{in}, ems^k)$;
  end if
else
  $∀c_j ∈ SC^{A}_{ci}(k) \{ c_i \} : Signal(c_i, p^{MSX}_{in}, ems^k)$;
  if $T_{c_i}^k = A ∧ c_i ≠ Top$ then
    $Signal(c_i, p^{MSX}_{in}, ems^k)$;
  end if
end if
```

An EMS may not be immediately handled after arriving at a component. For instance, if a component $c_i$ receives an EMS during its reconfiguration, the handling of the EMS will be delayed by the reconfiguration of $c_i$. 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 a component receives an EMS or triggers an EMS as the MSS, it will put the EMS in its EMS queue. As a component handles an EMS, it enters the Emergency Transition State (ETS):

**Definition 4.** A component $c_i$ is in the 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 $k$, i.e. when (1) $c_i$ has completed its mode switch for $k$ ($T_{c_i}^k = A$); or (2) $c_i$ has received an $msc^k$ from all $c_j ∈ SC^{A}_{ci}(k)$ ($T_{c_i}^k = B$).

Each component $c_i$ should remove the $ems^k$ from $c_i.Q_{ems}$ after $c_i$ has completed
handled \( k \). Hereafter we shall use Normal Transition State (NTS) to indicate a transition state based on a non-emergency scenario in contrast to ETS. When \( c_i \) is in either an NTS or ETS, it should not trigger a scenario. If \( c_i \) is not in an NTS or ETS but has triggered a non-emergency scenario which has not been completely handled by \( c_i \), then \( c_i \) may trigger an emergency scenario but should not trigger another non-emergency scenario.

Reading from the EMS queue of a component has higher priority than reading from its MSR and MSQ queues. Based on the definitions of EMS queue and ETS, the MSR/MSQ queue checking rule is replaced 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 queue, MSQ and 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 \( c_i.Q_{ems} = \emptyset \land c_i.Q_{msq} \neq \emptyset \), then \( x = c_i.Q_{msq}[1] \).
- If \( c_i.Q_{ems} = \emptyset \land c_i.Q_{msq} = \emptyset \land c_i.Q_{msr} \neq \emptyset \) and \( c_i.Q_{msr}[1] \) has not been propagated to \( P_c \), then \( x = c_i.Q_{msr}[1] \).

The pending scenario checking rule is implemented in Algorithm 2, where NTS and ETS are boolean variables. If NTS is true, then \( c_i \) is in an NTS. If ETS is true, then \( c_i \) is in an ETS. The boolean variable retained is set to true when \( c_i.Q_{msr}[1] \) has been propagated to \( P_c \). The functions HandleMSQ and HandleMSR handle an MSQ and an MSR respectively, their algorithms already provided in [4]. The algorithm for the function HandleEMS, which handles an EMS, will be provided in later sections.

![Algorithm 2](image)

According to Section 2.2, a component must check the validity of each MSR/MSQ in its MSR and MSQ queues and apply the MSR/MSQ queue updating rule when it is about to leave the NTS for \( k \) that has triggered a mode switch. The definition of the validity of MSR/MSQ remains the same while taking the emergency mode switch into account:

**Definition 6.** Valid/Invalid MSR/MSQ: Let \( c_i \) be a component which is about to leave the NTS or ETS by completing a mode switch for Scenario \( k \) (\( T^k_{ci} = A \)) or receiving an msc\( k \) from all \( c_j \in SC^A_c(k) \) (\( T^k_{ci} = B \)). If \( c_i.Q_{msr} \neq \emptyset \), then \( c_i \) will identify the validity of each msc\( k' \in c_i.Q_{msr} \) (\( k' \neq k \)):

- If \( c_i \in PC \), then \( c_i \) must be the MSS of \( k' \). Hence msc\( k' \) is invalid.
- If \( c_i \in CC \), then \( msr^{k'} \) comes from \( c_o \in SC_{c_i} \cup \{c_i\} \). If \( c_o = c_i \), then \( msr^{k'} \) is valid when \( T^{k}_{c_i} = B \) and invalid when \( T^{k}_{c_i} = A \). If \( c_o \in SC_{c_i} \), then \( msr^{k'} \) is invalid when one of the following two conditions is satisfied: (1) \( T^{k}_{c_o} = B \); (2) \( T^{k}_{c_o} = A \) and \( c_i \) has received an \( msc^k \) from \( c_o \). Otherwise, \( msr^{k'} \) is invalid.

If \( c_i = Top \), \( T^{k}_{c_i} = A \), and \( \exists msq^{k'} \in c_i.Q_{msq} \), then \( msq^{k'} \) is invalid. Otherwise, \( msq^{k'} \) is valid.

Based on Definition 6, the same MSR/MSQ queue updating rule is applied when \( c_i \) leaves the NTS/ETS for \( k \):

**Definition 7. The MSR/MSQ queue updating rule:** After \( c_i \) identifies the validity of each MSR/MSQ in its MSR and MSQ queues, 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 \( c_i.Q_{msq} \).

Algorithm 3 implements the MSR/MSQ queue updating rule which is not changed by the IHB approach. Here \( dequeue(A,B) \) is a function removing the primitive \( A \) from the corresponding MSR/MSQ queue. Likewise, the reverse operation is \( enqueue(A,B) \). The boolean variable \( valid^{k'} \) is set to true when \( c_i \) receives an \( msr^{k'} \) from \( c_j \in SC_{c_i} \) and \( c_j \) has already sent an \( msc^k \) to \( c_i \) when \( c_i \) is in an NTS or ETS for \( k \). After all the invalid MSR and MSQ primitives are removed, the function \( resetValidity(c_j) \) resets the validity of all the remaining MSR primitives in \( c_i.Q_{msr} \), i.e. by setting each \( valid^{k'} \) to false for each \( msr^{k'} \in c_i.Q_{msr} \). The only exception is: if \( \exists ems^{k'}_{c_j} \in c_i.Q_{ems} \), then \( c_i \) must be in the NTS for \( k \) and the \( msr^{k'} \) is still valid. The reason is that the \( msr^{k'} \) is still valid even after \( c_i \) completes the handling of the emergency scenario \( k'' \), as it is sent after the \( ems^{k''}_{c_j} \). If the validity of the \( msr^{k'} \) is reset as \( c_i \) leaves the NTS for \( k \), the \( msr^{k'} \) will be removed by the MSR/MSQ queue updating rule as \( c_i \) leaves the ETS for \( k'' \), though it is still valid. Stated in [4], Algorithm 3 is called by the functions \( HandleMSQ \) and \( HandleEMS \) in Algorithm 2, as a component leaves the NTS for \( k \) which has triggered a mode switch. Similarly, it is also called by the function \( HandleEMS \) as a component leaves an ETS.

### 3.2 Issues due to concurrent triggering of emergency and non-emergency scenarios

A component \( c_i \in CC \setminus \{Top\} \) may receive an EMS from either its parent or a subcomponent. An EMS from the parent is called a downstream EMS. Otherwise, if an EMS comes from \( c_i \) itself or a subcomponent, it is called an upstream EMS. 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 \( P_c \), with \( k_2 \) invalidating \( msr^{k_1} \).

Issue 1 is illustrated by Fig. 5(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 the pending scenario checking rule, \( b \) will handle the \( ems^{k_2} \) and switch to the new mode, making the \( msr^{k_1} \) previously sent to \( a \) invalid. Moreover, the \( msr^{k_1} \) may have been propagated further by \( a \) to its ancestors, and a corresponding \( msq^{k_1} \) may have been issued by the MSDM of \( k_1 \). It is the duty of \( b \) to let the other components abort the handling of \( k_1 \). Since we assume that the ongoing reconfiguration of a component cannot be aborted, \( k_1 \) can only be aborted before any component starts its reconfiguration for \( k_1 \).
Algorithm 3 UpdateQueue($c_i, k$)

\[
\begin{align*}
\text{if } c_i \in PC & \text{ then} \\
& \text{if } \exists msr_{c_i}^k \in c_i.Q_{msr} \text{ then} \\
& \quad \text{dequeue}(msr_{c_i}^k, c_i.Q_{msr}); \\
& \text{end if} \\
\text{else} \{c_i \in CC\} & \text{ if } (\exists msr_{c_j}^k \in c_i.Q_{msr}) \& \& (T_{c_i}^k = A) \text{ then} \\
& \quad \text{dequeue}(msr_{c_j}^k, c_i.Q_{msr}); \\
& \text{end if} \\
& \text{end if} \\
& \text{if } (c_i = \text{Top}) \& \& (T_{c_i}^k = A) \& \& (\exists msq_i^k \in c_i.Q_{msq}) \text{ then} \\
& \quad \text{dequeue}(msq_i^k, c_i.Q_{msq}); \\
& \text{if } (c_i \neq \text{Top}) \& \& (T_{c_i}^k = A) \text{ then} \\
& \quad \text{retained := false;}
\end{align*}
\]

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

Issue 2 is illustrated by Fig. 5(b) where $b$ receives an $msq^1$ from $a$ and an $ems^2$ 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 $msq^1$ in $b.Q_{msq}$ and put the $ems^2$ in $b.Q_{ems}$. According to the pending scenario checking rule, $b$ handles the $ems^2$ first. If $T_{b}^{k_2} = A$, $b$ will switch mode based on $k_2$. However, $a$ sends the $msq^1$ to $b$ assuming that $b$ is in its old mode. Therefore, $k_2$ makes the $msq^1$ invalid and $b$ should let all the involved components abort the handling of $k_1$ before it handles the $ems^2$.

**Issue 3:** When a composite component $c_i$ receives an EMS ($ems^2$) immediately after having propagated an MSQ ($msq^1$) to $SC^A_{c_i}(k_1)$ and before receiving all the replies, the handling of the $ems^2$ is blocked by $k_1$, which may even become invalid due to $k_2$.

Issue 3 is illustrated by Fig. 5(c). Component $b$ receives an $msq^1$ from its parent $a$ and then propagates the $msq^1$ to its subcomponents $c$ and $d$ at $t_0$. Meanwhile, $c$ has sent an $ems^2$ to $b$ before $c$ receives the $msq^1$. Since $b$ has entered the NTS for $k_1$ at $t_0$, it will complete the handling of $k_1$ before it can handle $ems^2$. The pending scenario checking rule protects an NTS from the interruption of an emergency scenario. The same problem may also occur when $b$ receives a downstream EMS. For instance, suppose $b$ immediately receives a downstream $ems^2$ from $a$ after having propagated an $msq^1$ to $c$ and $d$. Scenario $k_1$ is from $c$ and $b$ is the MSDM of $k_1$. The handling of $ems^2$ is also delayed due to $k_1$. Another even worse consequence is that $k_1$ may become invalid due to $k_2$. In Fig. 5(c), suppose $c$ is in mode $m_{c_1}^1$ before sending the $ems^2$ and switches to $m_{c_2}^2$ after the emergency mode switch. Component $b$ sends an $msq^1$ to $c$ assuming that the current mode of $c$ is $m_{c_1}^1$. Hence the $msq^1$ becomes invalid by the time $c$ sends the $ems^2$ to $b$. If $c$ handles the $msq^1$ while it is running in $m_{c_2}^2$, the system could be subject to unexpected anomalies.
3.3 Handling the identified issues

The issues pinpointed in Section 3.2 pose extra challenge to the handling of concurrent emergency and non-emergency scenarios.

Before we present solutions to these issues, let’s first observe the behavior of the MSS of an emergency scenario. Let $c_i$ be a component which has sent an $msr^{k_1}$ to $P_{c_i}$. Before $c_i$ receives an $msq^{k_1}$ from $P_{c_i}$, $c_i$ triggers an emergency scenario $k_2$. 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, $c_i$ is obliged to abort the handling of $k_1$ and notify all the other components which have received either an $msr^{k_1}$ or an $msq^{k_1}$. If the $msr^{k_1}$ comes from $c_j \in SC_{c_i}$, $c_i$ can make $c_j$ abort the handling of $k_1$ by sending an $msd^{k_1}$ to $c_j$. Similarly, we introduce an upstream MSA (Mode Switch Abort) primitive so that when $P_{c_i}$ receives an $msa^{k_1}$ from $c_i$, $P_{c_i}$ will abort the handling of $k_1$.

