Handling multiple mode switch scenarios in component-based multi-mode systems

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

June 13, 2013

Abstract

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

1 Introduction

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

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

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

The predominant challenge that a CBMMS exhibits is its mode switch handling. Figure 1 implies that the mode switch of a system may amount to the joint mode switches of many different components at various levels. For instance, the mode switch from m_a^1 to m_a^2 requires the activation of d, the deactivation of f, and the behavior change of b. In order to overcome this challenge, we have developed a Mode Switch Logic (MSL) [4] [5], a systematic approach to the development of CBMMSs and its mode switch handling. In MSL, some component can detect a mode switch event and trigger a mode switch scenario that is propagated to some other components which may also switch mode as a consequence. Currently, MSL is limited by an assumption that only a single mode switch scenario is handled each time. However, in reality, it is possible that multiple scenarios are triggered simultaneously, thus leading to a conflict situation. The contribution of this

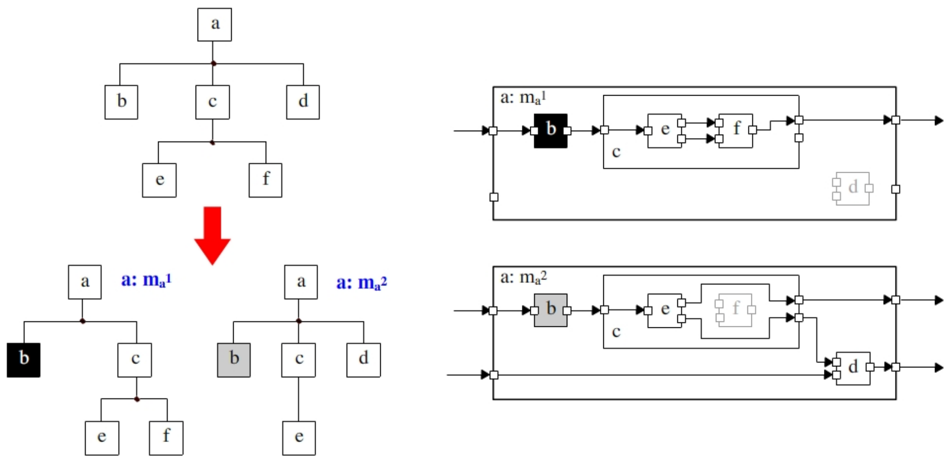

Figure 1: A component-based multi-mode system

report is to handle multiple mode switch scenarios based on MSL.

The remainder of the report is organized as follows: Section 2 gives a brief introduction of MSL. In Section 3, a conflict handling mechanism is proposed for MSL to resolve the conflict due to the triggering of multiple mode switch scenarios. The correctness of the conflict handling mechanism is verified in Section 6. Related work is reviewed in Section 7. Section 8 concludes the report and discusses our future work.

2 The Mode Switch Logic

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

The mode-aware component model defines essential features for a component to be mode-aware. Depicted in Figure 2, a component can support multiple modes, each of which represents a unique configuration. The mode switch of an individual component is realized by reconfiguration, viz. changing its configuration in the current mode to the configuration in the new mode. Furthermore, to enable cooperative mode switch, dedicated mode switch ports are introduced for the cross-layer communication in the component hierarchy. A multi-mode primitive component has a dedicated mode switch port p^{MSX}, which is used to exchange mode related information with its parent during a mode switch. A multi-mode composite component has two dedicated mode switch ports: apart from p^{MSX} that has the same role as for primitive components, the other one is p^{MSX}_{in}, used to exchange mode related information with its subcomponents during a mode switch. The dedicated mode switch ports are marked in blue in Figure 2.

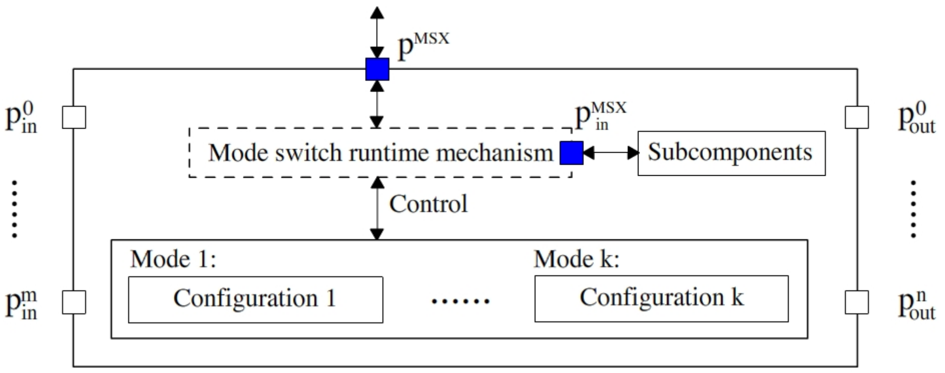

Figure 2: The mode-aware component model

The mode mapping mechanism describes the mode correlation between different components in a compositional manner. Given the current mode of a component at runtime, mode mapping can tell the current modes of other components. In addition, it also determines the new modes of different components when a mode switch is taking place. More details of the mode-aware component model and the mode mapping mechanism can be found in [4].

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

- The Mode Switch Source (MSS): a (primitive or composite) component which can detect a mode switch event (e.g. the value of a sensor reaches a threshold) and actively request to switch mode by triggering a mode switch scenario that can be denoted as $c_k : m^i_{c_k} \rightarrow m^j_{c_k}$, i.e. an MSS c_k requests to switch mode from its current mode $m^i_{c_k}$ to the target mode $m^j_{c_k}$. We hereafter shall use "scenario" to indicate "mode switch scenario".

- The Mode Switch Decision Maker (MSDM): a component which has the authority to approve or reject a mode switch scenario. This component is scenario-dependent and must be either directly or indirectly composed by the MSS that triggers the scenario. A mode switch is triggered only when the MSDM approves a scenario.

- Type A/B components: For a given scenario, a Type A component must switch mode as a consequence, while a Type B component should keep running in its current mode without being affected. Type A and Type B components are determined by mode mapping and are scenario-dependent. In this paper, we use $T_{c_i} = A$ or $T_{c_i} = B$ to denote that c_i is a Type A or Type B component.

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

The MSP protocol to some extent resembles the 2-phase commit protocol for distributed database [6], as a mode switch is triggered when all Type A components are ready to switch mode. For a component c_k, let S_{c_i} denote that the current state of c_i allows a mode switch, and let $\neg S_{c_i}$ denote that the current state of c_i does not allow a mode switch. Also, let P_{c_i} be the parent of c_i and *Top* be the top component in the component hierarchy. Now consider a scenario triggered by an MSS c_i, with c_j as the MSDM and C_M as the set of vertically intermediate components between c_i and c_j in the component hierarchy. We slightly extend the MSP protocol presented in [4] as follows:

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

- *In Condition (2), c_j will reject the `MSR` by issuing an `MSD` (Mode Switch Denial) primitive that is propagated back to c_i via C_M. Mode switch propagation is terminated when c_i receives the `MSD` and no component will switch mode.*

- *In conditions (1) and (3), c_j will approve the `MSR` by issuing an `MSQ` (Mode Switch Query) primitive that is propagated downstream and stepwise to all Type A components. After receiving an `MSQ`, a component c_k is required to reply to P_{c_k} with either an `MSOK` or `MSNOK` primitive. Component c_k replies with an `MSOK` if S_{c_k} (and all its Type A subcomponents reply with an `MSOK` if c_k is composite). Otherwise, if $\neg S_{c_k}$, c_k will reply with an `MSNOK` (without propagating the `MSQ` downstream further if c_k is composite). If c_k receives at least one `MSNOK` from a subcomponent, it will also reply with an `MSNOK`.*

- *If all the Type A subcomponents of c_j have replied with an* `MSOK`*, c_j will trigger a mode switch by issuing an* `MSI` *(Mode Switch Instruction) primitive that follows the propagation trace of the* `MSQ`*. Mode switch propagation is completed when all Type A components have received the* `MSI`*. In contrast, if c_j receives at least one* `MSNOK`*, it will abort the mode switch plan by issuing an* `MSD` *that follows the propagation trace of the* `MSQ`*. Mode switch propagation is terminated when all Type A components have received the* `MSD` *and no component will switch mode.*

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

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

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

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

- *For any primitive component c_i ($T_{c_i} = A$), c_i starts its mode switch by reconfiguring itself upon receiving an* `MSI`*. The mode switch completion of c_i equals its reconfiguration completion. An* `MSC` *(Mode Switch Completion) primitive will be sent from c_i to P_{c_i} when c_i completes its mode switch.*

- *For any composite component c_i ($T_{c_i} = A$), c_i starts its mode switch by reconfiguring itself after its* `MSI` *propagation. Component c_i completes its mode switch when it completes its reconfiguration and it has received an* `MSC` *from all its Type A subcomponents. After that, if $c_i \neq c_j$, an* `MSC` *will be sent from c_i to P_{c_i} after c_i completes its mode switch.*

- *If $T_{c_j} = A$, the system mode switch is completed after the mode switch of c_j. Otherwise, if $T_{c_j} = B$, the system mode switch is completed after c_j has received an* `MSC` *from all its Type A subcomponents.*

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

The mode switch runtime mechanism, chiefly represented by the MSP protocol and the mode switch dependency rule, can be demonstrated by a complete mode switch process illustrated by Figure 3 based on the example in Figure 1. In Figure 3, c triggers a scenario by issuing an `MSR` as an MSS. The `MSR` from c is sent to its parent a which is the MSDM of this scenario. Component a approves the `MSR` by issuing an `MSQ` that is propagated to all Type A components. It should be noted that e is a Type B component, thus not affected by this scenario. After receiving the `MSQ`, each Type A component checks its current state. Here the current state of each Type A component allows a mode switch, therefore an `MSOK` is sent back in response to `MSQ`. Once the MSDM a receives the `MSOK` from its Type A subcomponents b, c and d, it will trigger a mode switch by issuing an `MSI` that follows the propagation trace of the `MSQ`. The propagation of `MSI` results in the reconfiguration of each Type A component, represented by black bars in Figure 3. Finally, `MSC` primitives are sent bottom-up to indicate mode switch completion. The white bars mean that the mode switch of a composite component is blocked by its subcomponents, i.e. a composite component has completed its reconfiguration but is still waiting for at least one `MSC`. Since the MSDM a is a Type A component, the system mode switch is completed upon the mode switch completion of a.

