A mode mapping mechanism for component-based multi-mode systems

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

***Abstract*—Component-Based Development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable components. A classical approach to reduce embedded systems design and run-time complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems. In addressing this issue, we previously developed a Mode Switch Logic (MSL) for component-based multi-mode systems. Our MSL implements seamless coordination and synchronization of mode switch in systems composed of independently developed components. However, our original MSL is based on the, in a setting of reusable components, unrealistic assumption, that all the components of a system support the same modes. This considerably limits the feasibility of our MSL. In this paper we lift this assumption and propose a mode mapping mechanism that enables assembly of components supporting different sets of modes. We demonstrate our mode mapping mechanism by a simple example application.**

***Index Terms*—component-based, multi-mode, mode switch, mode mapping**

I. Introduction

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

There are a variety of alternatives for design and development of multi-mode systems. We set our focus on Component-Based Development (CBD), a promising solution for the development of embedded systems. CBD boasts quite a number of appealing features such as complexity management, increased productivity, higher quality, faster developing time, lower maintenance costs and reusability [1]. What we appreciate most is the component reuse, which allows us to build a system from reusable components, i.e. a system does not have to be developed from scratch, instead, some of its components or subsystems may be directly obtained from a repository of pre-developed components.

Our target is component-based multi-mode systems (CBMMSs), i.e. multi-mode systems built by a set of hierarchically organized components. Different system behaviors in different modes are characterized by different:

- Component availability. Some components can be activated in one mode and deactivated in another mode.
- Component connection. Component connections can be changed in different modes.
- The mode specific behavior of a component. Some components themselves have different behaviors in different modes.

Figure 1 illustrates a simple CBMMS (used throughout the paper). From the top level, the system consists of Component a (which is further decomposed into Component c, d and e) and b (which is further decomposed into Component f and g). Besides, the system and its components all support two modes: $m1$ and $m2$. In mode $m1$, Component g is deactivated (invisible) and Component f has one mode-specific behavior (indicated by black color). In mode $m2$, Component g becomes activated while Component d is deactivated. Besides, Component f has another mode-specific behavior (indicated by grey color). Due to the availability change of Component d and g, component connections are different in these two modes, although component connection is not our concern in this paper.

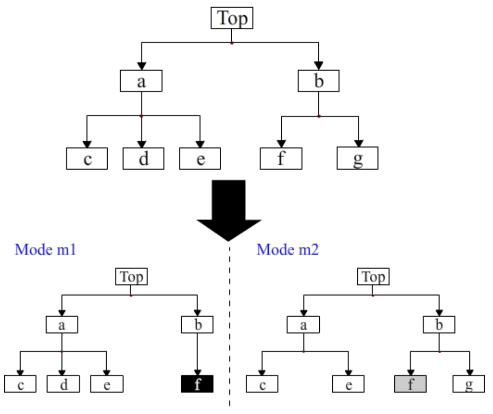


Fig. 1. A component-based multi-mode system

Apparently, the system behavior is highly dependent on its components. This dependency also holds during a mode switch. The central issue is that the mode switches of different components must be coordinated and synchronized to achieve a successful and efficient global mode switch. Notwithstanding that there is plenty of research work dealing with mode switch, little attention has been paid to this composable mode switch problem. We have developed a Mode Switch Logic (MSL) for CBMMSs [2], and based on our MSL, we have also provided timing analysis for a composable mode switch [3]. The correctness of our MSL has been verified by model-checking using UPPAAL [4]. Up till now, our MSL has been constrained by an unrealistic assumption that all the components support the same modes. However, when a composite component is built by a composition of reusable components, chances are that the desired modes of this composite component are different from the supported modes of its subcomponents, and the subcomponents would probably support different modes. This mode incompatibility is a challenge for component composition and our MSL. In this paper, we introduce a mode mapping mechanism to overcome this problem.