Algorithm 4 implements the behavior of an MSS when it triggers a scenario, where new Scenario is a boolean variable set to true when the MSS $c_i$ is about to trigger a new scenario, and Emergency is a boolean variable set to true if the scenario triggered by $c_i$ is emergency.

Upon receiving an MSA, a component can abort a scenario by applying the MSA handling rule:

**Definition 8. The MSA handling rule:** Let $c_i \in CC$ be a component that receives an $msa^k$ from $c_j \in SC_{c_i}$. Then there must be one or two MSR primitives from $c_j$ in $c_i.Q_{msr}$:

- If there is only 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 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 $c_i.Q_{msr}$.
Algorithm 4 \textit{MS\_detection}(c_i)

\textbf{loop}
\begin{itemize}
\item if \textit{new\_Scenario} then
\begin{itemize}
\item if \textit{Emergency} then
\begin{itemize}
\item if \(\exists\text{msr}^k \in c_i.Q_{\text{msr}} \land \text{retained}\) then
\begin{itemize}
\item Signal(c_i, \text{p}^{\text{MSX}}, \text{msq}^k);
\item retained := false;
\item dequeue(msr^k, c_i.Q_{\text{msr}});
\item if \(k' \leftarrow c_j \in SC_{c_i}\) then
\begin{itemize}
\item c_j : Signal(c_i, \text{p}^{\text{MSX}}, \text{msd}^k);
\end{itemize}
\end{itemize}
\end{itemize}
\item else{\textit{Emergency}}
\begin{itemize}
\item if \(c_i \neq \text{Top}\) then
\begin{itemize}
\item enqueue(msr^k_{c_i}, c_i.Q_{\text{msr}});
\end{itemize}
\item else\{\textit{c_i = Top}\}
\begin{itemize}
\item enqueue(msq^k, c_i.Q_{\text{msq}});
\end{itemize}
\end{itemize}
\end{itemize}
\end{itemize}
\item else
\begin{itemize}
\item enqueue(emsk^k, c_i.Q_{ems});
\end{itemize}
\end{itemize}
\textbf{end if}
\end{itemize}
\textbf{end loop}

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

In addition, if \(c_i\) has propagated an \(\text{msq}^k\) to \(SC^A_c(k)\) without receiving all the replies, then \(c_i\) will send an \(\text{msd}^k\) to \(SC^A_c(k) \setminus \{c_j\}\) and leave the NTS for \(k\).

The purpose of the rule above is to let all the components which have received either an \(\text{msr}^k\) or \(\text{msq}^k\) abort the handling of \(k\). One may wonder how \(c_i\) can have two \(\text{MSR}\) primitives from the same subcomponent in \(c_i.Q_{\text{msr}}\). Fig. 6 illustrates one possible scenario based on a simple example with a composite component \(b\) and its two subcomponents \(c\) and \(d\). At \(t_0\), \(b.Q_{\text{msr}}\) has already had an \(\text{msr}^k_{c_1}\) from \(c\) and finished the first phase of the propagation of \(k\), and in the second phase \(b\) starts to propagate an \(\text{msq}^k\) to \(c\) and \(d\). After that, \(b\), \(c\) and \(d\) start their reconfiguration. Since \(b\) has a long reconfiguration time, it is still in the NTS for \(k\) when it receives an \(\text{msr}^k_{c_1}\) from \(c\) and \(d\). Component \(c\) starts to run in its new mode after sending the \(\text{msr}^k_{b_1}\) to \(b\). Then \(c\) sends an \(\text{msr}^k_{b_2}\) to \(b\) in its new mode such that at \(t_3\), \(b.Q_{\text{msr}}\) has both \(\text{msr}^k_{c_1}\) and \(\text{msr}^k_{c_2}\) from \(c\). Shortly after that, \(c\) sends an \(\text{msd}^k_{b_2}\) to \(b\) due to an emergency scenario \(k'\) which makes \(\text{msr}^k_{c_2}\) invalid. Note that \(\text{msr}^k_{c_1}\) in \(c_i.Q_{\text{msr}}\) is still valid because it has already triggered a mode switch. Therefore, \(b\) should remove \(\text{msr}^k_{c_2}\) from \(b.Q_{\text{msr}}\) at \(t_4\) by Definition 8. As \(b\) completely handles \(k_1\) at \(t_5\), \(\text{msr}^k_{c_1}\) is removed from \(b.Q_{\text{msr}}\).

The MSA handling rule is demonstrated by Fig. 7. A composite component \(b\), with \(a\) as its parent and \(c\) and \(d\) as its subcomponents, receives an \(\text{msq}^k\) from \(c\) at \(t_0\), right after propagating an \(\text{msq}^k\) to \(c\) and \(d\). By Definition 8, \(b\) first removes the \(\text{msr}^k\) and \(\text{msq}^k\) from \(c_i.Q_{\text{msr}}\) and \(c_i.Q_{\text{msq}}\) respectively. Since \(b\) has sent an \(\text{msr}^k\) to \(a\), an \(\text{msq}^k\) is sent from \(b\) to \(a\), which will apply the same MSA handling rule. Moreover, since \(b\) has propagated an \(\text{msq}^k\) to \(c\) and \(d\) while \(d\) still does not know that the \(\text{msq}^k\) has become invalid, \(b\) also sends an \(\text{msd}^k\) to \(d\). Component \(c\) will ignore the \(\text{msq}^k\) from \(b\) after sending the \(\text{msq}^k\) to \(b\).

The MSA handling rule is implemented in Algorithm 5, as a composite component \(c_i\) receives an \(\text{msa}^k\) from \(c_j \in SC_{c_i}\), with the following additional notations:
Figure 6: A scenario enabling a component to have two MSR primitives from the same subcomponent

Figure 7: Demonstration of the MSA handling rule

- \textit{findMSRfrom}(c_i, c_j) is a function checking how many MSR primitives in \( c_i.Q_{msr} \) are from \( c_j \in SC_{c_i} \). The result is assigned to an integer \( \text{number} \). If \( \text{number} = 1 \), the MSR is identified as \( msr_{c_j}^{k_1} \). If \( \text{number} = 2 \), the first MSR is identified as \( msr_{c_j}^{k_2} \) while the second one is identified as \( msr_{c_j}^{k_3} \).

- \( TS_{c_i} \) is the ID of the scenario currently being handled by \( c_i \) either in an NTS or ETS.

- \( \text{wait4msok} \) is a boolean variable set to true when \( c_i \) has propagated an MSQ to its Type A subcomponents and is waiting for the reply.

The MSA handling rule only solves Issue 1 identified in Section 3.2. Concerning Issue 2, we propose a \textit{preliminary EMS handling rule} which is applied as an additional step before each component starts its propagation of an upstream EMS. This rule consists of two parts:

\textbf{Definition 9. The preliminary EMS handling rule (Part 1):} Let \( c_i \) be a component which is about to handle an upstream \( \text{ems}^{k_2} \). Let \( c_i.Q_{msr}[1] = msr_{c_i}^{k_1} \) (\( c_i \in SC_{c_i} \cup \{c_i\} \)) if \( c_i.Q_{msr} \neq \emptyset \). If \( c_i \neq \text{Top} \) and \( msr_{c_i}^{k_1} \) 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) \( T_{c_i}^{k_2} = A \); (2) \( T_{c_i}^{k_2} = B \) and \( T_{c_i}^{k_2} = A \). 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 Part 1 of the preliminary EMS handling rule is to check if \( c_i \) has sent an \( msr^{k_1} \) to \( P_{c_i} \) which becomes invalid due to an upstream \( \text{ems}^{k_2} \), where \( k_1 \) and \( k_2 \)
Algorithm 5 MSAHandling \((c_i \in CC, c_j, k)\)

\[
\text{number} := \text{findMSRfrom}(c_i, c_j);
\]

\[
\text{if number} = 1 \text{ then}
\quad \text{dequeue}(msr_{c_i}^{k_1}, c_i, Q_{msr});
\]

\[
\text{else} \{ \text{number} = 2 \}
\quad \text{if NTS && TS_{c_i} = k_1 \ then}
\quad \quad \text{dequeue}(msr_{c_j}^{k_1}, c_i, Q_{msr});
\]

\[
\text{else}
\quad \quad \text{dequeue}(msr_{c_i}^{k_1}, c_i, Q_{msr});
\]

\[
\text{dequeue}(msr_{c_j}^{k_1}, c_i, Q_{msr});
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{if c_i} \neq \text{Top then}
\quad msr_{c_i}^{k'} := c_i, Q_{msr}[1];
\]

\[
\text{if retained} \&\& c_i = c_j \&\& k' = k \text{ then}
\quad \text{Signal}(c_i, p_{MSX}^{MS}, msa^k);
\quad \text{retained} := \text{false};
\quad \text{if } \exists msq^k \in c_i, Q_{msq} \text{ then}
\quad \quad \text{dequeue}(msq^k, c_i, Q_{msq});
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{end if}
\]

\[
\text{if TS_{c_i} = k} \&\& \text{wait4msk} \text{ then}
\quad \forall c_l \in SC_{c_i}(k) \cup \{ c_j \} : \text{Signal}(c_i, p_{in}^{MS}, msd^k);
\]

\[
\text{end if}
\]

Part 2 of the preliminary EMS handling rule directly follows Part 1:

Definition 10. The preliminary EMS handling rule (Part 2): Let \(c_i\) be a component that has just finished Part 1 of the preliminary EMS handling rule based on an upstream emergency scenario \(k_2\), then,
• If \( c_i = \text{Top} \), 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 \( c_i \neq \text{Top} \), 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 \( c_i.Q_{msq} \).

Part 1 guarantees that no \( msq^k \) exits in \( c_i.Q_{msq} \) such that \( k \) comes from \( c_j \in SC_{c_i} \). The purpose of Part 2 is for \( c_i \) to abort the handling of an \( msq^{k_1} \) while \( k_1 \) comes from \( P_{c_i} \), if the \( msq^{k_1} \) exists in \( c_i.Q_{msq} \). In this case, \( c_i \) cannot send an \( msa^{k_1} \) to \( P_{c_i} \) in that \( msa^{k_1} \) 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 \( T^{k_2}_{c_i} = A \) (since the emergency mode switch of \( c_i \) makes the \( msq^{k_1} \) invalid).

The preliminary EMS handling rule is demonstrated by Fig. 8 with the same example and legend as in Fig. 7. In Fig. 8(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. 8(b) and (c), \( b \) receives an \( msq^{k_1} \) and an upstream \( ems^{k_2} \) at the same time. Following Part 2, \( b \) sends an \( msnok^{k_1} \) to \( a \) to abort the handling of \( k_1 \) when \( T^{k_2}_{b} = A \), shown in Fig. 8(b).

![Figure 8: Demonstration of the preliminary EMS handling rule](image)

The preliminary EMS handling rule (Part 2) is implemented in Algorithm 7. Integrating the preliminary EMS handling rule, the EMSP protocol, the mode switch dependency rule and the MSR/MSQ queue updating rule, Algorithm 8 implements the complete handling of an emergency scenario, with the following notations:

- \( \text{Reconfiguration}(c_i) \) is a function reconfiguring \( c_i \) based on the current scenario.
• CheckMSC($c_i$) is a function checking if $c_i$ has received an $msc^k$ from all $c_j \in SC_{c_i}^A(k)$ when $c_i \in CC$. If yes, the boolean variable $mscOK$ is set to true.

• resetMSC($c_i$) is a function setting each boolean variable $mscFrom_{c_j}$ to false, where $c_i \in CC$, $c_j \in SC_{c_i}^A(k)$, and $mscFrom_{c_j}$ is set to true when $c_i$ receives an $msc^k$ from $c_j$. This function is also called when $c_i$ leaves an NTS.

It should be noted that Algorithm 8 does not include the low level implementations such as starting or stopping the execution of $c_i$ and updating the mode of $c_i$.