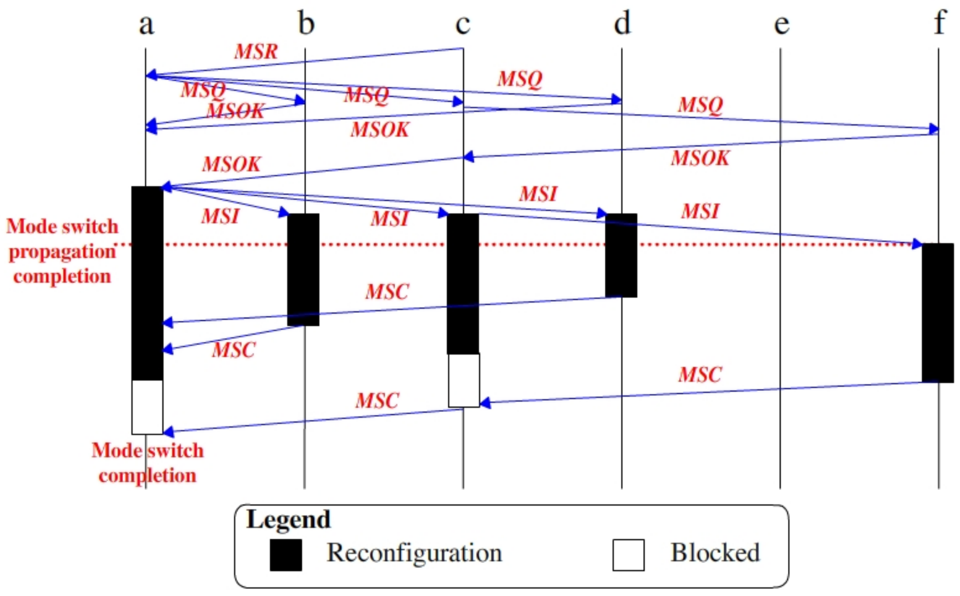

Figure 3: A complete mode switch process

3 The handling of multiple mode switch scenarios

The mode switch runtime mechanism presented in Section 2 assumes that no new scenarios are triggered until the current scenario is completely handled. However, it is likely that a system has multiple MSSs potentially triggering concurrent scenarios. In this section, we propose a conflict handling mechanism to cope with multiple scenarios based on the following assumptions:

1. An ongoing mode switch cannot be aborted or rolled back.
2. An MSS will not trigger a new scenario until the current scenario triggered by itself is completely handled.
3. There is no mode switch failure.

In essence, the conflict handling mechanism introduces separate queues for storing incoming `MSR` and `MSQ` and updates these queues according to a set of pre-defined criteria.

3.1 MSR and MSQ queues

When multiple scenarios are considered, each component must be able to distinguish different scenarios. Hence we introduce the concept *(mode switch) scenario ID*:

Definition 3. *Mode switch scenario ID is a unique ID of a scenario* $c_k : m^i_{c_k} \to m^j_{c_k}$*. Any* `MSX` *primitive must carry a (mode switch) scenario ID* x *that it is associated with, denoted as* msx^x*.*

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

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

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

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

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

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

Whenever c_i receives an `MSQ` from P_{c_i} or decides to trigger a scenario by issuing an `MSQ` ($c_i = Top$), the `MSQ` is enqueued in $c_i.Q_{msq}$. The dequeue conditions of an msq^k in $c_i.Q_{msq}$ are exactly the same as those of the $msr^k_{c_j}$ in $c_i.Q_{msr}$.

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

3.2 The MSR/MSQ queue checking rule

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

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

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

- *If $c_i = Top$, t_1 is the time when c_i issues an msq^k to $SC^A_{c_i}(k)$, while t_2 is the time when the dequeuing conditions (1), (2), or (4) are satisfied for either $c_i.Q_{msr}$ or $c_i.Q_{msq}$ for k.*

Definition 7. *An* `MSR` *in $c_i.Q_{msr}$ is locked if it has been forwarded by c_i to P_{c_i}.*

Based on the aforementioned definitions, the MSR/MSQ queue checking rule is described as follows:

Definition 8. ***The MSR/MSQ queue checking rule:*** *If c_i is not in transition state, then: If $c_i.Q_{msq} \neq \varnothing$, the first* `MSQ` *in $c_i.Q_{msq}$ will be immediately handled, else if $c_i.Q_{msr} \neq \varnothing$ and the first* `MSR` *in $c_i.Q_{msr}$ is not locked, the first* `MSR` *will be immediately handled. The handling of the* `MSQ` *or* `MSR` *follows the mode switch runtime mechanism presented in Section 2.*

The MSR/MSQ queue checking rule enables a component to handle multiple scenarios sequentially. When a component is in transition state for Scenario k, it is dedicated to the handling of k until it leaves the transition state, i.e. when it has completely handled k. By this means, its handling of k can never be interfered by the arrival of another scenario k' which is simply enqueued and handled afterwards.

3.3 The MSR/MSQ queue updating rule

The MSR/MSQ queue checking rule alone is still insufficient to handle multiple scenarios correctly, as it is unaware of the impact of a scenario upon other pending scenarios, implicitly assuming that different scenarios are independent of each other. Nevertheless, after a component completes its mode switch for Scenario k, if $\exists msr^{k'}_{c_j}$ in $c_i.Q_{msr}$ ($c_j \in SC_{c_i} \cup \{c_i\}$), $msr^{k'}_{c_j}$ may not be valid any more. This problem can be illustrated by a small example. Let's consider a system with its component hierarchy presented in Figure 1. Tables 1 and 2 give the basic mode mappings of the two composite components a and c. In each table, modes in the same column are mapped. For example, according to Table 1, when a is running in mode m^1_a, b must be running in m^1_b, c can run in either m^1_c or m^2_c, and d is deactivated. Figure 4 depicts three scenarios: (1) $S_1 = (b : m^1_b \to m^2_b)$; (2) $S_2 = (e : m^1_e \to m^2_e)$; (3) $S_3 = (f : m^1_f \to m^2_f)$. For S_1 and S_3, all components are Type A components, whilst for S_2, only c and e are Type A components. For each scenario, the current possible mode and the target mode of each Type A component are also defined in Figure 4. For instance, for S_1, $m^1_f \to m^2_f$ means that S_1 will imply the mode switch of f from m^1_f to m^2_f. The given mode mappings imply that when b is running in m^1_b, e can be in either m^1_e or m^2_e while f must be running in m^1_f. Therefore, all three scenarios can be simultaneously triggered.

Table 1: The mode mapping table of a

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

Table 2: The mode mapping table of c

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

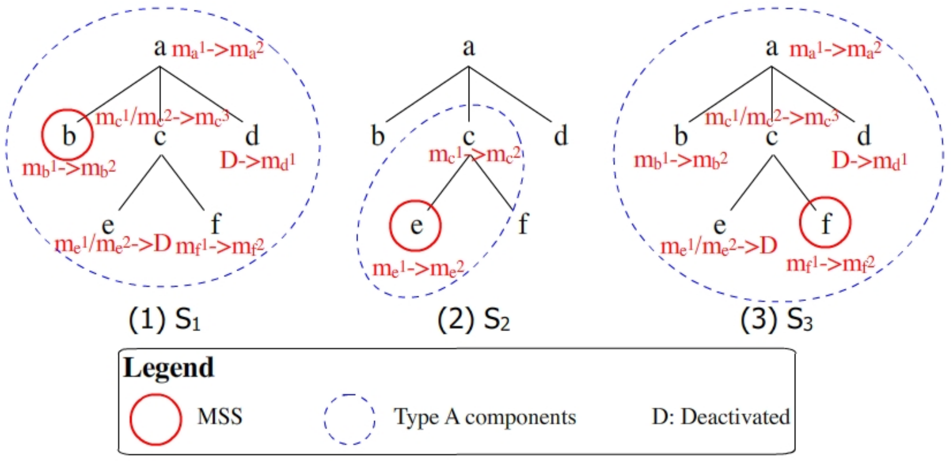

Figure 4: Mode switch scenarios

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

Sometimes a pending scenario may still remain valid in spite of the mode switch completion of another scenario. Suppose S_2 and S_3 are simultaneously triggered. A possible outcome is that msr^{S_2} arrives at $c.Q_{msr}$ earlier than msr^{S_3}. Evidently, c will first handle msr^{S_2}. If a mode switch is performed and completed based on S_2, the pending scenario S_3 is still valid because f (the MSS of S_3) is a Type B component for S_2 and is unaffected by S_1. In this case, the pending MSR primitives $msr_f^{S_3}$ in $c.Q_{msr}$ and $msr_f^{S_3}$ in $f.Q_{msr}$ should not be removed without handling.

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

Definition 9. ***The MSR/MSQ queue updating rule:*** *Consider a component c_i and a scenario k. After the mode switch completion of c_i based on k ($T_{c_i}^k = A$) or after c_i has received an MSC from all $c_j \in SC_{c_i}^A(k)$ ($c_i \in CC, T_{c_i}^k = B$),*

- *If $c_i \in PC$, then $T_{c_i}^k = A$. If $\exists msr_{c_i}^{k'} \in c_i.Q_{msr}$ ($k' \neq k$), then c_i must be the MSS of k'. Hence c_i will remove $msr_{c_i}^{k'}$ from $c_i.Q_{msr}$.*
- *If $c_i \in CC$, a pending MSR in $c_i.Q_{msr}$ can derive from either c_i or $c_j \in SC_{c_i}$. If $\exists msr_{c_i}^{k'} \in c_i.Q_{msr}$ ($k' \neq k$), then (1) if $T_{c_i}^k = A$, then $msr_{c_i}^{k'}$ will be removed from $c_i.Q_{msr}$; (2) if $T_{c_i}^k = B$, then $msr_{c_i}^{k'}$ will be kept in $c_i.Q_{msr}$. If $\exists msr_{c_j}^{k'} \in c_i.Q_{msr}$, then (1) if $T_{c_j}^k = A$, $msr_{c_j}^{k'}$ will be removed from $c_i.Q_{msr}$; (2) if $T_{c_j}^k = B$, $msr_{c_j}^{k'}$ will be kept in $c_i.Q_{msr}$. In Case (2), if $msr_{c_j}^{k'}$ is locked, c_i will handle it as a new MSR. As an exception, if $\exists msr_{c_j}^{k'} \in c_i.Q_{msr}$ which arrives while c_i is waiting for the MSC from SC_{c_i}, this $msr_{c_j}^{k'}$ will be kept for the current round of $c_i.Q_{msr}$ updating, even if $T_{c_j}^k = A$.*
- *If $c_i = Top$, $T_{c_i}^k = A$, and $\exists msq^{k'} \in c_i.Q_{msq}$, then $msq^{k'}$ will be removed from $c_i.Q_{msq}$.*

The essence of the MSR/MSQ queue updating rule is to remove a pending MSR/MSQ which becomes invalid due to the mode switch of a previous scenario. It should be noted that the MSR/MSQ queue updating rule does not remove any msr^k or msq^k since k is the currently handled scenario instead of a pending scenario. A msr^k or msq^k is dequeued when the dequeuing condition (see Section 3.1) is satisfied.