A. Related work

Mode switch problems (sometimes also called ”Mode change”) can be found in a multitude of related ongoing research on miscellaneous topics, a majority of which delve into multi-mode real-time systems, in particular the study of mode switch protocols and scheduling issues during mode switch. One of the earliest publications related to mode switch is by Sha et al. [5], who developed a simple mode switch protocol in a prioritized preemptive scheduling environment guaranteeing short and bounded mode switch latency. Later Real and Crespo [6] conducted a survey of different mode switch protocols and proposed several new protocols along with associated schedulability analysis. Protocols for symmetrical multiprocessor platforms are presented in [7], and extended to uniform multiprocessor platforms in [8]. There are also a number of papers, e.g. [9] and [10], targeting mode switch schedulability analysis, including EDF scheduling for multi-mode real-time systems [11] [12] [13].

In addition, Phan et al. study multi-mode real-time systems from a different perspective. They extend the traditional Real-Time Calculus (RTC) into multi-mode RTC to determine typical system properties [14]. They also present a multimode automaton model for modeling multi-mode applications and an interface-based technique for their compositional analysis [15]. Their most recent work presents a semantic framework for mode switch protocols [16].

Several frameworks have been developed for the support of multi-mode systems, such as COMDES-II [17] and MyCCM-HI [18]. Moreover, mode switch can be supported by a few programming languages/models, such as AADL [19], Giotto [20] and TDL [21] (implemented in the Ptolemy II framework [22]).

No related work has been found with respect to mode mapping, although Pedro briefly mentions mode mapping in his PhD dissertation [23], yet without any detailed discussion.

B. Contribution

The contribution of this paper is that we propose a mode mapping mechanism as a supplement to our MSL for CB-MMSs. This mechanism enables us to make mode mapping rules for each composite component. The Mode Switch Request (MSR) propagation mechanism of our MSL can be assisted by these rules so that the right *MSR* primitive can always be propagated to the right component even when different components support different modes.

The rest of this paper is organized as follows: In Section II, we give a general introduction of our MSL, especially component configuration and the MSR propagation mechanism. Section III offers a thorough explanation of our mode mapping mechanism and Section IV describes how to implement our mode mapping mechanism in a typical CBMMS, a path finding car. Finally, we make our conclusion in Section V.

II. Mode Switch Logic (MSL) for component-based multi-mode systems

Figure 1 illustrates what a typical CBMMS looks like. Such a system bears two distinctive features. First, it is built in a component-based manner. Therefore, it usually has a hierarchical structure and is composed of both primitive and composite components at different hierarchical levels. In the realm of CBD, a composite component (e.g. Component *a* in Figure 1) consists of other components (denoted subcomponents) whilst a primitive component (e.g. Component *c* in Figure 1) cannot be further decomposed. Second, the system supports multiple operational modes and can switch from one mode to another mode under certain circumstances. In each mode, each component has its own configuration and each mode switch of the system corresponds to the mode switches of certain components. Since the mode switches of these components are not independent, they must be guided and coordinated by some rules to achieve a successful global mode switch of the system. These rules can be found in our MSL, which currently includes three facets: (1) component reconfiguration, (2) MSR propagation mechanism, and (3) mode switch dependency rules. Since our mode mapping mechanism has a major influence on the MSR propagation, we shall give a brief introduction to (1) and (2), but refer to [2] for information about (3).

A. Component reconfiguration

Using our MSL, mode switch support is added to both primitive and composite components. Based on traditional primitive and composite components, which do not support multiple modes, we introduce specific ports dedicated to mode switch, define component configuration for each mode and integrate the MSL into each component. The mode switch related communication between different components is realized by communication over those dedicated ports. Some components may reconfigure themselves during mode switch and this is controlled by the MSL of each component. Component configuration varies between primitive and composite components. For primitive components, component configuration is defined by the component running status (activated/deactivated) and the mode-specific behavior. For composite components, component configuration is defined by the component running status, the activated subcomponents and the active inner component connections. In each mode, only activated components are running, while deactivated components are temporarily unavailable. Likewise, in each mode, only active component connections are considered. A connection becomes inactive when it is disconnected due to a mode switch. When a component starts its mode switch, it will reconfigure itself by changing the aforementioned elements for each configuration.