Algorithm 7 pre EMSHandling 2($c_i, k_2$)

\[
\text{if } c_i = \text{Top} \text{ then} \\
\quad \text{if } \exists msq^{k_1} \in c_i.Q_{msq} \& \& T_{c_i}^{k_2} = A \text{ then} \\
\quad \quad \text{dequeue}(msq^{k_1}, c_i.Q_{msq}); \\
\quad \text{end if} \\
\text{else} \{c_i \neq \text{Top}\} \\
\quad \text{if } \exists msq^{k_1} \in c_i.Q_{msq} \& \& T_{c_i}^{k_2} = A \text{ then} \\
\quad \quad \text{Signal}(c_i, p^{MSX}, msnok^{k_1}); \\
\quad \quad \text{Wait}(c_i, p^{MSX}, \text{primitive}); \\
\quad \quad \text{if } \text{primitive} = \text{msd}^{k_1} \text{ then} \\
\quad \quad \quad \text{dequeue}(msq^{k_1}, c_i.Q_{msq}); \\
\quad \text{end if} \\
\text{end if} \\
\text{end if}
\]

Algorithm 8 HandleEMS($c_i, k$)

\[
ETS := \text{true}; \\
\text{if } k \leftarrow SC_c || k \leftarrow c_i \text{ then} \\
\quad \text{pre EMSHandling 1($c_i, k$); } \\
\quad \text{pre EMSHandling 2($c_i, k$); } \\
\text{end if} \\
\text{EMSP($c_i, k$); } \\
\text{Reconfiguration($c_i$); } \\
\text{if } c_i \in CC \& \& SC_{c_i}^A(k) \neq \emptyset \text{ then} \\
\quad \text{repeat} \\
\quad \quad \text{CheckMSC($c_i$); } \\
\quad \quad \text{until } mscOK \\
\text{end if} \\
\text{if } T_{c_i} = A \& \& c_i \neq \text{Top} \text{ then} \\
\quad \text{Signal}(c_i, p^{MSX}, msc^k); \\
\text{end if} \\
\text{dequeue}(ems^k, c_i.Q_{ems}); \\
\text{UpdateQueue($c_i, k$); } \\
\text{resetMSC($c_i$); } \\
ETS := \text{false};
\]

Issue 3 can be resolved by applying the following EMS receiving rule to each composite component:

**Definition 11. The EMS receiving rule:** Let $c_i \in CC$ be a component that propagates an $msq^{k_1}$ to $SC_{c_i}^A(k_1)$ and then receives an $ems^{k_2}$ from $c_j \in \{P_{c_i}\} \cup SC_{c_i}$ before $c_i$ receives all the expected $msok^{k_1}$ or $msnok^{k_1}$ from $SC_{c_i}^A(k_1)$. 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 $SC_{c_i}^A(k_1)$ and leaves the NTS for $k_1$. In addition, if $c_i \neq \text{Top}$, $T_{c_i} = A$ and $c_j \in SC_{c_i}$, $c_i$ will also send an $msa^{k_1}$ to $P_{c_i}$. 

16
This rule is applied right after a composite component puts an EMS in its EMS queue. Fig. 9 demonstrates the EMS receiving rule with the same example in Fig. 8. In Fig. 9(a), b receives a downstream $ems^k$ from a right after propagating an $msq^k$ to $SC_b^k(k_1)$ (i.e. c and d) as the MSDM of $k_1$. Since b has not received the reply from c and d, it can abort the handling of $k_1$ by propagating an $msd^k$ to c and d so that the handling of $ems^k$ is not delayed by $k_1$. In Fig. 9(b), b receives an upstream $ems^k$ from c right after propagating an $msg^k$ to $SC_b^k(k_1)$, whereas b is not the MSDM of $k_1$. Apart from propagating an $msd^k$ to c and d, Component b also sends an $msa^k$ to a.

![Figure 9: Demonstration of the EMS receiving rule](image)

Taking both the EMS receiving rule and the MSA handling rule into account, Algorithm 9 implements the behavior of a component $c_i$ when $c_i$ receives a new scenario or some specific primitives. The duty of the function $updateBothQueues(c_i, k)$ is: If $c_i.Q_{msq}[1] = msq^k$, then the $msq^k$ is removed from $c_i.Q_{msq}$; if $c_i.Q_{msr}[1] = msr^k$, then the $msr^k$ is removed from $c_i.Q_{msr}$. Besides, after $c_i$ removes the $msr^k$, if retained is true, $c_i$ will set it to false.

Now we have provided all the algorithms for the IHB approach. Fig. 10 summarizes the relationship between these algorithms. The mode switch behavior of a component $c_i$ can be represented by three separate tasks: algorithms 2, 9, and 4, respectively. Algorithm 4 is only required when $c_i$ is an MSS. Algorithm 9 implements the MSA handling rule by calling Algorithm 5 as well as the EMS receiving rule. The handling of an emergency scenario is implemented in Algorithm 8 that is called by Algorithm 2 which implements the pending scenario checking rule. Algorithm 8 calls algorithms 6 and 7 (the preliminary EMS handling rule), 1 (the EMSP protocol), and 3 (the MSR/MSQ queue updating rule).

### 3.4 Summary of the IHB approach

Integrating all the aforementioned protocols and rules into the mode switch runtime mechanism of MSL, our IHB approach is composed by the following elements:

1. The pending scenario checking rule
2. The MSP protocol
Algorithm 9 newScenario($c_i$)

loop
  Wait($c_i$, $p_{MSX}^A \lor p_{in}^{MSX}$, primitive);
  if primitive = ems$^k$ then
    enqueue(ems$^k$, $c_i$.Q$_{ems}$);
    if $TS_{c_i} = k'$ && wait4msok then
      $\forall c_i \in SC_{c_i}(k') : Signal(c_i, p_{in}^{MSX}, msd^{k'})$;
      if $c_i \neq Top$ && $TS_{c_i} = k'$ then
        $\forall c_i \in SC_{c_i} : Signal(c_i, p_{MSX}^A, msd^{k'})$;
        retained := false;
      end if
    end if
  end if
  else if primitive = msr$^k$cj && $c_j \in SC_{c_i} \cup \{c_i\}$ then
    if $c_j \in CC$ && $c_j \in SC_{c_i}$ then
      if $((NTS || ETS) && mscFrom_{cj})$ then
        valid$^k$ := true;
      end if
    end if
    enqueue(msr$^k$cj, $c_i$.Q$_{msr}$);
  end if
  else if primitive = msq$^k$ then
    enqueue(msq$^k$, $c_i$.Q$_{msq}$);
  else if primitive = msd$^k$ then
    if $c_j \in CC$ && $k \leftarrow c_j \in SC_{c_i}$ then
      $c_j : Signal(c_i, p_{ms}^{MSX}, msd^{k})$;
    end if
    updateBothQueues($c_i$, $k$);
  end if
  else if primitive = msa$^k$cj then
    mscFrom$_{cj} := true$;
  else {[primitive = msa$^k$ && k $\leftarrow c_j \in SC_{c_i}$]}
    MSAHandling($c_i$, $c_j$, $k$);
  end if
end loop
3. The EMSP protocol
4. The MSA handling rule
5. The preliminary EMS handling rule
6. The EMS receiving rule
7. The mode switch dependency rule
8. The MSR/MSQ queue updating rule

Compared with the previous mode switch runtime mechanism of MSL that does not consider emergency scenarios, IHB replaces the MSR/MSQ queue checking rule with the pending scenario checking rule. The EMSP protocol is particularly introduced for the propagation of an emergency scenario. The EMS receiving rule prevents the handling of an emergency scenario from being delayed by an ongoing non-emergency scenario, as long as the ongoing scenario has not triggered a mode switch. The MSA handling rule and the preliminary EMS handling rule are only used for the handling of an upstream emergency scenario. The mode switch dependency rule and the MSR/MSQ queue updating rule require no revision. The workflow of IHB for each component is depicted in Fig. 11, 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 task, i.e. Algorithm 9.

The additional computation overhead of IHB, which only counts when an emergency scenario is triggered, is rather minor. For each component, IHB is just about sending the right primitive to the right receivers (among its parent and subcomponents) and managing the MSR/MSQ/EMS queues. In [3], it states that the maximum size of \( c_i.Q_{msq} \) is 2 and the maximum size of \( c_i.Q_{msr} \) is \( 2 \times |SC_{c_i}| + 1 \). Additionally, the size of \( c_i.Q_{ems} \) is always no bigger than 1. Therefore, the computation overhead of IHB for each component scales with the number of subcomponents.

### 3.5 Improvement by IHB

We use an example to demonstrate how the mode switch handling of an emergency scenario is improved by IHB. Depicted in Fig. 12, 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 complete mode switch processes are compared in Fig. 12, one on the left and the other on the right. The only difference between the two processes is that $k_2$ triggered by $e$ is a non-emergency scenario for the left one while $k_2$ is an emergency scenario for the right one. Component $b$ receives an $msq^{k_0}$ from $a$ and an $msr^{k_1}$ from $d$ at the same time. Right after that, $b$ receives either an $msr^{k_2}$ or an $ems^{k_2}$ from $e$. If $k_2$ is a non-emergency scenario, its handling is delayed first by $k_0$ and then by $k_1$. In contrast, if $k_2$ is an emergency scenario, it is handled immediately. As $b$ receives the $ems^{k_2}$, it aborts the handling of $k_0$ by sending an $msd^{k_0}$ 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. Since $T_d^{k_2} = A$, $k_1$ is aborted by the MSR/MSQ queue updating rule as $b$ completes the handling of $k_2$. Apparently, IHB brings substantial improvement to the mode switch time of the emergency scenario $k_2$.

4 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: Upon triggering or receiving a scenario $k$, a component completes the handling of $k$ within bounded time.
We resort to model checking for the verification of IHB, using the model checker UPPAAL [5]. 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 verifying that both Property 1 (deadlock freeness) and Property 2 (completeness) hold for the model.

2. Proving that the UPPAAL model faithfully captures the relevant behavior of an arbitrary complex finite system of components.

4.1 Verification of the abstract model

Inspired by [6], 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 above the target component and a number of child stubs below the target component. Illustrated in Fig. 13, the modeled system consists of four components: a target component \( b \) together with a parent stub \( a \) and two child stubs \( c \) and \( d \). The target component \( b \) can receive non-emergency scenarios 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 \( c \). We do not consider the case when \( b \) triggers a scenario itself because this can be simulated by adding a virtual subcomponent of \( b \) (e.g. Child stub 3) which triggers the scenario instead.

The verification includes both the case when the target component is a non-top composite component (shown in Fig. 13) and the case when the target component is the top component or a primitive component. Shown in Fig. 14, let \( c_i \) be the target component and the verification includes three cases: (1) \( c_i = Top \); (2) \( c_i \in CC \setminus \{ Top \} \); and (3) \( c_i \in PC \). Case (2) has already been described together with Fig. 13. For Case (1), the system has no parent stub. Component \( c_i \) itself can trigger an \( msq^{k_0} \) whenever possible, and \( c_i \) may also receive an \( msr^{k_3} \) from Child stub 1 and an \( msr^{k_2} \) from Child stub 2 whenever possible. Moreover, depending on whether \( c_i \) can trigger an \( ems^{k_3} \), Case (1) can be divided into two sub-cases. If \( c_i \) cannot trigger an \( ems^{k_3} \), it may...
receive an $ems^{k_3}$ from Child stub 1. For Case (3), the system consists of only the target component $c_i$ and its parent stub, without child stubs. Component $c_i$ may receive an $msq^{k_0}$ from the parent stub and trigger an $msr^{k_1}$ whenever possible. Case (3) is also divided into two sub-cases depending on whether $c_i$ can trigger an $ems^{k_3}$. If $c_i$ cannot trigger an $ems^{k_3}$, it may receive an $ems^{k_3}$ from the parent stub. Among these cases, Case (2) yields the most complex UPPAAL model. To reduce the verification time, an additional assumption is made for Case (2) with upstream $ems^{k_3}$ that the emergency scenario $ems^{k_3}$ is triggered only once. 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 each component to complete the corresponding emergency mode switch. Then each triggering of the $ems^{k_3}$ is independent and it is sufficient to trigger the $ems^{k_3}$ only once in the model for Case (2). Based on this assumption and Fig. 13, Case (2) with upstream $ems^{k_3}$ can be divided into four sub-cases: (2a) $T^{k_3}_c = T^{k_3}_d = T^{k_3}_b = A$; (2b) $T^{k_3}_c = T^{k_3}_d = A, T^{k_3}_b = B$; (2c) $T^{k_3}_c = A, T^{k_3}_d = T^{k_3}_b = B$; (2d) $T^{k_3}_c = T^{k_3}_d = A, T^{k_3}_b = B$. Note that $c$ is always a Type A component for $k_3$ as the $ems^{k_3}$ comes from $c$.