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

Additional attention must be paid to the exception that c_i receives a $msr^{k'}_{c_j}$ from $c_j \in SC_{c_i}$ while waiting for the `MSC` from SC_{c_i}. Under this condition, $msr^{k'}_{c_j}$ is not removed from $c_i.Q_{msr}$ even if $T^k_{c_j} = A$. The reason is ascribed to the mode switch dependency rule which ensures that c_j must complete mode switch before c_i. Hence $msr^{k'}_{c_j}$ must be sent after the mode switch of c_j based on k and must be valid.

3.4 Discussion

The MSR/MSQ queue checking rule assigns higher priority to MSQ queue than MSR queue such that MSQ queue is always checked before MSR queue. A potential problem of this priority assignment is the bias towards scenarios triggered by a component closer to *Top*. For instance, let's compare two scenarios, k_1 and k_2, triggered by c_1 and c_2 respectively at the same time. *Top* is the MSDM for both scenarios and c_1 is much closer to *Top*. Since it takes more steps for msr^{k_2} to reach *Top* compared with msr^{k_1}, k_1 is more likely to be handled by *Top* before k_2 despite their simultaneous triggering. When a component c_k, with msq^{k_1} in $c_k.Q_{msq}$ and msr^{k_2} in $c_k.Q_{msr}$, checks its MSR/MSQ queues, msq^{k_1} will be first handled while msr^{k_2} may be even removed afterwards due to the MSR/MSQ queue updating rule.

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

4 Analysis of the MSR/MSQ queue sizes

Section 3 has described how multiple scenarios can be handled by MSR and MSQ queues. Now the question is: Are MSR queue and MSQ queue always necessary for all components? And what about their queue sizes?The answers are as follows:

Theorem 1. *For a component $c_i \in PC$, if c_i is never an MSS, c_i has no MSR queue; if c_i is the MSS of at least one scenario, c_i has an MSR queue with the maximum size 1.*

Proof. An incoming `MSR` to be enqueued in the MSR queue of a component c_k can only come from SC_{c_k} (if $c_k \in CC$) or from c_k itself (if c_k is the corresponding MSS). Since $c_i \in PC$, $SC_{c_i} = \varnothing$, thus an `MSR` enqueued into $c_i.Q_{msr}$ can only come from c_i itself. If c_i is never an MSS, $c_i.Q_{msr}$ will always be empty, hence no MSR queue is needed. If c_i is the MSS of at least one scenario, c_i can enqueue an `MSR` issued by itself in $c_i.Q_{msr}$. Since it is assumed that an MSS does not trigger a new scenario before its current scenario is completely handled, c_i cannot enqueue another self-triggered `MSR` before the previous `MSR` issued by it is dequeued from $c_i.Q_{msr}$. Therefore, the maximum size of $c_i.Q_{msr}$ is 1. $\square$

Theorem 2. *The maximum size of the MSQ queue of a component $c_i \neq Top$ is 2.*

Proof. Consider a component c_i with an MSQ queue $c_i.Q_{msq}$. Let N be the number of elements in $c_i.Q_{msq}$ and $c_j = P_{c_i}$. When $c_i.Q_{msq} = \varnothing$, c_i can receive a msq^k from P_{c_i}. According to Definition 6, after sending msq^k to c_i, c_j must be in transition state for k. Apparently, c_j cannot send another `MSQ` to c_i before it leaves the transition state.

If a mode switch is performed based on k, the mode switch dependency rule specifies that c_i must complete mode switch before c_j. As c_i completes the mode switch for k, msq^k will be removed from $c_i.Q_{msq}$. Therefore, if c_j sends another `MSQ` $msq^{k'}$ to c_i right after c_j completes its mode switch for k, N will become 1 from 0. The condition will be the same as when c_i receives msq^k.

If k is rejected, c_j leaves transition state by sending a msd^k to c_i. Then c_i may leave transition state after it receives msd^k (if $c_i \in PC \vee SC^A_{c_i}(k) = \varnothing$), or after it propagates msd^k to $SC^A_{c_i}$ (if $SC^A_{c_i}(k) \neq \varnothing$). Since it may take some time for c_i to dequeue msq^k after it receives msd^k, if c_j immediately sends another `MSQ` $msq^{k'}$ to c_i as c_j leaves transition state, by the time c_i receives $msq^{k'}$, msq^k may have not been removed from $c_i.Q_{msq}$. Therefore, N can reach 2. By the time c_i leaves transition state for k and dequeues msq^k, N will become 1 while c_j must be in transition state for k' and cannot send another `MSQ`. Hence, c_j can never send any `MSQ` to c_i when $N = 2$.

The reasoning above proves that N cannot be bigger than 2, i.e. the maximum size of $c_i.Q_{msq}$ is 2 when $c_i \neq Top$. $\square$

Theorem 3. *The maximum size of the MSQ queue of a component $c_i = Top$ is 1.*

Proof. Since c_i has no parent, $c_i.Q_{msq}$ may only contain an `MSQ` issued by c_i itself as the MSS. Therefore, if c_i is never an MSS, there is no need to have $c_i.Q_{msq}$. If c_i is the MSS of at least one scenario, $c_i.Q_{msq}$ can never contain multiple `MSQ` primitives due to the assumption that an MSS will not trigger a new scenario until the current scenario triggered by itself is completely handled. Therefore, the maximum size of $c_i.Q_{msq}$ is 1 when $c_i = Top$. $\square$

Lemma 1. *For $c_i \in CC$, at any instant, there are at most two `MSR` primitives from $c_j \in SC_{c_i}$ in $c_i.Q_{msr}$.*

Proof. Suppose $\exists msr^k_{c_j} \in c_i.Q_{msr}$. Usually, this implies that c_j has a locked msr^k at the head of $c_j.Q_{msr}$. Obviously, c_j will not send another `MSR` $msr^{k'}$ to c_i before msr^k is completely handled. Therefore, there is supposed to be at most one `MSR` from c_j in $c_i.Q_{msr}$ at any instant. However, the mode switch dependency rule requires that c_j completes mode switch before c_i. Consequently, a special case is that c_j sends another $msr^{k'}$ to c_i right after its mode switch. At that instant, c_j must have removed msr^k from $c_j.Q_{msr}$. However, c_i may have not completed its mode switch. By the time c_i receives $msr^{k'}$, there will be two `MSR` primitives from c_j: $msr^k_{c_j}$ and $msr^{k'}_{c_j}$. After c_i completes its mode switch based on k, according to the dequeuing condition for an `MSR` and the MSR/MSQ queue updating rule, $msr^k_{c_j}$ will be dequeued while $msr^{k'}_{c_j}$ is kept in $c_i.Q_{msr}$. At that moment, c_j cannot send any other `MSR` to c_i because $msr^{k'}$ is till locked. Therefore, there are at most two `MSR` primitives from c_j in $c_i.Q_{msr}$ at any instant. $\square$

Theorem 4. *For a component $c_i \in CC$, if c_i is never an MSS, the maximum size of $c_i.Q_{msr}$ is $2 * |SC_{c_i}|$. If $c_i \neq Top$ and c_i is the MSS of at least one scenario, the maximum size of $c_i.Q_{msr}$ is $2 * |SC_{c_i}| + 1$.*

Proof. If c_i is never an MSS, any `MSR` in $c_i.Q_{msr}$ must come from a subcomponent of c_i. Lemma 1 has proved that for each $c_j \in SC_{c_i}$, there are at most two `MSR` primitives from c_j in $c_i.Q_{msr}$ at any instant. If any subcomponent of c_i can send an `MSR` to c_i, the maximum number of elements of $c_i.Q_{msr}$ will be $2 * |SC_{c_i}|$. If $c_i \neq Top$ and c_i is the MSS of at least one scenario, $c_i.Q_{msr}$ can also include one and at most one `MSR` issued by c_i itself at any instant. Hence, the maximum size of $c_i.Q_{msr}$ is $2 * |SC_{c_i}| + 1$. $\square$

In summary, Table 3 shows the allocation of MSR and MSQ queues to different types of components together with the maximum possible size of each queue.

Table 3: The allocation of MSR and MSQ queues

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

5 Implementing the conflict handling mechanism

In this section, we implement our conflict handling mechanism as algorithms. In general, the conflict handling mechanism of a component can be realized by at most three tasks:

- $MS\_detection(c_i)$: This task triggers a scenario of an MSS c_i. It is only executed when c_i does not have any pending scenario to be handled.
- $MSR\_MSQ\_enqueue(c_i)$: This task checks the arrival of an `MSR` or `MSQ` and puts it in the corresponding queue of c_i. Furthermore, if c_i has sent a msr^k to P_{c_i} and msr^k is directly rejected by the MSDM, c_i will receive a msd^k from P_{c_i}. Then c_i should dequeue the locked msr^k from $c_i.Q_{msr}$ (and further propagate the msd^k to $c_j \in SC_{c_i}$ if $c_i \in CC$ and msr^k comes from c_j).
- $CheckQueue(c_i)$: This task implements the MSR/MSQ queue checking rule. It is only executed when c_i is not in any transition state. It calls functions $HandleMSQ(c_i)$ and $HandleMSR(c_i)$.