Let’s briefly explain our UPPAAL model based on the most representative case, Case (2) with an upstream $ems^{k_3}$, illustrated in Fig. 12 in Section 3. The emergency scenario is triggered by the child stub $c$. IHB is implemented on the target component $b$ with two separate models, NewScenario and HandleScenario, illustrated in figures 15 and 16, respectively. NewScenario has three major tasks: (1) the handling of an MSD; (2) putting an MSR/MSQ in the MSR/MSQ queue; and (3) propagating an MSA to the parent if necessary by applying either the MSA handling rule or the EMS receiving rule. The rest of IHB is implemented in HandleScenario. Moreover, the models of the parent stub and each child stub are illustrated in figures 17 and 18, respectively. Both child stubs share the same model, yet distinguished by different parameters. To reduce model complexity, certain parts of the behaviors of $b$ and its child stubs irrelevant to the handling of an emergency scenario are abstracted. For instance, the MSQ from $b$ to its child stubs and the reply MS0K or MSN0K are hidden in the model. The MSC from a child stub is also hidden for the same reason.
When $b$ is not involved in any mode switch, its two models, NewScenario and HandleScenario, are in their initial states, i.e. the states with a double circle. The handling of an MSQ and MSR is started as the transitions 6 and 1 of HandleScenario are fired, respectively. The handling of an emergency scenario is realized by the transitions 37-48 of HandleScenario, started by Transition 37. Transitions 1, 6, 37 have their own guards which comply with the pending scenario checking rule. Since $b$ receives the upstream $ems^{k_3}$ from $c$, the preliminary EMS handling rule must be applied before $b$ can propagate the $ems^{k_3}$. Part 1 of the preliminary EMS handling rule is implemented...
by the transitions 37–39 of HandleScenario. The function MSAorNot() associated with Transition 37 determines if \( b \) should send an msa\(^{k_3} \) to \( a \). If yes, Transition 39 is fired. Otherwise, Transition 38 is fired. Part 2 of the preliminary EMS handling rule is implemented by the transitions 40–42 of HandleScenario. If \( \exists \)msg\(^{k_0} \in b.Q_{msg} \) and \( T^A_{b} = A \), then \( b \) should send an msnok\(^{k_0} \) to \( a \) to abort the handling of \( k_0 \). This corresponds to Transition 41. Transitions 43–46 implement the EMSP protocol. If \( T^A_{d} = A \), \( b \) should propagate the ems\(^{k_3} \) to \( d \) by Transition 44. Otherwise, Transition 43 is taken instead. If \( T^A_{d} = A \), \( b \) should also propagate the ems\(^{k_3} \) to \( a \) by Transition 46. Otherwise, \( b \) is the MSDM for \( k_3 \) and Transition 45 is selected.

When \( b \) receives an msa\(^{k_1} \) from \( c \), the MSA handling rule is applied by taking Transition 6 of NewScenario. If \( b \) needs to propagate the msa\(^{k_1} \) further to \( a \), it will take Transition 7. In addition, if \( b \) has propagated an msg\(^{k_1} \) to \( \mathcal{SC}^A_b (k_1) \), it will take Transition 8 of NewScenario which is synchronized with Transition 22 of HandleScenario. If \( T^A_{d} = A \), \( b \) will send an msd\(^{k_1} \) to \( d \) by Transition 27 of HandleScenario. When \( b \) receives an ems\(^{k_2} \) from \( c \), the EMS receiving rule is applied by taking Transition 12 of NewScenario. If \( b \) needs to send an MSA to \( a \) due to the ems\(^{k_3} \), it will take Transition 15 of NewScenario. If \( b \) needs to send an MSD to its Type A subcomponents, Transition
17 of NewScenario will be taken, synchronized with Transition 23 of HandleScenario.

We have mentioned that IHB is expected to satisfy the deadlock freeness and completeness properties. The verification of the properties was initially carried on a MacBook Pro, with 2.66GHz Intel Core 2 Duo CPU and 8GB 1067 MHz DDR3 memory. Among the cases classified in Fig. 14, the modeling and verification of Case (3) requires the least effort. In Case (3), the model only consists of two components: the parent stub and the target primitive component. Case (3) is divided into two sub-cases: Case (3a) where an EMS (ems\textsuperscript{k2}) is propagated from a, and Case (3b) where b can trigger an EMS (ems\textsuperscript{k2}) itself. In Case (3a), the following five properties, formulated in the UPPAAL query language which is a subset of Timed Computation Tree Logic (TCTL), were verified:

- P1: A\[\] not deadlock
- P2: ParentStub.EMSpending->!ParentStub.EMSpending
- P3: (ParentStub.Wait4OK && ParentStub.localSID==k0)->ParentStub.Init
- P4: pending->!pending
- P5: ! HandleScenario.Idle-> HandleScenario.Idle

Here P1 is equivalent to Property 1 while P2-P5 jointly represent Property 2. The notation “A->B” denotes that if the expression A is true, then the expression B is eventually true. P2 means that if the parent stub a sends an EMS to b, then a eventually completes the handling of the EMS. Similarly, P4 means that if the target component b triggers a scenario by sending an MSR to a, then b eventually completes the handling of the MSR. P5 means that HandleScenario can always return to its initial state Idle from any other state. In other words, once b starts to handle a scenario, the handling can eventually be completed. All properties except P4 were all successfully verified. In P4, “pending” is a boolean variable when b triggers an msr\textsuperscript{k1}. This result does not reflect any error of IHB or our model. On the contrary, it is expected because a can send an msq\textsuperscript{k0} or an ems\textsuperscript{k2} to b whenever possible. If a keeps sending the msq\textsuperscript{k0} or ems\textsuperscript{k2} to b, the msr\textsuperscript{k1} triggered by b will never be handled. We allow a to send the msq\textsuperscript{k0} to b unlimited number of times, however, Property 2 is only guaranteed when the constant arrival of the msq\textsuperscript{k0} from a to b is bounded. Therefore, we duplicated our model into two versions. In the second version, we slightly changed the behavior of the parent stub such that for every two consecutive MSQ primitives\textsuperscript{1} (msq\textsuperscript{k} and msq\textsuperscript{k'}) from the parent stub, at least either k or k' is from the target component b. This does not alter the nature of IHB, yet satisfying Property 2. Actually, it is only theoretically possible that ci keeps receiving scenarios from Pci without breaks. In practice, the interval between the arrival of two consecutive scenarios for each component should mostly be very large since mode switch should not be a frequent event.

The verification result for Case (3a) is summarized in Table 1, where “x” means that a property is not satisfied, Case (3a)-I means that the emergency scenario k2 can be triggered by a as many times as possible and k0 can be continuously sent by a. In contrast, Case (3a)-II means that the behavior of a is changed to satisfy P4.

In Case (3b), the same five properties as in Case (3a) were verified except that the expression of P2 was changed to “EMSpending->!EMSpending", as the emergency scenario is triggered by b itself. If an msq\textsuperscript{k0} can be continuously sent from a(Case (3b)-I), P4 will not be satisfied. However, the parent stub a can be changed in the same way

\textsuperscript{1}More generally, for every n consecutive MSQ primitives from the parent stub, there must exist an msq\textsuperscript{k} such that k originates from b. Here n = 2 for simplifying reasons.
Table 1: Verification result for Case (3a)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case (3a)-I</td>
<td>0.004s</td>
<td>0.004s</td>
<td>0.004s</td>
<td>x</td>
<td>0.005s</td>
</tr>
<tr>
<td>Case (3a)-II</td>
<td>0.005s</td>
<td>0.005s</td>
<td>0.004s</td>
<td>0.007s</td>
<td>0.006s</td>
</tr>
</tbody>
</table>

as for Case (3a) (this corresponds to Case (3b)-II). The verification result for Case (3b) is summarized in Table 2.

Table 2: Verification result for Case (3b)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case (3b)-I</td>
<td>0.012s</td>
<td>0.017s</td>
<td>0.012s</td>
<td>x</td>
<td>0.018s</td>
</tr>
<tr>
<td>Case (3b)-II</td>
<td>0.013s</td>
<td>0.018s</td>
<td>0.013s</td>
<td>0.019s</td>
<td>0.015s</td>
</tr>
</tbody>
</table>

In Case (1), the model consists of three components: the target component \(a\) (the top component), and two child stubs \(b\) and \(c\). Similar to Case (3), Case (1) can also be divided into two sub-cases: Case (1a) where an emergency scenario comes from \(b\), and Case (1b) where an emergency scenario is triggered by \(a\) itself. In Case (1a), the following six properties were verified:

- P1: A[] not deadlock
- P2: EMSBuffering→!EMSBuffering
- P3: pending→!pending
- P4: ChildStub1.MSRpending→!ChildStub1.MSRpending
- P5: ChildStub2.MSRpending→!ChildStub2.MSRpending
- P6: ! HandleScenario.Idle→ HandleScenario.Idle

Here P2-P5 represent Property 2 for the \(ems^{k_3}\) from \(b\), the \(msq^{k_0}\) from \(a\), the \(msr^{k_1}\) from \(b\), and the \(msr^{k_2}\) from \(c\), respectively. P6 also belongs to Property 2, stating that once \(a\) starts the handling of a scenario, it eventually completes its handling. Similar to Case (3), Case (1a) is divided into two sub-cases: (1) Case(1a)-I where \(k_0\) can be continuously triggered; and (2) Case (1a)-II where \(k_0\) cannot be continuously triggered. The verification result is summarized in Table 3, which indicates that P4 and P5 are only satisfied in Case (1a)-II.

Table 3: Verification result for Case (1a)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case (1a)-I</td>
<td>65.279s</td>
<td>59.089s</td>
<td>82.211s</td>
<td>x</td>
<td>x</td>
<td>78.329s</td>
</tr>
<tr>
<td>Case (1a)-II</td>
<td>41.843s</td>
<td>37.409s</td>
<td>44.97s</td>
<td>52.07s</td>
<td>52.941s</td>
<td>48.786s</td>
</tr>
</tbody>
</table>

In Case (1b), the properties to be verified are the same as in Case (1a). The only difference is the expression of P2, which becomes “\(Epending→!Epending\)” yet with the same meaning, since the emergency scenario is triggered by \(a\) itself. To satisfy P4 and P5, additional constraints were imposed in Case (1b)-II to the triggering conditions of
Table 4: Verification result for Case (1b)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case (1b)-I</td>
<td>12.502s</td>
<td>14.107s</td>
<td>15.303s</td>
<td>x</td>
<td>x</td>
<td>13.419s</td>
</tr>
<tr>
<td>Case (1b)-II</td>
<td>11.36s</td>
<td>9.874s</td>
<td>12.205s</td>
<td>15.87s</td>
<td>16.226s</td>
<td>13.949s</td>
</tr>
</tbody>
</table>

Table 5: Verification result for Case (2) with downstream \( ems^{k_3} \)

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case (2)-I</td>
<td>157.104s</td>
<td>116.218s</td>
<td>163.714s</td>
<td>x</td>
<td>x</td>
<td>177.712s</td>
</tr>
<tr>
<td>Case (2)-II</td>
<td>101.084s</td>
<td>77.534s</td>
<td>80.982s</td>
<td>165.411s</td>
<td>172.899s</td>
<td>113.003s</td>
</tr>
</tbody>
</table>

The verification for Case (2) with upstream \( ems^{k_3} \) turned out to be rather computationally expensive. This forced us to run the verification on a remote Linux server (16 cores, 0.8GHz, 32GB memory). Currently the UPPAAL verifier only utilizes a single processor, i.e. only 1 of the 16 cores, however, the 32GB memory plays a vital role in the verification. Since we assume that the emergency scenario can be triggered at most once for Case (2), we have mentioned that Case (2) with upstream \( ems^{k_3} \) can be divided into four sub-cases depending on whether \( b \) and \( d \) are Type A components for \( k_3 \). We realized that the verification time of P6 (\(! \text{HandleScenario.Idle} \rightarrow \text{HandleScenario.Idle}\)) was extremely long and it required more memory than the other properties. For that reason, we replaced P6 with the following three properties:

- P6: MSQRhandler.Temp \( \rightarrow \) MSQRhandler.Idle
- P7: MSQRhandler.MSQHandling \( \rightarrow \) MSQRhandler.Idle
• P8: MSQRhandler.Emergency1 -> MSQRhandler.Idle

Here states **Temp, MSQHandling** and **Emergency1** are the first states that **HandleScenario** goes to upon receiving an **MSR, MSQ** and **EMS**, respectively. Therefore, P6-P8 jointly imply that once the target component \( b \) starts to handle a scenario, it will eventually complete the handling. The verification result for Case (2) with upstream \( \text{ems}^{k_3} \) is summarized in tables 6 and 7.

<table>
<thead>
<tr>
<th>Case (2a)</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1610.59s</td>
<td>1380.39s</td>
<td>2062.09s</td>
<td>x</td>
<td>x</td>
<td>1331.94s</td>
<td>1702.09s</td>
<td>1001.28s</td>
</tr>
<tr>
<td>Case (2b)</td>
<td>1591.15s</td>
<td>1181.97s</td>
<td>1860.92s</td>
<td>x</td>
<td>x</td>
<td>1318.98s</td>
<td>1597.92s</td>
<td>1003.13s</td>
</tr>
<tr>
<td>Case (2c)</td>
<td>1092.49s</td>
<td>908.74s</td>
<td>1296.28s</td>
<td>x</td>
<td>x</td>
<td>842.49s</td>
<td>1057.77s</td>
<td>703.36s</td>
</tr>
<tr>
<td>Case (2d)</td>
<td>1625.76s</td>
<td>1232.66s</td>
<td>1899.94s</td>
<td>x</td>
<td>x</td>
<td>1424.86s</td>
<td>1557.01s</td>
<td>1076.08s</td>
</tr>
</tbody>
</table>

Table 6: Verification result for Case (2) with upstream \( \text{ems}^{k_3} \) (continuous triggering of \( k_0 \))

<table>
<thead>
<tr>
<th>Case (2a)</th>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
<th>P6</th>
<th>P7</th>
<th>P8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1038.3s</td>
<td>881.86s</td>
<td>1054.85s</td>
<td>1329.36s</td>
<td>1337.93s</td>
<td>921.01s</td>
<td>937.1s</td>
<td>674.47s</td>
</tr>
<tr>
<td>Case (2b)</td>
<td>989.6s</td>
<td>827.74s</td>
<td>1051.36s</td>
<td>1236.87s</td>
<td>1426.03s</td>
<td>853.47s</td>
<td>908.47s</td>
<td>637.38s</td>
</tr>
<tr>
<td>Case (2c)</td>
<td>683.23s</td>
<td>628.18s</td>
<td>697.21s</td>
<td>901.62s</td>
<td>988.36s</td>
<td>556.34s</td>
<td>595.23s</td>
<td>448.7s</td>
</tr>
<tr>
<td>Case (2d)</td>
<td>1117.25s</td>
<td>808.4s</td>
<td>1108.25s</td>
<td>1387.71s</td>
<td>1621.11s</td>
<td>952.63s</td>
<td>950.43s</td>
<td>711.17s</td>
</tr>
</tbody>
</table>

Table 7: Verification result for Case (2) with upstream \( \text{ems}^{k_3} \) (no continuous triggering of \( k_0 \))

According to the verification result in tables 6 and 7, the maximum verification time is 2062.09s originating from P3 in Case (2a) (with continuous triggering of \( k_0 \)). Besides, the maximum memory usage is around 25GB originating from P3 in Case (2d) (with continuous triggering of \( k_0 \)).

4.2 Generalization to an arbitrary complex finite system of components

In order to prove that the verification results based on the abstract UPPAAL model are valid for any CBMMSs, we need to prove that our UPPAAL model faithfully represents an arbitrary complex finite system of components with respect to the execution of IHB.

We first define the **external/internal mode switch behavior** of a component:

**Definition 12. External mode switch behavior:** The external mode switch behavior, or simply external behavior of a component \( c_i \), is the visible behavior of \( c_i \) from the perspective of \( P_{c_i} \) during a mode switch. The external behavior of \( c_i \) is represented by (1) the response of \( c_i \) after \( P_{c_i} \) sends a primitive to \( c_i \); and (2) the capability of \( c_i \) to actively send any primitive to \( P_{c_i} \) and the precondition for sending the primitive.

**Definition 13. Internal mode switch behavior:** The internal mode switch behavior, or simply internal behavior of a composite component \( c_i \), is the visible behavior of \( c_i \) from the perspective of each \( c_j \in SC_{c_i} \). The internal behavior of \( c_i \) is represented by (1) the response of \( c_i \) after \( c_j \) sends a primitive to \( c_i \); and (2) the capability of \( c_i \) to actively send any primitive to \( c_j \) and the precondition for sending the primitive.
Obviously, both the external and internal behaviors of a component are dependent on the mode switch runtime mechanism of MSL. Our IHB approach, which belongs to the mode switch runtime mechanism, specifies 9 primitives in total, among which 3 can only be transmitted as a downstream primitive, including MSQ, MSI and MSD; other 5 can only be transmitted as an upstream primitive, including MSOK, MSNOK, MSC, MSR, and MSA; and additionally an EMS can be transmitted both upstream and downstream. The external and internal behaviors of a component are limited to the transmission of these primitives. Note that Top has no visible external mode switch behavior while a primitive component has no visible internal mode switch behavior.

Furthermore, each component is associated with a composition level and a depth level:

**Definition 14. Composition level:** Each component \( c_i \) is associated with a composition level denoted as \( l_{c_i} \). If \( c_i \in P \mathcal{C} \), then \( c_i \) has a composition level 0, denoted as \( l_{c_i} = 0 \). If \( c_i \in \mathcal{C} \), \( l_{c_i} = \{ \max \{ l_{c_j} \} + 1 \mid c_j \in \mathcal{S}c_{c_i} \} \).

**Definition 15. Depth level:** Each component \( c_i \) is associated with a depth level denoted as \( d_{c_i} \). \( d_{\text{Top}} = 0 \), and for each \( c_p \) and \( c_q = P c_p \), \( d_{c_p} = d_{c_q} + 1 \).

Based on the definitions above, we interpret the three assertions (A1-A3) stated in Section 4.2 of the report as follows:

- **A1:** The internal behavior of the parent stub of our UPPAAL model faithfully represents the internal behavior of any composite component with arbitrary depth level.
- **A2:** The external behavior of each child stub of our UPPAAL model faithfully represents the external behavior of any component with arbitrary composition level.
- **A3:** For each composite component, two child stubs in our UPPAAL model faithfully represent an arbitrary number of child stubs.

In our UPPAAL model, the parent stub is modeled as Top while each child stub is modeled as a composite component with composition level 1 (hereafter we shall call it a CL1 component). Hence A1 and A2 can be re-interpreted as A1b and A2b respectively:

- **A1b:** The internal behavior of Top faithfully represents the internal behavior of any composite component with arbitrary depth level.
- **A2b:** The external behavior of a CL1 component faithfully represents the external behavior of any component with arbitrary composition level.

In order to prove A1b, we must compare the internal behavior of Top with that of any composite component in general. We first prove the internal behavior equivalence between Top and a composite component with depth level 1. By Definition 13, the comparison is based on two criteria: (1) the response upon receiving a primitive from a subcomponent; and (2) the capability of actively sending any primitive to a subcomponent as well as the precondition for sending the primitive. According to IHB, a composite component may receive an \( m_{sr}^k \), \( m_{sa}^k \), \( m_{sok}^k \), \( m_{snok}^k \), \( m_{sc}^k \), or \( e_{ms}^k \) from a subcomponent, and may actively send an \( m_{sq}^k \) or \( e_{ms}^k \) to a subcomponent. We must analyze the transmission of all these primitives to prove the following lemma:

**Lemma 1.** Let \( c_i \) be a composite component with \( c_p = P c_i \) and \( c_p = \text{Top} \). Then the internal behavior of \( c_p \) is equivalent to that of \( c_i \).
Proof. We first enumerate all the possible primitives that can be sent from \(c_i\) to \(c_p\), analyze all the possible responses of \(c_p\) and compare them with the responses of \(c_i\) upon receiving all possible primitives from each \(c_j \in SC_{c_i}\):

1. An \(msr^k\) from \(c_i\) to \(c_p\): Since \(c_p = Top\), \(c_p\) is always the MSDM for \(k\). According to the MSP protocol, \(c_p\) is supposed to either propagate an \(msg^k\) to \(SC_{c_p}(i)\) (including \(c_i\)) if it is ready to switch mode or send an \(msd^k\) to \(c_i\) if it is not ready. In addition, \(c_p\) may also send an \(msg^{k_1}\) (\(k_1 \neq k\)) to \(c_i\) instead. For instance, \(c_p\) may receive an \(msr^{k_1}\) from another subcomponent before it receives the \(msr^k\) from \(c_i\). If the concurrent triggering of an emergency scenario is also considered, \(c_p\) may also send an \(ems^{k_2}\) (\(k_2 \neq k\)) to \(c_i\). Similarly, after \(c_i\) receives an \(msr^k\) from \(c_j \in SC_{c_i}\), all the possible responses of \(c_i\) are also limited to an \(msq^k\), \(msg^{k_1}\), \(msd^k\), or \(ems^{k_2}\). \(c_i\) may or may not be the MSDM for \(k\), however, this makes no difference for \(c_j\).

2. An \(msa^k\) from \(c_i\) to \(c_p\) or from \(c_j \in SC_{c_i}\) to \(c_i\): The MSA handling rule implies that there is no need to send any primitive to the \(msa^k\) sender in either case.

3. An \(msok^k\) from \(c_i\) to \(c_p\): Following the MSP protocol, \(c_p\) will send either an \(msi^k\) or an \(msd^k\) to \(c_i\). As a special case, \(c_p\) may receive an \(ems^{k_1}\) from a subcomponent other than \(c_i\). Then according to the EMS receiving rule, \(c_p\) will abort the handling of \(k\) by sending an \(msd^k\) to \(SC_{c_p}(i)\) (including \(c_i\)). Likewise, after \(c_i\) receives an \(msok^k\) from \(c_j \in SC_{c_i}\), the response of \(c_i\) will be either an \(msi^k\) or \(msd^k\). In case that either \(c_p\) or \(c_i\) applies the EMS receiving rule to abort the handling of \(k\) due to the concurrent triggering of an emergency scenario, \(c_i\) will send an \(msd^k\) to \(SC_{c_i}(k)\) (including \(c_j\)).

4. An \(msnok^k\) from \(c_i\) to \(c_p\): Since an \(msnok^k\) suffices to abort the handling of \(k\), \(c_p\) will send an \(msd^k\) to \(c_i\). Even if \(c_p\) applies the EMS receiving rule due to an \(ems^{k_1}\) from a subcomponent other than \(c_i\), \(c_p\) will still send an \(msd^k\) to \(c_i\). Similarly, if an \(msnok^k\) is sent from \(c_j \in SC_{c_i}\) to \(c_i\), no matter which among \(c_i\) and \(c_p\) is the MSDM for \(k\), \(c_i\) will always send an \(msd^k\) to \(c_j\) as the response.

5. An \(msc^k\) from \(c_i\) to \(c_p\) or from \(c_j \in SC_{c_i}\) to \(c_i\): The mode switch dependency rule suggests that neither \(c_p\) nor \(c_i\) is expected to respond to the \(msc^k\) sender.

6. An \(ems^k\) from \(c_i\) to \(c_p\) or from \(c_j \in SC_{c_i}\) to \(c_i\): The EMSP protocol suggests that neither \(c_p\) nor \(c_i\) is expected to respond to the \(ems^k\) sender.

The enumeration above summarizes the responses of \(c_p\) and \(c_i\) to all possible primitives from a subcomponent in Table 8.