Tasks $MSR\_MSQ\_enqueue(c_i)$ and $CheckQueue(c_i)$ are essential for any component, while task $MS\_detection(c_i)$ is only essential if c_i is the MSS of at least one scenario. With respect to these three tasks, the following notations deserve extra explanation:

- $MS\_event\_detected$ is a boolean variable set to true when the MSS c_i detects a mode switch event.
- $Derive\_new\_mode(c_i)$ is a function that derives the new mode $m_{c_i}^{new}$ of c_i when c_i detects a mode switch event as the MSS.
- $Wait(c_i, A, B)$ and $Signal(c_i, A, B)$ are used for c_i to receive and send primitive B via the dedicated mode switch port A, which is either p^{MSX} (the port for exchanging primitives with P_{c_i}) or p_{in}^{MSX} (the port for exchanging primitives with SC_{c_i}).

- $msr^k(c_i, m_{c_i}, m_{c_i}^{new})$ represents an MSR associated with a scenario k with the immediate sender c_i requesting to switch from mode m_{c_i} to $m_{c_i}^{new}$. Similarly, $msx^k(c_i, m_{c_i}^{new})$ represents a primitive other than MSR for k.
- *MSC_Collecting* is a boolean variable set to true when a composite component is expecting an MSC from its subcomponents.
- $valid^k$ is a boolean variable of $c_i \in CC$ set to true when c_i receives an $msr_{c_j}^k$ from $c_j \in SC_{c_i}$ while *MSC_Collecting* is true.
- $k \leftarrow c_j$: Scenario k comes from c_j.
- *locked* is a boolean variable of c_i set to true if the first MSR in $c_i.Q_{msr}$ is locked.
- $enqueue(A, B)$ is a function enqueuing the primitive A (either MSR or MSQ) in queue B.
- $Dequeue(c_i, k)$: If $\exists msq^k \in c_i.Q_{msq}$, then c_i will remove msq^k from $c_i.Q_{msq}$; if $\exists msr^k \in c_i.Q_{msr}$, then c_i will remove msr^k from $c_i.Q_{msr}$. Besides, after c_i removes msr^k, if *locked* is true, c_i will set it to false.
- $TransitionS$ is a boolean variable of c_i set to true when c_i is in transition state.
- $HandleMSQ(c_i)$ and $HandleMSR(c_i)$ are functions for the handling of an MSQ and MSR, respectively, following the mode switch runtime mechanism presented in Section 2.

Algorithm 1 $MS\_detection(c_i)$

```
loop
  if $MS\_event\_detected$ then
    $Derive\_new\_mode(c_i)$;
    if $c_i \neq Top$ then
      $enqueue(msr_{c_i}^k, c_i.Q_{msr})$;
    else{$c_i = Top$}
      $enqueue(msq^k, c_i.Q_{msq})$;
    end if
  end if
end loop
```

Consistent with the MSR/MSQ queue checking rule, Algorithm 3 only checks $c_i.Q_{msr}$ and $c_i.Q_{msq}$ when c_i is not in transition state. Since $c_i.Q_{msq}$ is always checked before $c_i.Q_{msr}$, this implies that $c_i.Q_{msq}$ has a higher priority. Algorithm 3 further calls $HandleMSQ(c_i)$ and $HandleMSR(c_i)$. Depending on the type of c_i (primitive, non-top composite, or the top component),

Algorithm 2 *MSR_MSQ_enqueue*(c_i)

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

Algorithm 3 *CheckQueue*(c_i)

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

$\textit{HandleMSQ}(c_i)$ is described by algorithms 4, 6, 8 while $\textit{HandleMSR}(c_i)$ is described by algorithms 5, 7, 9. Generally speaking, algorithms 4-9 handle an `MSR` or an `MSQ` based on a single mode switch scenario. They are essentially the same as the algorithms of the mode switch runtime mechanism given in [4], yet they additionally consider transition state, locked `MSR`, the dequeuing of an `MSR` or `MSQ`, as well as the MSR/MSQ queue updating rule which is implemented in Algorithm 11.

Algorithms 6-9 further call another function $\textit{ModeSwitch}(c_i, \textit{MSDM}, \textit{Top})$ described in Algorithm 10 which extends the same algorithm in [4] with the additional concerns mentioned above. Function $\textit{ModeSwitch}(c_i, \textit{MSDM}, \textit{Top})$ implements the behavior of c_i when c_i is about to propagate an `MSQ` to $SC_{c_i}A$. *MSDM* is a boolean variable set to true when c_i is the MSDM for a specific mode switch scenario; *Top* is a boolean variable set to true when $c_i = \textit{Top}$.

Please note that atomic component execution is not considered in this report as the handling of atomic component execution is a separate topic which has already been discussed in [4]. Since atomic component execution can be easily handled by delaying the `MSQ` propagation to some components, our conflict handling mechanism also works when atomic component execution is considered. Algorithms 4-10 contain the following functions:

- $\textit{Reconfiguration}(c_i, m_{c_i}, m_{c_i}^{new})$ is a function reconfigurating c_i from its current mode m_{c_i} to its new mode $m_{c_i}^{new}$.
- $\textit{Stop\_running}(c_i, m_{c_i})$ and $\textit{Resume}(c_i, m_{c_i})$ mean that c_i stops and resumes execution in the current mode m_{c_i}. Similarly, $\textit{Start\_running}(c_i, m_{c_i}^{new})$ starts the execution of c_i in its new mode. The mode switch runtime mechanism requires that c_i stops its current execution when it enters transition state as a Type A component. When c_i leaves transition state, it will either run in the new mode, or resume its execution in the current mode.
- $\textit{MS\_possible}(c_i, m_{c_i})$ is a boolean function returning true if the current state of c_i in the mode m_{c_i} allows a mode switch.
- $\textit{ModeMapping}(c_i)$ is a function performing the mode mapping of $c_i \in CC$ in its current mode.
- $\textit{MSOKNOK\_Collection}(c_i)$ is a boolean function returning true if c_i has received an `MSOK` from all c_j in $SC_{c_i}^{A}$. Similarly, $\textit{MSC\_Collection}(c_i)$ is a boolean function returning true if c_i has received an `MSC` from all c_j in $SC_{c_i}^{A}$.
- $\textit{UpdateQueue}(c_i, k)$, i.e. Algorithm 11, is a function implementing the MSR/MSQ queue updating rule.

Algorithm 4 *HandleMSQ*($c_i \in PC$)

```
$TransitionS := true$;
$Stop\_running(c_i, m_{c_i})$;
if $MS\_possible(c_i, m_{c_i})$ then
    $Signal(c_i, p^{MSX}, msok^k(c_i, m_{c_i}^{new}))$;
    $Wait(c_i, p^{MSX}, primitive)$;
    if $primitive = msi^k(c_i, m_{c_i}^{new})$ then
        $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
        $Signal(c_i, p^{MSX}, msc^k(c_i, m_{c_i}^{new}))$;
        $UpdateQueue(c_i, k)$;
        $TransitionS := false$;
        $Start\_running(c_i, m_{c_i}^{new})$;
    else{$c_i$ receives an MSD}
        $Dequeue(c_i, k)$;
        $TransitionS := false$;
        $Resume(c_i, m_{c_i})$;
    end if
else{The current state of $c_i$ does not allow a mode switch}
    $Signal(c_i, p^{MSX}, msnok^k(c_i, m_{c_i}^{new}))$;
    $Wait(c_i, p^{MSX}, primitive)$;
    if $primitive = msd^k(c_i, m_{c_i}^{new})$ then
        $Dequeue(c_i, k)$;
        $TransitionS := false$;
        $Resume(c_i, m_{c_i})$;
    end if
end if
```

Algorithm 5 *HandleMSR*($c_i \in PC$)

```
$Signal(c_i, p^{MSX}, msr^k(c_i, m_{c_i}, m_{c_i}^{new}))$;
$locked := true$;
```

Algorithm 6 *HandleMSQ*$(c_i \in \widetilde{CC})$

```
$TransitionS := true$;
$Stop\_running(c_i, m_{c_i})$;
**if** $MS\_possible(c_i, m_{c_i})$ **then**
  $ModeMapping(c_i)$;
  **if** $SC^A_{c_i} = \varnothing$ **then**
    $Signal(c_i, p^{MSX}, msok^k(c_i, m^{new}_{c_i}))$;
    $Wait(c_i, p^{MSX}, primitive)$;
    **if** $primitive = msd^k(c_i, m^{new}_{c_i})$ **then**
      $Dequeue(c_i, k)$;
      $TransitionS := false$;
      $Resume(c_i, m_{c_i})$;
    **else**$\{primitive = msi^k(c_i, m^{new}_{c_i})\}$
      $Reconfiguration(c_i, m_{c_i}, m^{new}_{c_i})$;
      $Signal(c_i, p^{MSX}, msc^k(c_i, m^{new}_{c_i}))$;
      $UpdateQueue(c_i, k)$;
      $TransitionS := false$;
      $Start\_running(c_i, m^{new}_{c_i})$;
    **end if**
  **else**$\{SC^A_{c_i} \neq \varnothing\}$
    $ModeSwitch(c_i, false, false)$;
  **end if**
**else**{The current state of $c_i$ does not allow a mode switch}
  $Signal(c_i, p^{MSX}, msnok^k(c_i, m^{new}_{c_i}))$;
  $Wait(c_i, p^{MSX}, primitive)$;
  **if** $primitive = msd^k(c_i, m^{new}_{c_i})$ **then**
    **if** $k \leftarrow c_j \in SC_{c_i}$ **then**
      $Signal(c_i, p^{MSX}_{in}, msd^k(c_j, m^{new}_{c_j}))$;
    **end if**
    $Dequeue(c_i, k)$;
    $TransitionS := false$;
    $Resume(c_i, m_{c_i})$;
  **end if**
**end if**
```

Algorithm 7 *HandleMSR*$(c_i \in \widetilde{CC})$

```
$ModeMapping(c_i)$;
**if** $m_{c_i}^{new} = m_{c_i}$ **then**
  $TransitionS := true$;
  $ModeSwitch(c_i, true, false)$;
**else**$\{m_{c_i}^{new} \neq m_{c_i}\}$
  **if** $MS\_possible(c_i, m_{c_i})$ **then**
    $Signal(c_i, p^{MSX}, msr^k(c_i, m_{c_i}, m_{c_i}^{new}))$;
    $locked := true$;
  **else**$\{\neg MS\_possible(c_i, m_{c_i})\}$
    $Signal(c_i, p_{in}^{MSX}, msd^k(c_j, m_{c_j}^{new}))$; $\{k \leftarrow c_j\}$
    $Dequeue(c_i, k)$;
  **end if**
**end if**
```