<table>
<thead>
<tr>
<th>Primitive from a subcomponent</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>(msr^k)</td>
<td>(msg^k), (msg^{k_1}), (msd^k), or (ems^{k_2})</td>
</tr>
<tr>
<td>(msa^k)</td>
<td>no response</td>
</tr>
<tr>
<td>(msok^k)</td>
<td>(msi^k) or (msd^k)</td>
</tr>
<tr>
<td>(msnok^k)</td>
<td>(msd^k)</td>
</tr>
<tr>
<td>(msc^k)</td>
<td>no response</td>
</tr>
<tr>
<td>(ems^{k_2})</td>
<td>no response</td>
</tr>
</tbody>
</table>

Table 8: The response of \(c_i (d_{c_i} < 2)\) to a primitive from a subcomponent

In addition, both \(c_p\) and \(c_i\) can actively send the following primitives to a subcomponent:
1. An msq\textsuperscript{k} (possibly followed by an msd\textsuperscript{k}). When \( c_i \) is not in any NTS/ETS for \( k \), \( c_p \) may actively send an msq\textsuperscript{k} to \( c_i \), while \( k \) is triggered by \( c_p \) itself or \( c_r \in \mathcal{SC}_{c_p} \setminus \{c_i\} \). In case that \( c_p \) applies the EMS receiving rule due to an upstream ems\textsuperscript{k}, \( c_p \) may subsequently send an msd\textsuperscript{k} to \( c_i \) to abort the handling of \( k \). Similarly, \( c_i \) may actively send an msq\textsuperscript{k} to each \( c_j \in \mathcal{SC}_{c_i} \) when \( c_j \) is not in any NTS/ETS for \( k \), with \( k \) being triggered by \( c_p, c_r \in \mathcal{SC}_{c_p} \setminus \{c_i\}, c_i \), or \( c_s \in \mathcal{SC}_{c_i} \setminus \{c_j\} \). In case that either \( c_p \) or \( c_i \) applies the EMS receiving rule due to an upstream ems\textsuperscript{k}, \( c_i \) may subsequently send an msd\textsuperscript{k} to \( c_j \) to abort the handling of \( k \).

2. An ems\textsuperscript{k} from \( c_p \) to \( c_i \) or from \( c_i \) to \( c_j \in \mathcal{SC}_{c_i} \). The precondition for the propagation of an ems\textsuperscript{k} from \( c_p \) to \( c_i \) is satisfied when \( c_i, Q_{ems} = \emptyset \). At one level down, the propagation of an ems\textsuperscript{k} from \( c_i \) to \( c_j \in \mathcal{SC}_{c_i} \) is satisfied when \( c_j, Q_{ems} = \emptyset \).

Therefore, \( c_p \) and \( c_i \) can actively send the same primitives, e.g. an msq\textsuperscript{k} (possibly with a subsequent msd\textsuperscript{k}) or an ems\textsuperscript{k}, to a subcomponent, with the same preconditions. Lemma 1 follows.

*Lemma 1* further implies:

**Theorem 1.** For each composite component \( c_i \) with \( c_p = P_{c_i} \). The internal behavior of \( c_i \) is equivalent to that of Top.

*Proof.* Theorem 1 can be proven by mathematical induction based on \( d_{c_p} \):

**Basis:** \( d_{c_p} = 0 \), i.e. \( c_p = \text{Top} \). By Lemma 1, the internal behaviors of \( c_p \) and \( c_i \) are equivalent. Theorem 1 follows.

**Inductive step:** Suppose \( d_{c_p} = n \ (n \in \mathbb{N}) \) and the internal behavior of \( c_p \) is equivalent to that of \( \text{Top} \). Then we need to prove that the internal behavior of \( c_i \) with \( d_{c_i} = n + 1 \) is also equivalent to that of \( \text{Top} \). Since the internal behaviors of \( c_p \) and \( \text{Top} \) are equivalent, it is OK to replace \( c_p \) with \( \text{Top} \) without changing the internal behavior of \( c_i \). Once \( c_p \) is replaced with \( \text{Top} \), \( c_i \) will become a component with depth level 1. By Lemma 1, the internal behaviors of \( \text{Top} \) and \( c_i \) will be equivalent. Since such a replacement does not change the internal behavior of \( c_i \), Theorem 1 also follows when \( d_{c_i} = n + 1 \).

Since the internal behaviors of \( \text{Top} \) and any composite component are equivalent, Theorem 1 satisfies A1b. In our UPPAAL model, the parent stub is modeled as \( \text{Top} \) whose internal behavior faithfully represents the internal behavior of any composite component, thus satisfying A1 as well.

The proof structure of A2b resembles the proof structure of A1b. The first step is to compare the external behaviors of a primitive component and a CL1 component. By Definition 12, the comparison is based on two criteria: (1) the response upon receiving a primitive from the parent; and (2) the capability of actively sending any primitive to the parent as well as the precondition for sending the primitive. According to our analysis result, the external behaviors of a CL1 component and a primitive component are equivalent with respect to the response to a primitive from the parent. Yet a CL1 component has a more relaxed precondition for sending an msd\textsuperscript{k} to the parent. Therefore, the external behavior of a CL1 component faithfully represents the external behavior of a primitive component but not the other way round. Then the correctness of A2b can be formulated by the following theorem:

**Theorem 2.** For each composite component \( c_i \) with \( c_p = P_{c_i} \) and \( l_{c_i} = n \ (n \in \mathbb{N}, n > 1) \). The external behavior of \( c_i \) is equivalent to that of a CL1 component.

*Proof.* We again resort to mathematical induction for the proof, based on \( n \).


**Basis:** If \( n = 2 \), then for each \( c_j \in SC_{c_i} \), \( c_j \) is either a primitive component \( (l_{c_j} = 0) \) or a CL1 component \( (l_{c_j} = 1) \). If all \( c_j \in SC_{c_i} \) behaves as a primitive component by restricting the preconditions for sending an \( msa^k \) to \( c_i \), then the external behavior of \( c_i \) is apparently equivalent to the external behavior of a CL1 component. However, if some subcomponents of \( c_i \) are CL1 components, there will be more circumstances where \( c_i \) receives an \( msa^k \) from a subcomponent. If \( c_i \) and a CL1 component have the same preconditions for sending an \( msa^k \) to the parent, the external behavior equivalence between \( c_i \) and a CL1 component will be guaranteed. According to IHB, a composite component \( c_r \) sends an \( msa^k \) to its parent \( c_s \) under the following four conditions: (1) \( c_r \) decides to trigger an \( ems^{k'} \) which makes \( k \) invalid; (2) \( c_r \) applies the MSA handling rule upon receiving an \( msa^k \) from a subcomponent; (3) \( c_r \) applies the preliminary EMS handling rule (Part 1) due to an upstream \( ems^{k'} \) from a subcomponent; (4) \( c_r \) applies the EMS receiving rule. Taking all these conditions into account, we derive that the preconditions for the propagation of an \( msa^k \) from \( c_r \) to \( c_s \) are: (1) \( \exists msr_{c_j}^k \in c_s.Q_{msr} \); and (2) \( c_r.Q_{ems} = \emptyset \). This implies that \( c_i \) and any \( c_j \in SC_{c_i} \) with \( l_{c_j} = 1 \) have the same preconditions for actively sending an \( msa^k \) to the parent. Hence Theorem 2 follows for the base case.

**Inductive step:** Assume the external behavior of \( c_j \) with \( l_{c_j} \leq n \) is equivalent to that of a CL1 component. Then we need to prove that \( c_i = P_{c_j} \) with \( l_{c_i} \leq n + 1 \) also has the same external behavior as that of a CL1 component. By virtual of the external behavior equivalence between \( c_j \) with \( l_{c_j} \leq n \) and a CL1 component, it is OK to replace all \( c_j \in SC_{c_i} \) with a CL1 component without changing the external behavior of \( c_i \). Consequently, \( c_i \) will become a component with composition level 2. Since we in the base case have proven that \( c_i \) with \( l_{c_i} = 2 \) have the same external behavior as that of a CL1 component, and the replacement does not change the external behavior of \( c_i \), the external behavior of \( c_i \) with \( l_{c_i} \leq n + 1 \) is also equivalent to that of a CL1 component, thus completing the proof.

In our UPPAAL model, each child stub is modeled as a CL1 component whose external behavior faithfully represents the external behavior of any composite component. Therefore, A2 is also satisfied. However, Theorem 2 is only valid if the external behavior of a CL1 component faithfully represents the external behavior of a primitive component in the following manner:

- The external behaviors of a CL1 component and a primitive component are equivalent with respect to the response to a primitive from the parent.

- A CL1 component has a more relaxed precondition for sending an \( msa^k \) to the parent than a primitive component.

The second bullet is obvious because only a composite component can apply the preliminary EMS handling rule, the MSA handling rule and the EMS receiving rule, among which the latter two enable the component to send an \( msa^k \) to its parent even when this component is in an NTS. In contrast, a primitive component can never send an \( msa^k \) when it is in an NTS. According to IHB, a component may receive an \( msq^k \), \( msi^k \), \( msd^k \), or \( ems^k \) from its parent. Among these primitives, an \( msd^k \) does not expect any reply, whereas the responses to the other primitives deserve further analysis. Therefore, it is necessary to compare the external behaviors of a CL1 component and a primitive component in response to an \( ems^k \), \( msi^k \), or \( msq^k \) from the parent, which corresponds to lemmas 2, 3 and 4.

**Lemma 2.** Let \( c_i \) be a CL1 component with \( c_j \in SC_{c_i} \cup \{c_i\} \). Then upon receiving an \( ems^k \) from \( P_{c_j} \) at \( t_1 \), \( c_j \) will send an \( msc^k \) to \( P_{c_j} \) at \( t_2 > t_1 \). No other primitives are sent between \( P_{c_j} \) and \( c_j \) within the interval \([t_1, t_2]\).
Lemma 3. Let \(c_j \in SC_{c_i}\), then since \(l_{c_i} = 1\), \(c_j \in PC\). According to the mode switch dependency rule, upon receiving an \(ems^k\) from \(c_i\), \(c_j\) will immediately reconfigure itself and send an \(msc^k\) to \(c_i\) after its reconfiguration. Therefore, Lemma 2 holds when \(c_j \in SC_{c_i}\).

At a higher composition level, suppose \(c_i\) receives an \(ems^k\) from \(c_P = P_{c_i}\). First we ignore the impact from a concurrently triggered non-emergency scenario (since there is only one emergency scenario for a system, \(k\) implies that no other emergency scenario is concurrently triggered). If \(SC_{c_i}(k) = \emptyset\), applying the mode switch dependency rule, \(c_i\) will immediately reconfigure itself and send an \(msc^k\) to \(c_i\) after its reconfiguration. If \(SC_{c_i}(k) \neq \emptyset\), applying the EMSP protocol, \(c_i\) will propagate the \(ems^k\) to \(SC_{c_i}(k)\) and start its reconfiguration. Since the reconfiguration times of \(c_i\) and \(SC_{c_i}(k)\) are all bounded, \(c_i\) will complete the mode switch for \(k\) and send an \(msc^k\) to \(P_{c_i}\) within bounded time.

The only case where the handling of \(ems^k\) by \(c_i\) is affected by a concurrently triggered scenario is that \(c_i\) has propagated an \(msi^{k'}\) to \(SC_{c_i}(k')\) and is waiting for the \(msc^{k'}\) from \(SC_{c_i}(k')\) when \(c_i\) receives the \(ems^k\) from \(c_p\). In this case, \(c_i\) must be the MSDM for \(k'\) and must first leave the NTS for \(k'\) by having received an \(msc^{k'}\) from all \(c_r \in SC_{c_i}(k')\). Since all \(c_r \in SC_{c_i}(k')\) have bounded reconfiguration times, \(c_i\) can eventually (i.e. within bounded time) leave the NTS for \(k'\). For \(c_i\), the only impact of \(k'\) on \(k\) is the bounded delay of the propagation of the \(ems^k\). However, this does not change the fact that \(c_i\) will send an \(msc^k\) to \(c_p\) at \(t_2\) and no other primitives are sent between \(c_p\) and \(c_i\) within the interval \([t_1, t_2]\), where \(t_1\) is the time when \(c_p\) sends the \(ems^k\) to \(c_i\) and \(t_2\) is the time when \(c_i\) sends the \(msc^k\) to \(c_p\).

Therefore, Lemma 2 also holds when \(c_j = c_i\) and this completes the proof. \(\square\)

**Lemma 3.** Let \(c_j\) be a CL1 component with \(c_j \in SC_{c_i} \cup \{c_i\}\). Then upon receiving an \(msi^k\) from \(P_{c_j}\) at \(t_1\), \(c_j\) will send an \(msc^k\) to \(P_{c_j}\) at \(t > t_1\). If \(c_j \in SC_{c_i}\), no other primitives are sent between \(P_{c_j}\) and \(c_j\) within the interval \([t_1, t_2]\). If \(c_j = c_i\), there is a chance that \(c_j\) sends an \(msa^{k'}\) to \(P_{c_j}\) (\(k' \neq k\)) within the interval \([t_1, t_2]\).