Algorithm 8 *HandleMSQ*$(c_i = Top)$

```
$TransitionS := true$;
**if** $SC_{c_i}^A = \varnothing$ **then**
  $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
  $UpdateQueue(c_i, k)$;
  $TransitionS := false$;
  $Start\_running(c_i, m_{c_i}^{new})$;
**else**
  $Stop\_running(c_i, m_{c_i})$;
  $ModeSwitch(c_i, true, true)$;
**end if**
```

Algorithm 9 *HandleMSR*$(c_i = Top)$

```
$ModeMapping(c_i)$;
**if** $m_{c_i}^{new} = m_{c_i}$ || $MS\_possible(c_i, m_{c_i})$ **then**
  **if** $m_{c_i}^{new} \neq m_{c_i}$ **then**
    $Stop\_running(c_i, m_{c_i})$;
  **end if**
  $TransitionS := true$;
  $ModeSwitch(c_i, true, true)$;
**else**$\{\neg MS\_possible(c_i, m_{c_i})\}$
  $Signal(c_i, p_{in}^{MSX}, msd^k(c_j, m_{c_j}^{new}))$; $\{k \leftarrow c_j\}$
  $Dequeue(c_i, k)$;
**end if**
```

Algorithm 10 *ModeSwitch*($c_i \in CC, MSDM, Top$)

```
$\forall c_j \in SC_{c_i}^A : Signal(c_i, p_{in}^{MSX}, msq^k(c_j, m_{c_j}^{new}))$;
if $MSOKNOK\_Collection(c_i)$ then
  if $MSDM \mid\mid Top$ then
    $\forall c_j \in SC_{c_i}^A : Signal(c_i, p_{in}^{MSX}, msi^k(c_j, m_{c_j}^{new}))$;
    if $T_{c_i} = A$ then
      $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
    end if
    if $MSC\_Collection(c_i)$ then
      $UpdateQueue(c_i, k)$;
      $TransitionS := false$;
      if $T_{c_i} = A$ then
        $Start\_running(c_i, m_{c_i}^{new})$;
      else{$T_{c_i} = B$}
        $Resume(c_i, m_{c_i})$;
      end if
    end if
  else{$c_i$ is neither an MSDM nor the top component}
    $Signal(c_i, p^{MSX}, msok^k(c_i, m_{c_i}^{new}))$;
    $Wait(c_i, p^{MSX}, primitive)$;
    if $primitive = msi^k(c_i, m_{c_i}^{new})$ then
      $\forall c_j \in SC_{c_i}^A : Signal(c_i, p_{in}^{MSX}, msi^k(c_j, m_{c_j}^{new}))$;
      $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
      if $MSC\_Collection(c_i)$ then
        $Signal(c_i, p^{MSX}, msc^k(c_i, m_{c_i}^{new}))$;
        $UpdateQueue(c_i, k)$;
        $TransitionS := false$;
        $Start\_running(c_i, m_{c_i}^{new})$;
      end if
    else{$c_i$ receives an MSD}
      $\forall c_j \in SC_{c_i}^A : Signal(c_i, p_{in}^{MSX}, msd^k(c_j, m_{c_j}^{new}))$;
      $Dequeue(c_i, k)$;
      $TransitionS := false$;
      $Resume(c_i, m_{c_i})$;
    end if
  end if
```

```
else{$c_i$ receives at least one MSNOK}
  if $MSDM$ || $Top$ then
    $\forall c_j \in SC^A_{c_i} : Signal(c_i, p^{MSX}_{in}, msd^k(c_j, m^{new}_{c_j}))$;
    $Dequeue(c_i, k)$;
    $TransitionS := false$;
    $Resume(c_i, m_{c_i})$;
  else{$c_i$ is neither an MSDM nor the top component}
    $Signal(c_i, p^{MSX}, msnok^k(c_i, m^{new}_{c_i}))$;
    $Wait(c_i, p^{MSX}, primitive)$;
    if $primitive = msd^k(c_i, m^{new}_{c_i})$ then
      $\forall c_j \in SC^A_{c_i} : Signal(c_i, p^{MSX}_{in}, msd^k(c_j, m^{new}_{c_j}))$;
      $Dequeue(c_i, k)$;
      $TransitionS := false$;
      $Resume(c_i, m_{c_i})$;
    end if
  end if
end if
```

The MSR/MSQ queue updating rule is implemented as Algorithm 11, where $dequeue(A, B)$ is a function dequeuing a primitive A from queue B. Actually, $dequeue(A, B)$ can be considered as a sub-function of $Dequeue(c_i, k)$.

The relations between algorithms 3-11 are illustrated in Figure 5.

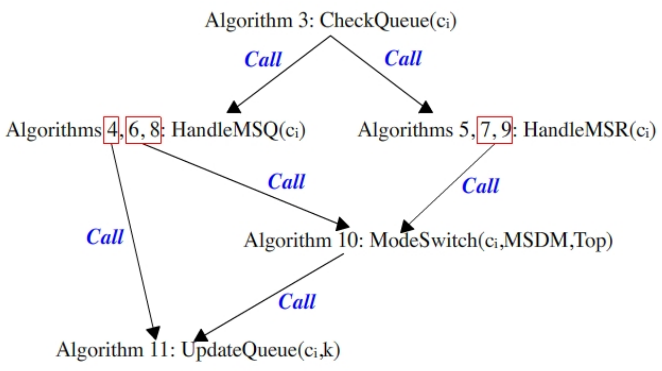
Algorithm 3: CheckQueue(ci)
Call
Call
Algorithms 4, 6, 8: HandleMSQ(ci)
Algorithms 5, 7, 9: HandleMSR(ci)
Call
Call
Call
Algorithm 10: ModeSwitch(ci,MSDM,Top)
Call
Algorithm 11: UpdateQueue(ci,k)

Figure 5: The call graph of algorithms 3-11

6 Verification of the conflict handling mechanism

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

Algorithm 11 *UpdateQueue*(c_i, k)

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

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

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

We resort to the combination of model checking and mathematical induction for the verification of the conflict handling mechanism. In the following subsections, we shall shed light on the modeling of the conflict handling mechanism and its verification, respectively.

6.1 Modeling the conflict handling mechanism

The conflict handling mechanism is modeled in the model checker UPPAAL [7] with regard to the six cases listed in Table 3. In each case, the conflict handling mechanism is implemented in a target component. Shown in Figure 6, the target component can have a parent stub or two child stubs for some cases which simulate the behaviors of its parent and subcomponents by running the same conflict handling mechanism. Both the parent stub and child stub can trigger scenarios at any time, as long as they do not have any self-triggered pending scenarios.

Next we shall elaborate on the UPPAAL models that implement the conflict handling mechanism. We have built six sets of UPPAAL models based on the six cases identified in Table 3. As indicated in Figure 6, only

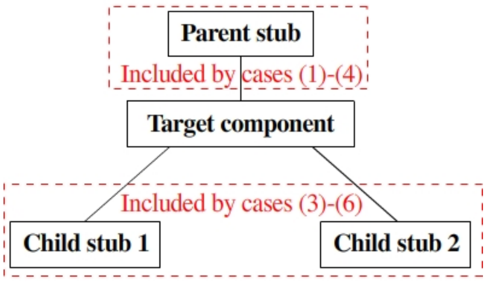

Figure 6: The modeling structure in UPPAAL

cases (3) and (4) require both the parent stub and child stubs. Compared with Case (3), Case (4) additionally considers the self-triggered scenario from the target component itself. Therefore, we shall focus on describing the UPPAAL model for Case (4) which basically covers all the important issues addressed by the models for the other five cases.

The UPPAAL model for Case (4) consists of four automata, each automaton being a UPPAAL template:

- MSRMSQenqueue: It implements the algorithms $MS\_detection(c_i)$ and $MSR\_MSQ\_enqueue(c_i)$ (i.e. algorithms 1 and 2).
- TargetComp: It implements the parent stub.
- ParentStub: It implements the target component with the conflict handling mechanism.
- ChildStub: It implements the child stub and can be instantiated into arbitrary number of child stubs. In our model, two child stubs are instantiated.

In our model, the transmission of a primitive is realized by channels which can synchronize different automata. A primitive is sent by a channel ending with "!" while the primitive is received by the same channel, yet ending with "?". Figure 7 depicts MSRMSQenqueue whose transitions are numbered in red. Transitions 1 and 2 enqueue an incoming `MSR` or `MSQ`. Besides, the target component itself can also trigger a scenario by Transition 7, if it does not have any pending scenario to be handled. Transition 3 handles an `MSD` sent from the parent. If this `MSD` is associated with an `MSR` that comes from a child stub, the target component will further propagate the `MSD` to the child stub by Transition 4. Otherwise, if this `MSD` is associated with an `MSR` issued by the target component, Transition 5 will be taken without further `MSD` propagation. Transition 6 is simply a modeling artifact used to synchronize an urgent channel.

The parent stub is illustrated in Figure 8. The parent stub can receive an `MSR` from the target component at any time when the parent stub is in

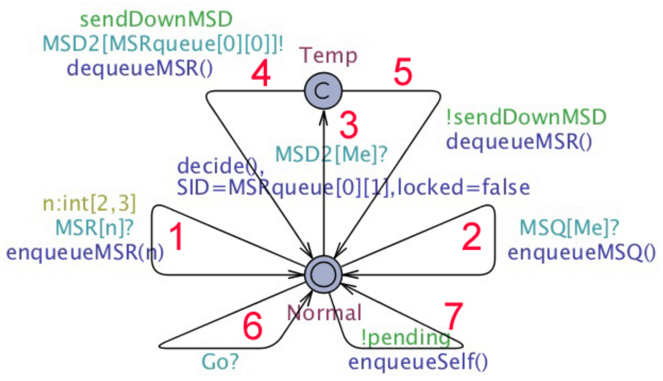

Figure 7: The UPPAAL model for enqueuing `MSR` and `MSQ`

its initial state **Init**. After receiving an `MSR` from the target component, the parent stub will set a boolean variable *MSRpending* to true. When *MSRpending* is true, the parent stub can take one of three possible actions: (1) to send an `MSQ` not associated with the `MSR` from the target component; (2) to send an `MSQ` associated with the `MSR` from the target component; (3) to send an `MSD` associated with the `MSR` from the target component. Actions (1) and (2) correspond to Transition 5, yet with different scenario IDs. Action (3) corresponds to Transition 3. If *MSRpending* is false, the parent stub can only take Action (1) by transitions 4 and 5. The other transitions exchange primitives with the target component, simulating a real parent of the target component. The parent exhibits some random behavior in the sense that it may send either an `MSI` or `MSD` to the target component after it receives an `MSOK` from the target component.