**Proof.** If \(c_j \in SC_{c_i}\), then since \(l_{c_i} = 1\), \(c_j \in PC\). According to the mode switch dependency rule, upon receiving an \(msi^k\) from \(c_i\), \(c_j\) will immediately reconfigure itself and send an \(msc^k\) to \(c_i\) after its reconfiguration. Therefore, Lemma 3 holds when \(c_j \in SC_{c_i}\).

At a higher level, suppose \(c_i\) receives an \(msi^k\) from \(c_p = P_{c_i}\). First we ignore the impact from a concurrently triggered scenario. If \(SC_{c_i}(k) = \emptyset\), applying the mode switch dependency rule, \(c_i\) will immediately reconfigure itself and send an \(msc^k\) to \(c_i\) after its reconfiguration. If \(SC_{c_i}(k) \neq \emptyset\), applying the MSP protocol, \(c_i\) will propagate the \(msi^k\) to \(SC_{c_i}(k)\) and start its reconfiguration. Since the reconfiguration times of \(c_i\) and \(SC_{c_i}(k)\) are all bounded, \(c_i\) will eventually complete the mode switch for \(k\) and send an \(msc^k\) to \(c_p\).

Since the MSDM has already issued the \(msi^k\), the handling of the \(msi^k\) by \(c_i\) will not be delayed by any concurrently triggered non-emergency scenario. However, there is one case where a concurrent emergency scenario triggered by \(c_s \in SC_{c_i}\) can make \(c_i\) send an \(msa^{k_1}\) (\(k_1 \neq k\)) to \(c_p\) before \(c_i\) sends the \(msc^k\) to \(c_p\). Suppose \(c_s\) has sent an \(msi^{k_1}\) to \(c_i\) which has propagated the \(msr^{k_1}\) to \(c_p\), and \(k\) is triggered by \(c_p\). Also, \(T_{c_s} = B\). Then \(c_s\) may trigger an emergency scenario at any time after \(c_i\) propagates the \(msi^{k_1}\) to \(SC_{c_i}(k)\). Hence \(c_i\) may first send an \(msa^{k_1}\) to \(c_s\) to abort the handling of \(k_1\) during the reconfiguration of \(c_i\) for \(k\). Since \(c_i\) immediately applies the MSA handling rule even during its reconfiguration for \(k\), \(c_i\) may propagate the \(msa^{k_1}\) to \(c_p\) before it sends the \(msc^k\) to \(c_p\).

Therefore, Lemma 3 also holds when \(c_j = c_i\) and this completes the proof. \(\square\)
Lemma 4. Let $c_i$ be a CL1 component with $c_j \in SC_{c_i} \cup \{c_i\}$. If $c_j \in SC_{c_i}$, upon receiving an $msq^k$ from $P_{c_j}$ at $t_1$, $c_j$ will send an $msok^k$ or $msnok^k$ to $P_{c_j}$ at $t_2 > t_1$. If $c_j$ receives an $msd^k$ from $P_{c_j}$ within the interval $[t_1, t_2]$, then $c_j$ will skip sending the reply to $P_{c_j}$. In addition, if $c_j = c_i$, upon receiving an $msq^k$ from $P_{c_j}$ at $t_1$, $c_j$ may also send an $msa^k$ or $msq^{k'} (k' \neq k)$ to $P_{c_j}$ instead of an $msok^k$ or $msnok^k$ at $t_2$.

Proof. If $c_j \in SC_{c_i}$, then since $l_{c_i} = 1$, $c_j \in PC$. According to the MSP protocol, after receiving the $msq^k$ from $c_i$, $c_j$ must send an $msok^k$ or $msnok^k$ to $c_i$ as soon as it knows if its current state allows the mode switch. Since $c_j$ immediately enters the NTS for $k$ after receiving the $msq^k$ from $c_i$, it cannot send any other primitive before replying with an $msok^k$ or $msnok^k$ to $c_i$. However, the concurrent triggering of an emergency scenario may request $c_j$ to send an $msd^k$ to $SC^A_{c_i}(k)$ (including $c_j$) to abort the handling of $k$. For instance, if $c_i$ receives an $emsk^k$ from $c_s \in SC_{c_i} \setminus \{c_j\}$ right after propagating the $msq^k$ to $SC^A_{c_i}(k)$. Then applying the EMS receiving rule, $c_i$ must send an $msd^k$ to $SC^A_{c_i}(k)$, thus making $c_j$ skip sending the reply to $c_i$. Therefore, Lemma 4 holds when $c_j \in SC_{c_i}$.

At a higher composition level, suppose $c_i$ receives an $msq^k$ from $c_p = P_{c_i}$. If no other scenarios are considered, $c_i$ is expected to send (1) an $msok^k$ to $c_p$ immediately if $c_i$ is ready to switch mode and $SC^A_{c_i}(k) = \emptyset$, or (2) an $msnok^k$ to $c_p$ immediately if $c_i$ is not ready to switch mode. Otherwise, if $c_i$ is ready to switch mode and $SC^A_{c_i}(k) \neq \emptyset$, it will immediately propagate the $msq^k$ to $SC^A_{c_i}(k)$. Since Lemma 4 holds for all $c_r \in SC^A_{c_i}(k)$, all $c_r$ will eventually send an $msok^k$ or $msnok^k$ back to $c_i$. As $c_i$ receives all the replies from $SC^A_{c_i}(k)$, it will also eventually send an $msok^k$ or $msnok^k$ to $c_p$. In addition, we enumerate all the possible cases where the response of $c_i$ is affected by another concurrently triggered Scenario $k'$:

1. $c_i$ propagates the $msq^k$ to $SC^A_{c_i}(k)$ and receives an $msa^k$ or $msa^{k'} (k' \neq k)$ from $c_s$ before $c_i$ receives all the replies from $SC^A_{c_i}(k)$. By following the MSA handling rule, $c_i$ should immediately propagate the $msa^k$ or $msa^{k'}$ to $c_p$.

2. After $c_i$ propagates the $msq^k$ to $SC^A_{c_i}(k)$, $c_j$ receives an $emsk^k$ from $c_s \in SC_{c_i}$. By following the EMS receiving rule, $c_i$ will send an $msa^k$ to $c_p$.

3. After $c_i$ receives the $msq^k$ from $c_p$, before $c_i$ replies to $c_p$, $c_i$ receives an $msd^k$ from $c_p$, e.g. because $c_p$ applies the EMS receiving rule. Then $c_i$ skips sending the reply to $c_p$.

4. Before $c_i$ is about to handle the $msq^k$ from $c_p$, $c_i$ receives an $emsk^k$ from $c_s \in SC_{c_i}$. By following the preliminary EMS handling rule (Part 2), $c_i$ will send an $msnok^k$ to $c_p$ to abort the handling of $k$.

5. When $c_i$ receives the $msq^k$ from $c_p$, $c_i$ may have propagated an $msq^{k'}$ to $SC^A_{c_i}(k')$ as the MSDM for $k'$, with $k'$ being triggered by $c_s \in SC_{c_i}$. Since all $c_r \in SC^A_{c_i}(k')$ are primitive components, $c_i$ can receive the replies from all $c_r \in SC^A_{c_i}(k')$ within bounded time. If all replies are $msok^{k'}$, $c_i$ will immediately trigger the mode switch for $k'$ by propagating an $msi^{k'}$ to $SC^A_{c_i}(k')$. Since all $c_j \in SC^A_{c_i}(k')$ have bounded reconfiguration times, applying the mode switch dependency rule, $c_i$ can eventually receive an $msa^{k'}$ from all $c_j \in SC^A_{c_i}(k')$. If $c_i$ receives at least one $msnok^{k'}$, it will immediately abort the handling of $k'$ by propagating an $msd^{k'}$ to $SC^A_{c_i}(k')$. In case that $c_i$ receives an MSA or EMS from a subcomponent, $k'$ will be aborted by the MSA handling rule or the EMS receiving rule. Hence $c_i$ can eventually leaves the NTS for $k'$. After that, according to the pending scenario
checking rule, $c_i$ will immediately handle the $msq^k$. Therefore, $k'$ only delays the propagation of the $msq^k$ to $SC_i(k)$ in bounded time. The reply from $c_i$ to $c_p$ still conforms to Lemma 4.

Therefore, without the influence of a concurrently triggered scenario, $c_i$ can eventually send an $msok^k$ or $msnok^k$ to $c_p$ after it receives an $msq^k$ from $c_p$. Indicated by Case (5), a concurrently triggered non-emergency scenario only potentially delays the reply of $c_i$. If a concurrently triggered emergency scenario is considered, the reply of $c_i$ can be an $msa^k$ or $msa^{k'}$ (cases (1) and (2)) or an $msnok^k$ (Case(4)). Moreover, indicated by Case (3) which is similar to the case when $c_s \in SC_i$, $c_i$ may receive an $msd^k$ from $c_p$ and skip its reply. Since all the cases above conform to Lemma 4, Lemma 4 also holds when $c_j = c_i$ and this completes the proof. \qed

The essentials of lemmas 2, 3, and 4 are summarized in Table 9 that compares the responses of $c_j$ ($l_{c_j} = 0$) and $c_i$ ($l_{c_i} = 1$) to a primitive from the parent. It is self-evident that the response of $c_i$ is more general than the response of $c_j$.

<table>
<thead>
<tr>
<th>Primitive from the parent</th>
<th>The response of $c_j$</th>
<th>The response of $c_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ems^k$</td>
<td>$msek$</td>
<td>$msek$</td>
</tr>
<tr>
<td>$msi^k$</td>
<td>$msek$</td>
<td>$msek$, $msnok^k$</td>
</tr>
<tr>
<td>$msq^k$</td>
<td>$msok^k$ or $msnok^k$</td>
<td>$msok^k$, $msnok^k$, $msa^k$, or $msa^{k'}$</td>
</tr>
<tr>
<td>$msd^k$</td>
<td>No response</td>
<td>No response</td>
</tr>
</tbody>
</table>

Table 9: The responses of $c_j$ ($l_{c_j} = 0$) and $c_i$ ($l_{c_i} = 1$) to a primitive from the parent

Lemmas 2, 3 and 4 jointly imply Lemma 5, which fundamentally supports Theorem 2.

**Lemma 5.** Consider a composite component $c_i$ with two subcomponents: $c_j \in PC$ and $c_r \in CC$ ($l_{c_r} = 1$). For $c_i$, the external behavior of $c_r$ can faithfully represent the external behavior of $c_j$, but not the other way round.

**Proof.** Lemmas 2, 3, and 4 jointly indicate that when $c_i$ sends an $msq^k$, $msi^k$ or $ems^k$ to $c_j$ and $c_r$, the responses of $c_j$ and $c_r$ are the same except that $c_r$ can also send an $msa^k$ to $c_i$ in response to an $msq^k$ or $msi^k$ from $c_i$. Therefore, the response of $c_r$ is more general than the response of $c_j$.

Furthermore, both $c_j$ and $c_r$ can actively send an $msr^k$, $msa^k$ or $ems^k$ to $c_i$. Apparently, $c_j$ and $c_r$ have the same preconditions for sending an $msr^k$ to $c_i$. Either $c_j$ or $c_r$ can send an $msr^k$ to $c_i$ if it is not in any NTS/ETS and it has not sent another $msr^{k'}$ to $c_i$, without completing the handling of $k'$ yet. Likewise, $c_j$ and $c_r$ also have the same preconditions for sending an $ems^k$ to $c_i$, i.e. when they are not in any NTS or the EMS queue is empty. As a primitive component, $c_j$ has the following preconditions for sending an $msa^k$ to $c_i$: (1) $\exists msr^k_j \in c_i.Q_{msr};$ (2) $c_j.Q_{ems} = \emptyset;$ and (3) $c_j$ is not in any NTS. In contrast, $c_r$ have more relaxed preconditions for sending an $msa^k$ to $c_i$ in the sense that $c_r$ can even be in an NTS while sending the $msa^k$, due to the MSA handling rule and the EMS receiving rule applied to a composite component when an emergency scenario is concurrently triggered.