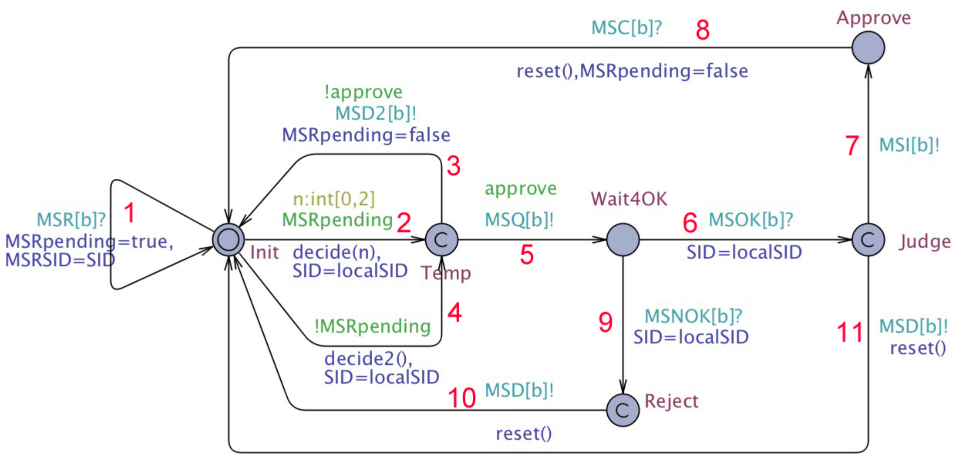

Figure 8: The UPPAAL model for the parent stub

Figure 9 delineates the common template for the child stub. It can exchange primitives with the target component, simulating a real child of the target component. A child stub can trigger a scenario by Transition 1. If this scenario is directly rejected by the MSDM, the child stub will receive an `MSD` from the target component, realized by Transition 2. Note that such an `MSD` is modeled by a channel *MSD2[Me]* in UPPAAL, with *Me* as the identity of the child stub. This distinguishes the case when the child stub receives an `MSD` for a different reason by the channel *MSD[Me]*. The child stub also has random behavior as it can send either an `MSOK` or `MSNOK` in response to the `MSQ` from the target component.

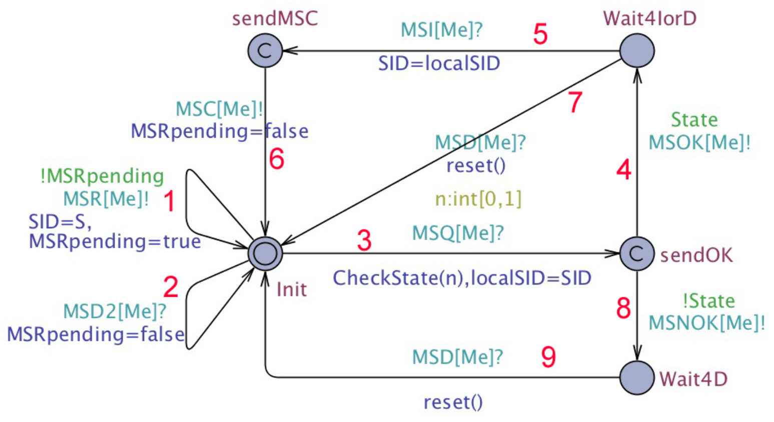

Figure 9: The UPPAAL model for the child stub

The UPPAAL model of the target component is presented in Figure 10. In its initial state **Idle**, it checks its MSQ and MSR queues. If its MSQ is not empty, it will handle the first MSQ in its MSQ queue by Transition 7. If the current state of the target component allows such a mode switch, it will fire Transition 8 or 9. Transition 8 is taken if the `MSQ` being handled is associated with the `MSR` sent by the target component before. In this case, the target component can reuse the mode mapping result based on the `MSR`. Otherwise, Transition 9 is taken and mode mapping must be performed. Alternatively, if the current state of the target component does not allow such a mode switch, it will send an `MSNOK` to the parent stub by Transition 10. After that it will receive an `MSD` from the parent stub by Transition 11. If the scenario being handled comes from a child stub, the target component will propagate the `MSD` to the child stub by Transition 13. Otherwise, if this scenario is triggered by the target component itself, Transition 12 will be taken with no further `MSD` propagation.

State **MSQprop** has two outgoing transitions 14 (if the target component has at least one Type A subcomponent for the current scenario) and 35 (if the target component has no Type A subcomponents for the current scenario). If Transition 8 is taken, the next transition must be 14 because the current scenario comes from a child stub which must be Type A. If Transition 9 is taken, then the next transition can be either 14 or 35. The target component propagates an `MSQ` to its Type A subcomponents by transitions 14-16. Then in State **OKCollection** the target component can receive either an `MSOK` (Transition 17) or `MSNOK` (Transition 18) from its child stub(s). Depending on whether the target component is the MSDM for the current scenario and the response from the child stub(s), four alternative transitions 19-22 can be fired subsequently. Transition 19 means that the target component is the MSDM and it receives at least one `MSNOK`, hence it propagates an `MSD` to its Type A child stubs by transitions 26 and 27, and then leaves transition state by Transition 28. Transition 21 also means that the target component is the MSDM which yet only receives `MSOK`. This leads to the `MSI` propagation to its Type A child stubs by transitions 29-31. Transitions 20 and 22 are fired when the target component is not the MSDM. Transition 20 is taken, i.e. the target component sends an `MSNOK` to the parent stub, if the target component receives at least one `MSNOK` and it is followed by Transition 25 as the target component receives an `MSD` from the parent stub. Transition 22 is taken as the target component sends an `MSOK` to the parent stub. This is followed by Transition 23, where the target component receives an `MSI` from the parent stub. After the `MSI` propagation to the Type A child stubs, the target component can receive `MSC` from a child stub via Transition 32. After that, the target component can leave transition state by Transition 33 by sending an `MSC` to the parent stub to indicate mode switch completion (if the target component is not the MSDM), or by Transition 34 (if the target component is the MSDM).

Transitions 35-38 are taken only when the target component has no Type A subcomponents. Therefore, the target component will only interact with the parent stub. Furthermore, the initial handling of an `MSR` is realized by transitions 1-6. Transition 3 represents a self-triggered scenario which always leads to the `MSR` transmission to the parent stub by Transition 6. Transitions 1 and 2 represent a scenario from a child stub which could be approved (Transition 4), or rejected (Transition 5), or forwarded to the parent stub (Transition 6). The MSR/MSQ queue updating rule is implemented in the functions *updateQueue()* and *updateQueue2()*, which can be found in transitions 33, 34 and 38. If the target component is the MSDM (i.e. a Type B component), *updateQueue2()* is applied so that the target component does not remove a pending `MSR` issued by itself. Otherwise, *updateQueue()* is applied which also removes any pending `MSR` issued by the target component.

6.2 Model equivalence between a component and a stub

In order to ensure that a parent stub indeed behaves as a real parent and a child stub indeed behaves as a real child, we need to prove the equivalence between a stub and the corresponding component. A composite component

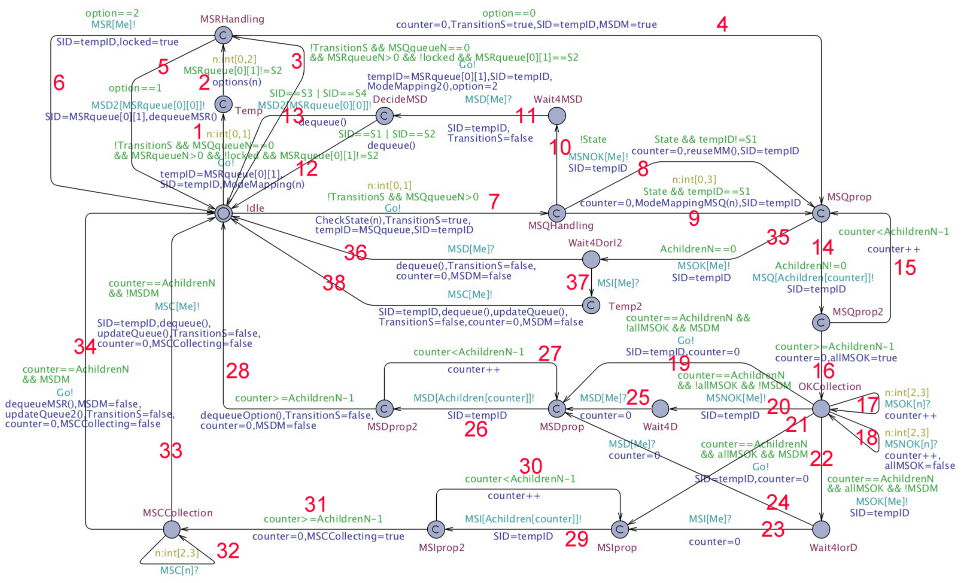
Figure 10: The UPPAAL model for the target component

should be equivalent to a parent stub for its subcomponents. A non-Top component should be equivalent to a child stub for its parent. Our focus here will be on Case (4) since a non-Top composite component can take both roles, either being a parent stub for its subcomponents or being a child stub for its parent. In Case (4), the complete behavior of the target component is jointly represented by the automata in figures 10 and 7. Basically, the model of the target component includes the interaction (i.e. exchanging primitives) with the parent and subcomponents, as well as local computations. If the target component is considered as a parent stub, its observable behavior should only be the interaction with a single subcomponent. Likewise, if the target component is considered as a child stub, its observable behavior should only be the interaction with the parent. Therefore, the observable behavior of the target component should be extracted from its complete model and then compared with the corresponding stub.

First let's consider the target component as a parent stub for one of its subcomponents. Then the observable behavior of the target component is only the interaction with this subcomponent. The first step of extracting the observable behavior of the target component is to remove all the guards, assignments as well as transitions related to the primitive exchange with the parent stub. Also, since a single subcomponent is unaware of the interaction between the target component and its other subcomponents, the MSQ/MSI/MSD propagation in the model of the target component can be changed to the single transmission of a primitive to an individual subcomponent. After this transformation, the model of the target component (see Figure 10) will be simplified to the model depicted in Figure 11. We can further remove some redundant committed locations in Figure 11 and combine it with the model in Figure 7. In addition, transitions 7, 8 and 13 in Figure 11 are actually used to receive a primitive from all the Type A subcomponents of the target component, thus can be changed while considering a single subcomponent. Figure 12 depicts the model of the target component after the second step of transformation. Comparing Figure 12 and Figure 8, we can conclude that the simplified model of the target component is equivalent to a parent stub for a single subcomponent. It should be noted that Transition 2 in Figure 12 is only taken if an `MSR` is pending after Transition 1.

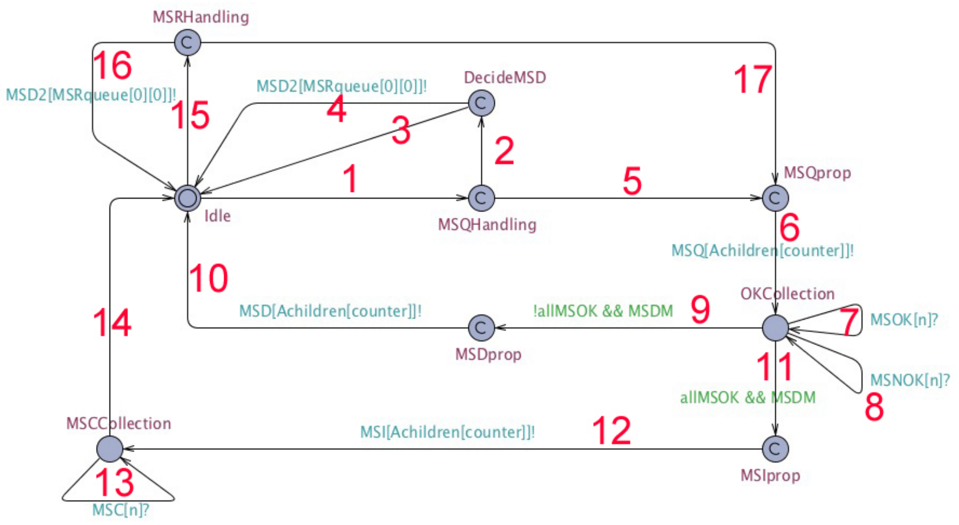

Figure 11: Manual model transformation from the target component to a parent stub (Step 1)

For the parent of the target component, the target component will take the role of a child stub. Similarly, by removing all the guards, assignments as well as interactions with the subcomponents, the model of the target component can be reduced to the model shown in Figure 13. The model in Figure 13 can be further simplified by removing some redundant transitions and combining the model in Figure 7. The model after this transformation is depicted in Figure 14. According to Figure 14, some redundant transitions can be found. For instance, as a child stub, transitions 4 and 5 are essentially the same as transitions 6-9. Transitions 14-17 are essentially the same as transitions 10-13. After removing these redundant transitions, we can compare the simplified model in Figure 15 with the model of a child stub in Figure 9, which are equivalent with regard to the interaction with the parent.

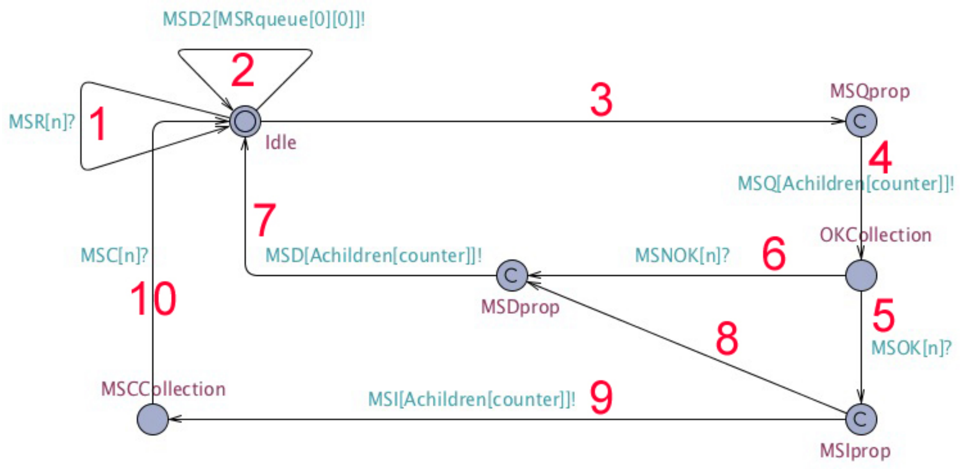

Figure 12: Manual model transformation from the target component to a parent stub (Step 2)

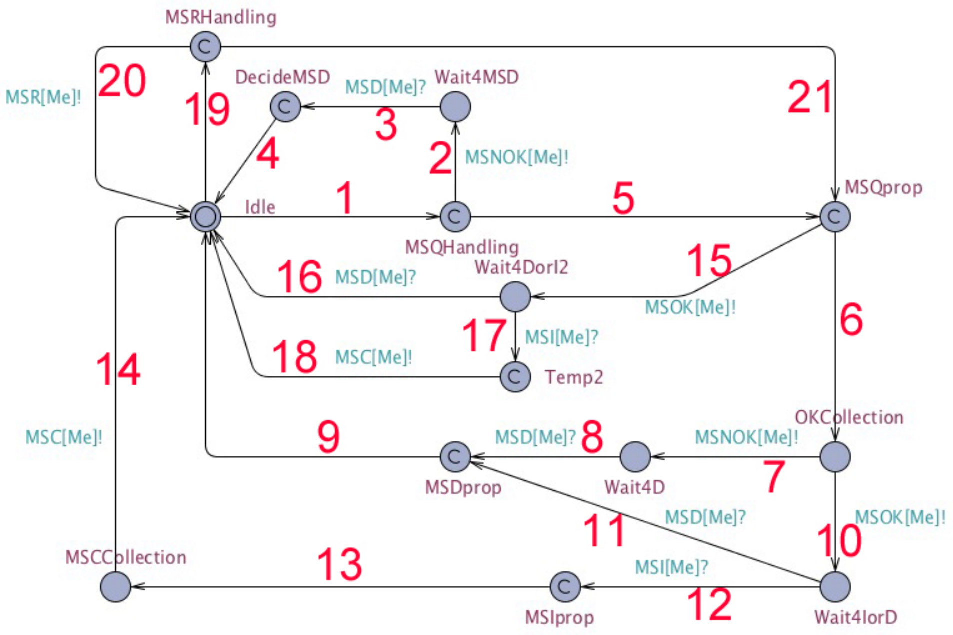

Figure 13: Manual model transformation from the target component to a child stub (Step 1)

6.3 Verification

Based on the UPPAAL model described in Section 6.1, Property 1 can be easily verified by the UPPAAL verifier, where Property 1 can be expressed

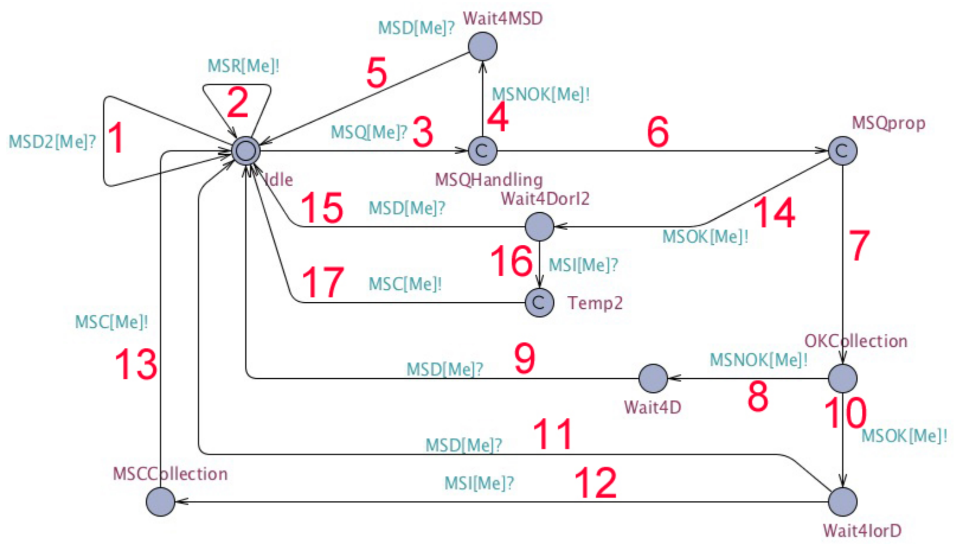

Figure 14: Manual model transformation from the target component to a child stub (Step 2)

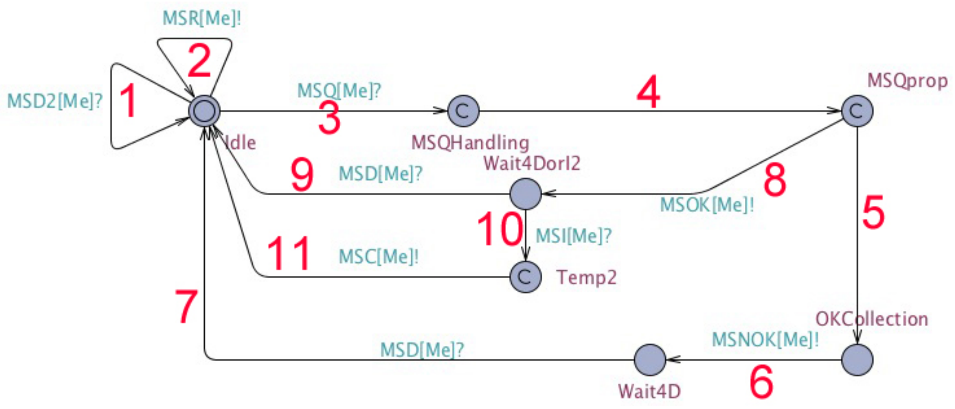

Figure 15: Manual model transformation from the target component to a child stub (Step 3)

by *A[] not deadlock*. Table 4 shows the verification time[1] for each case, yet without considering Case (4).