The comparison above signifies that $c_r$ has more relaxed preconditions for sending an $msa^k$ to $c_i$. Therefore, from the perspective of $c_j$, the external behavior of $c_r$ faithfully represents the external behavior of $c_j$, but not the other way round. \qed

Finally, A3 is addressed by Theorem 3:
Theorem 3. Each composite component \( c_i \) with \( l_i = n \) \((n \in \mathbb{N})\) can correctly execute IHB regardless of the number of its subcomponents.

Proof. Theorem 3 can also be proven by mathematical induction based on our UPPAAL verification results.

**Basis:** Suppose \( c_i \) has two subcomponents. Theorem 3 directly follows from our UPPAAL verification results.

**Inductive step:** Suppose \( c_i \) can correctly execute IHB with \( n \) subcomponents, i.e. \( S\mathcal{C}_{c_i} = \{c_j^0, c_j^1, \ldots, c_j^{n-1}\} \) \((n \in \mathbb{N}, n \geq 3)\). Then the goal is to prove that it also works if another \( c_j^n \) is added to \( S\mathcal{C}_{c_i} \).

Since we assume that only one emergency scenario is specified for each system and the triggering of this emergency scenario has already been considered before \( c_j^n \) is added to \( S\mathcal{C}_{c_i} \), \( c_j^n \) can only send non-emergency scenarios to \( c_i \). Each \( k \) from \( c_j^n \) is propagated as an \( msr^k \) to \( c_i \), which then puts the \( msr^k \) in \( c_i.Q_{msr} \) in the same way as all the other MSR primitives sent by the other subcomponents of \( c_i \). If \( k \) is directly rejected by the MSDM, \( c_j^n \) will eventually receive a \( msd^k \) from \( c_i \). This is guaranteed because \( c_i \) can correctly execute IHB without \( c_j^n \), and the internal behavior of \( c_i \) is the same for any subcomponent. Actually, \( c_i \) needs to know that \( c_j^n \) is the sender of \( k \) only for sending the \( msd^k \) to \( c_j^n \) when \( k \) is directly rejected by the MSDM. Otherwise, \( c_i \) only needs to know \( SC_{c_i}^A(k') \) for each interaction with \( SC_{c_i} \) based on any emergency or non-emergency scenario \( k' \). If \( T_{c_j^k}^k = B \), the interaction between \( c_i \) and \( SC_{c_i} \) will remain the same before or after \( c_j^n \) is added to \( SC_{c_i} \). If \( T_{c_j^k}^k = A \), whenever \( c_i \) propagates an \( msd^k, msr^k, msd^k \), or \( ems^k \) to \( SC_{c_i}^A(k') \), \( c_j^n \) will be included in \( SC_{c_i}^A(k') \) and receive the primitive. Since \( c_j^n \) and any other \( c_r \in SC_{c_i}^A(k') \) (no emergency scenario comes from \( c_r \)) have the same external behavior, \( c_j^n \) can eventually send an \( msr^k \) or \( msnok^k \) to \( c_i \) upon receiving an \( msq^k \) from \( c_i \), and can eventually send an \( msd^k \) to \( c_i \) upon receiving an \( msd^k \) or \( ems^k \) from \( c_i \). Therefore, \( c_i \) can still correctly execute IHB after \( c_j^n \) is added to \( SC_{c_i} \).

Combining the basis and the inductive step, the proof of Theorem 3 is completed. \( \square \)

Since A1, A2, and A3 are all proven, our UPPAAL model structure indeed faithfully represents an arbitrary complex finite system of components.

5 Additional improvement of IHB

The IHB approach can be optimized and improved in many different ways. First, the MSP protocol can be further revised to facilitate the propagation of a non-emergency scenario. When a component \( c_i \) sends an \( msnok^k \) to \( P_{c_i} \), it knows that \( k \) will not be triggered. Hence there is no need for \( c_i \) to send such an \( msnok^k \) and then wait for the expected \( msd^k \). Instead, \( c_i \) can directly send an \( msa^k \) to \( P_{c_i} \) (and an \( msd^k \) to \( SC_{c_i}^A(k) \) if an \( msq^k \) has been propagated to \( SC_{c_i}^A(k) \)). Let \( C_M \) be the set of components between the MSS \( c_i \) and the MSDM \( c_j \) such that for each \( c_l \in C_M, c_l \) is an ancestor of \( c_i \) but a descendant of \( c_j \). Also, let \( S_{c_o} \) denote that the current state of component \( c_o \) allows a mode switch for the associated scenario \( k \). Then we get the improved MSP protocol:

**Definition 16. The improved Mode Switch Propagation (MSP) protocol:** When an MSS \( c_i \) detects a mode switch event, it will request to switch mode by triggering a scenario \( k \). If \( c_i \neq Top, c_i \) will issue an \( msr^k \) which is sent to \( P_{c_i} \), eventually reaching the MSDM \( c_j \) through \( C_M \). For each \( c_o \in C_M \), identified when \( T_{c_o}^k = A \) and \( S_{c_o} \) upon receiving the \( msr^k \), \( c_o \) forwards the \( msr^k \) to \( P_{c_o} \). Upon receiving the \( msr^k \), the MSDM \( c_j \) is identified if one of the following conditions is satisfied: (1) \( T_{c_j}^k = B \); (2) \( T_{c_j}^k = A \) and \( \neg S_{c_j} \); (3) \( T_{c_j}^k = A \) and \( S_{c_j} \) and \( c_j = Top \). The MSDM \( c_j \) makes the following decisions:
• In Condition (2), \( c_j \) will reject the \( msr^k \) by issuing an \( msd^k \) that is propagated back to \( c_i \) via \( C_M \).

• In conditions (1) and (3), \( c_j \) will approve the \( msr^k \) by issuing an \( msq^k \) that is propagated downstream and stepwise to all Type A components. After receiving the \( msq^k \), a component \( c_i \) will check its current state. When \( c_i \in PC \), if \( S_{c_i} \), \( c_i \) will reply to \( P_{c_i} \) with an \( msok^k \); if \( ¬S_{c_i} \), \( c_i \) will propagate the \( msq^k \) to \( SC_{c_i}^A(k) \). Then if all \( c_n \in SC_{c_i}^A(k) \) reply with an \( msok^k \), \( c_i \) will send an \( msok^k \) to \( P_{c_i} \). When \( c_i \in CC \), if \( S_{c_i} \), \( c_i \) will propagate the \( msq^k \) to \( SC_{c_i}^A(k) \). If \( c_i \) receives an \( msq^k \) from \( c_n \in SC_{c_i}^A(k) \) after propagating the \( msq^k \) to \( SC_{c_i}^A(k) \), then \( c_i \) will propagate an \( msq^k \) to \( P_{c_i} \) and an \( msd^k \) to \( SC_{c_i}^A(k) \) \( \setminus \{ c_n \} \). If \( c_i \) receives an \( msd^k \) from \( P_{c_i} \), it will propagate an \( msd^k \) to \( SC_{c_i}^A(k) \).

• If all \( c_a \in SC_{c_i}^A(k) \) have replied with an \( msok^k \), \( c_j \) will trigger a mode switch by issuing an \( msq^k \) that follows the propagation trace of the \( msq^k \). Mode switch propagation is completed when all Type A components have received the \( msr^k \). Otherwise, if \( c_j \) receives an \( msq^k \) from \( c_r \in SC_{c_i}^A(k) \), \( c_j \) will propagate an \( msd^k \) to \( SC_{c_j}^A(k) \) \( \setminus \{ c_r \} \).

If \( c_i = Top \), then \( c_j = c_i \) and \( c_i \) triggers \( k \) by directly sending an \( msq^k \) to \( SC_{c_i}^A(k) \).

Based on the improved MSP protocol, Part 2 of the preliminary EMS handling rule can be revised so that the \( msnok^k \) sent by a component \( c_i \) to \( P_{c_i} \) can be replaced with an \( msak^l \). Then there will be no need for \( c_i \) to wait for an \( msd^k \) from \( P_{c_i} \) and the handling of \( k \) can be aborted faster.

Moreover, the EMSP protocol can also be adapted to the concurrent triggering of the same emergency scenario by multiple components. Sometimes one may allow an emergency scenario to be simultaneously triggered by different components in order to accelerate its propagation. Then a component could receive an \( EMS \) with the same scenario ID from different senders before it is ready to handle any \( EMS \). An efficient and reasonable strategy for each component is to only put the first \( EMS \) in its EMS queue. It is sufficient to only remember the identity of the senders of subsequent incoming \( EMS \) primitives (with the same scenario ID) without buffering them. When the component begins to propagate the \( EMS \), all the immediate \( EMS \) senders remembered by this component will be excluded from the set of \( EMS \) receivers. For instance, in Figure 5(a), \( b \) can receive an \( ems^k \) first from \( a \) and then from \( c \) before it handles the \( ems^k \). Then there will be no need for \( b \) to propagate the \( ems^k \) to \( a \) and \( c \). Algorithm 10 implements the EMSP protocol while the concurrent triggering of the same emergency scenario is taken into account. In Algorithm 1, \( ems^k \) is the \( EMS \) in \( c_i.Q_{ems} \) to be handled. However, \( c_i \) may subsequently receive the same \( ems^k \) from another component \( c_n \) before \( c_i \) handles the \( ems^k \). Here we use \( k \leftarrow c_n \) to indicate that \( c_i \) has not received an \( ems^k \) from \( c_n \), hence there is no need to propagate the \( ems^k \) to \( c_n \).

6 Related work

In extended MECHATRONIC UML (EUML) [6] by Heinzemann et al., component reconfiguration can be propagated and executed at different hierarchical levels. Mode is not defined in EUML, which instead allows reconfiguration rules to 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.
Algorithm 10 $EMSP^2 (c_i, k)$

if $ems^k \leftarrow c_i$ then
    if $c_i \in \mathcal{PC}$ \&\& $k \leftarrow P_{c_i}$ then
        $Signal(c_i, p^MSX, ems^k)$;
    else if $c_i \neq Top$ then
        if $k \leftarrow P_{c_i}$ then
            $Signal(c_i, p^MSX, ems^k)$;
        end if
        \forall c_j \in \mathcal{SC}_A(c_i)(k) \&\& k \leftarrow c_j : Signal(c_i, p^MSX, ems^k);
    end if
else $ems^k \leftarrow P_{c_i}$ then
    if $c_i \in \mathcal{PC}$ then
        return ;
    else \{ $c_i \in \mathcal{CC} \setminus \{ Top \}$ \}
        $Signal(c_i, p^MSX, ems^k)$;
    end if
else $ems^k \leftarrow c_i \in \mathcal{SC}_A(c_i)$ \}
    $Signal(c_i, p^MSX, ems^k)$;
    if $T_{c_i} = A \&\& c_i \neq Top \&\& k \leftarrow P_{c_i}$ then
        $Signal(c_i, p^MSX, ems^k)$;
    end if
end if
end if

Pop et al. [7] has proposed an oracle-based approach which abstracts component behaviors into a property network spread throughout the component hierarchy. The value change of a property associated with one component can be propagated throughout the property network, potentially changing the values of some properties associated with the other components. A finite-state machine called Oracle is offline constructed to guarantee predictable update time of the property network and derive the new mode of each component. The construction of Oracle requires global information of the property network, while no global information is required in MSL.

Mode switch has been addressed in a number of component models, e.g. SaveCCM [8], Koala [9], Rubus [10], and MyCCM-HI [11]. 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. In MyCCM-HI, each component is associated with a mode automaton which implements its mode switch mechanism. In addition, Fractal [12] is a reflective component model supporting component reconfiguration. Each Fractal component has a membrane that is able to control the reconfiguration of the component. Mode switch is also addressed by languages such as the Architecture Analysis & Design Language [13], where a state machine is used to represent the mode switch behavior of a component. An internal/external event may trigger the state transitions (i.e. mode switch) of the state machine. 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 related work is addressing the specific topic of emergency mode-switch handling for embedded systems.
7 Conclusion and future work

The software complexity of large systems can be effectively reduced and managed by building them with reusable software components and introducing multiple modes. The Mode Switch Logic (MSL) have been proposed for developing such Component-Based Multi-Mode Systems (CBMMSs) and handling their mode switch. In this report, MSL is extended with handling of both emergency and non-emergency concurrently triggered mode switch scenarios. Since an emergency scenario is more critical than non-emergency 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. IHB is proven to satisfy the desired properties such as deadlock freeness and completeness via model checking based verification.

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 [2]. 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.

References