All the five cases in Table 4 are proved to be deadlock-free with reasonable verification time. The verification for Case (4) demands for more efforts in that UPPAAL ends up with memory exhaustion due to the growing model complexity. This problem is remedied by two means. The first is to use *Compact Data Structure* in UPPAAL for state space representation

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

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

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

instead of the default representation DBM (Difference Bound Matrices). A direct consequence of this is longer verification time but much less memory consumption. In addition, we divide the model for Case (4) into four simpler models that can be verified separately. Case (4) considers $c_i \in \widetilde{CC}$ which is an MSS. Each scenario k triggered by c_i can affect $SC^A_{c_i}(k)$. Since c_i has two subcomponents (e.g. c_1 and c_2) in the model, k can lead to four sub-cases concerning $SC^A_{c_i}(k)$: (a) $T^k_{c_1} = T^k_{c_2} = A$; (b) $T^k_{c_1} = A, T^k_{c_2} = B$; (c) $T^k_{c_1} = B, T^k_{c_2} = A$; (d) $T^k_{c_1} = T^k_{c_2} = B$. Cases (b) and (c) are symmetrical, hence it is sufficient to only consider three cases, e.g. (a), (b) and (d). Using Compact Data Structure and taking (a), (b) and (d) as three sub-cases of Case (4), we successfully verified the deadlock-freeness for Case (4), with the verification time summarized in Table 5.

Table 5: Verification time of deadlock-freeness: Case (4) (using Compact Data Structure)

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

Property 2 cannot be directly verified by the UPPAAL models made for the verification of Property 1 except for cases (1) and (5). The reason is that these models allow the triggering of a scenario at any time if it is possible. Since the conflict handling mechanism assigns higher priority to MSQ queue, if the parent stub keeps sending a msq^k (not associated with the `MSR` sent from the child stubs) to the target component, a pending $msr^{k'}$ from a child stub which is Type B for k may never be handled. This problem should not exist in a real-world system because the triggering of a scenario is usually not a frequent event. In order to make Property 2 verifiable, we slightly modify the parent stub for cases (2), (3) and (4) such that for every two consecutive `MSQ` primitives (msq^k and $msq^{k'}$) sent by the parent stub, at least either k or k' originates from a child stub. The modified parent stub is shown in Fig. 16. Since this modification does not alter the conflict handling mechanism, the modified models can be used to verify both properties. For Case (6), we modify the target component so that each two consecutive scenarios triggered by itself must be separated by the handling of a scenario coming from a child stub. Since Case (1) only considers scenarios from the parent stub while Case (5) only considers scenarios from the child stubs, they do not need any modification.

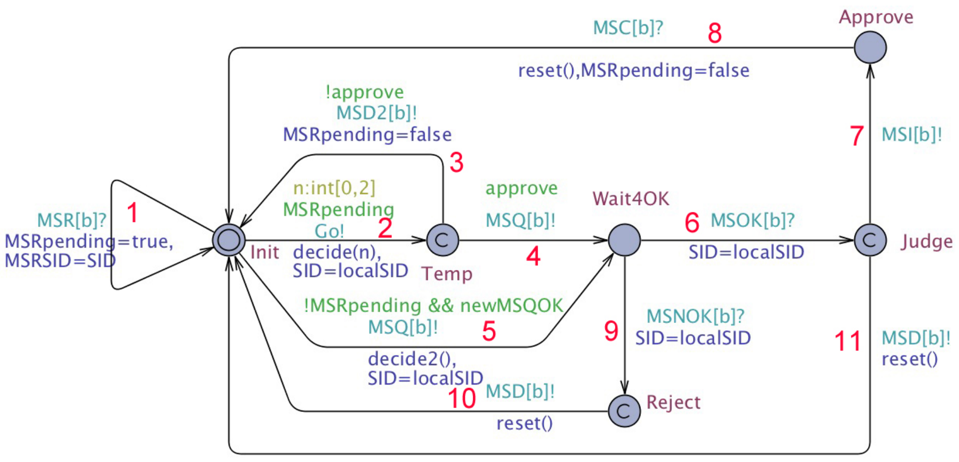

Figure 16: The UPPAAL model for the modified parent stub

Based on the modified UPPAAL models, Property 1 and Property 2 are both satisfied for cases (1)-(6). Property 2 splits into three sub-properties:

- Property 2.1—*MSQqueueN>0–>MSQqueueN==0*: A scenario from the parent stub can eventually be handled. *MSQqueueN* is the number of elements in the MSQ queue of the target component. When *MSQqueueN* is greater than 0, it will eventually become 0.

- Property 2.2—*pending–>!pending*: A scenario triggered by the target component itself can eventually be handled. *pending* is a boolean variable set to true when the target component triggers a scenario, and set to false when it is completely handled by the target component.

- Property 2.3—*ChildStub1.MSRpending–>!ChildStub1.MSRpending*: A scenario from a child stub can eventually be handled. *MSRpending* is a boolean variable set to true when a child stub triggers a scenario, and set to false when it is completely handled by the child stub.

Table 6 summaries the verification times of properties 1 and 2, excluding Case (4). Note that some sub-properties of Property 2 is only valid for certain cases. For instance, Property 2.2 is only valid when the target component is an MSS, i.e. for cases (2), (4) and (6).

Table 6: Verification times of properties 1 and 2, excluding Case (4)

Case	Property 1	Property 2.1	Property 2.2	Property 2.3
(1) $c_i \in PC, c_i \neq MSS$	0.002s	0.003s	–	–
(2) $c_i \in PC, c_i = MSS$	0.009s	0.008s	0.009s	–
(3) $c_i \in \widetilde{CC}, c_i \neq MSS$	25.14s	22.361s	–	33.915s
(5) $c_i = Top, c_i \neq MSS$	0.268s	–	–	0.283s
(6) $c_i = Top, c_i = MSS$	3.585s	–	3.514s	4.143s

The verification of properties 1 and 2 for Case (4) is carried out for the sub-cases listed in Table 5, using Compact Data Structure. The verification times for each case are summarized in Table 7.

Table 7: Verification times of properties 1 and 2 for Case (4) (using Compact Data Structure)

Sub-case	Property 1	Property 2.1	Property 2.2	Property 2.3
(a) $T^k_{c_1} = T^k_{c_2} = A$	385.772s	385.793s	617.648s	589.801s
(b) $T^k_{c_1} = A, T^k_{c_2} = B$	429.765s	446.608s	709.125s	700.909s
(d) $T^k_{c_1} = T^k_{c_2} = B$	240.807s	239.164s	419.807s	398.801s

Apart from properties 1 and 2, we additionally verified the maximum sizes of MSQ and MSR queues, as indicated in Table 3. However, our UPPAAL models assume that the handling of a primitive for each component is atomic and instantaneous, as the handling of a primitive in UPPAAL can be realized by committed states such as **MSIprop** and **MSIprop2** in Figure 10. This assumption is actually not required by the conflict handling mechanism, nevertheless, without such an assumption, many committed states in our UPPAAL models must be replaced with normal states. As a consequence, the verification for each sub-case of Case (4) will end up with state explosion, even if Compact Data Structure is used. Due to this assumption, the maximum number of elements in the MSQ queue is 1 but not 2. In our models, the maximum size of the MSQ queue is set to be 1 and the maximum size of the MSR queue is defined based on Table 3. The verification results show that the maximum number of elements can be reached for both MSQ and MSR queues, yet cannot be exceeded, otherwise an error will occur during the verification.

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

7 Related work

The extended MechatronicUML (EUML) [8] by Heinzemann et al. is currently the most closely related work to our MSL. In EUML, each component has reconfiguration ports which resemble the dedicated mode switch ports of our mode-aware component model. Each composite EUML component internally has a manager and executor which play the same role as our mode switch runtime mechanism. Components can propagate messages for requesting reconfiguration or executing reconfiguration. The propagation of these messages can be compared with our MSP protocol. However, the MSP protocol highly relies on the mode mapping between components whereas the message propagation in EUML is controlled by some reconfiguration rules defined in the manager of each component. EUML requires that reconfiguration should be performed bottom-up; this is similar to our mode switch dependency rule which yet allows concurrent reconfigurations of different components. The major difference between EUML and MSL is that EUML focuses more on component reconfiguration while mode is not addressed. In general, MSL is relatively more mature since EUML is more recently developed. EUML is also aware of the transmission of multiple request messages, which is comparable to the triggering of multiple scenarios described in this paper, however, no concrete solutions have been provided yet.

Another recent work related to MSL is the oracle-based approach by Pop et al. [9]. The basic idea is to abstract component behaviors into a property network spread throughout the component hierarchy. The mode of each component is modeled as a property and mapped from a set of properties to their valuations. A single property change can be propagated throughout the property network, potentially leading to the valuation change of other properties. And then the new mode of each component can be derived after the update of the property network. A finite-state machine called Oracle is offline constructed to guarantee predictable update time of the property network. The construction of Oracle implies that the mode switch handling requires global information of the property network. In contrast, MSL is fully distributed, requiring no global information.

Mode switch has been addressed in a number of component models, such as SaveCCM [10], COMDES-II [11] and MyCCM-HI [12], to name a few. There are also some other component models which have been commercialized, e.g. Koala [13] (targeting consumer electronics) and Rubus [14] (targeting ground vehicles). These component models have different notions of mode switch handling. In Koala and SaveCCM, a special *switch* is introduced to achieve the structural diversity of a component. Depending on the input data, *switch* can select one of multiple outgoing connections. COMDES-II uses a state-machine component to switch component configurations in different modes. In Rubus, mode is treated as a system property. A system-wide static configuration of components is defined for each mode. MyCCM-HI provides a more advanced mechanism for handling mode switch. Each MyCCM-HI component is mode-aware and is associated with a mode automaton which implements its mode switch mechanism. In addition, mode switch is also addressed by languages such as the Architecture Analysis & Design Language [15], where a state machine is used to represent the mode switch behavior of a component. Each state machine consists of a number of states (modes), transitions between these states (mode switches) and input/output event ports used for mode switch triggering. Compared with MSL, none of these works provide any systematic strategy to coordinate the mode switches of different components, due to the common assumption of independent mode switches between components.

8 Conclusion

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

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

Acknowledgment

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

References

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

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

[3] P. Hošek, T. Pop, T. Bureš, P. Hnětynka, and M. Malohlava, "Comparison of component frameworks for real-time embedded systems," in *Component-Based Software Engineering*, vol. 6092 of *Lecture Notes in Computer Science*, 2010.

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

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

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems.* Addison Wesley, 1987.

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

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

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

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

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

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

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

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

[15] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (AADL): An introduction," Tech. Rep. CMU/SEI-2006-TN-011, Software engineering institute, MA, Feb. 2006.