B. MSR propagation mechanism

Before we explain our MSR propagation mechanism, it is necessary to introduce Mode Switch Request (MSR) and Mode Switch Triggering Source (MSTS).

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

In our previous work [2], we made an assumption that all components support the same modes. In that case, the target mode of each *MSR* will be exactly the same for all components. This assumption simplifies the mode switch analysis, but is unrealistic in a component-based setting. In the next section, we will lift this assumption.

Definition 2. *The Mode Switch Triggering Source (MSTS) is the component who initiates a MSR. The MSTS could be either a primitive or composite component.*

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

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

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

For a composite component, the only difference is that it needs to send a *MSR* to itself if it is a MSTS, and that it is sensitive to where the current *MSR* comes from:

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

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

1) Component c sends a *MSR* to its parent a and to itself.
2) Component a propagates the *MSR* to d and e. It also sends the *MSR* to the top level component.
3) The top level component sends the *MSR* to b.
4) Component b broadcasts the *MSR* to f and g.

Our MSR propagation mechanism both guarantees that all the components are notified by the same *MSR*, and avoids any potential redundant *MSR* transmission.

III. The mode mapping mechanism

Our MSR propagation mechanism is able to distribute each *MSR* to all the components efficiently. However, it is assumed that all components support the same modes. For instance, in Figure 1, when the system is in mode $m1$, all the components must be in mode $m1$ as well. Mode becomes a global knowledge for all components. As a result, the target mode of any *MSR* originating from a specific *MSR* generated by the same MSTS is always the same. In a setting with reusable components, this assumption is unrealistic, since the requirement for all components to support the same modes substantially limits reuse of components across applications. Fortunately, this assumption can be lifted by our mode mapping mechanism.

A. The motivation of mode mapping

When we build Component a by the primitive components c, d and e, the supported modes of c, d and e are already provided by component developers. The supported modes of a can be either obtained from system requirements at higher levels or figured out according to the expected functionality of a. Table I presents a list of supported modes within Component a. The list indicates that both the supported modes and number of supported modes are different for different components. We call this the mode incompatibility problem. Due to this problem, our current MSR propagation mechanism will not be sufficient because the target mode conveyed by the *MSR* associated with any MSTS cannot be recognized by other components. The main purpose of our mode mapping mechanism is to give a clear mapping between the supported modes of different components and extend our MSR propagation mechanism to handle mode incompatibility.

Component	Supported modes
a	$m1a$, $m2a$
c	$m1c$, $m2c$, $m3c$
d	$m1d$
e	$m1e$, $m2e$, $m3e$, $m4e$

TABLE I
SUPPORTED MODES WITHIN COMPONENT a

Within a composite component, mode incompatibility can exist between the parent and its children, or between different subcomponents. We are particularly interested in the former case. For a composite component x and one of its subcomponents y, we distinguish two basic types of mode incompatibility:

1) Multiple modes of x is mapped to only one mode of y. Then y stays in the same mode when x is switching between those modes.
2) One mode of x is mapped to multiple modes of y. Then x stays in the same mode when y is switching between those modes. We call this phenomenon *Local mode switch*.

These two types of mode incompatibility can co-exist within the same composite component.

A mode mapping can be intuitively presented by a *Mode Mapping Table*. Table II displays the mode mapping within Component a. Each row consists of the supported modes of one component and each column corresponds to a mode mapping between different components. For instance, when a is in mode $m1a$, c, d and e are in $m1c$, $m1d$ and $m1e$, respectively. Both types of mode incompatibility can be found in Table II. The first type comes from a and d. When a is in $m1a$ or $m2a$, d is always in $m1d$. The second type comes from a and c or e. For example, when a is in $m2a$, c is in $m2c$ or $m3c$.

Component	Supported modes			
a	$m1a$	$m2a$		
c	$m1c$	$m2c$		$m3c$
d	$m1d$			
e	$m1e$	$m2e$	$m3e$	$m4e$

TABLE II
MODE MAPPING TABLE FOR COMPONENT a

A system should always start by running in a safe and predictable mode. This must be consistent with the mode mapping. For instance, according to the mode mapping in Table II, if $m1a$ is defined as the initial mode of Component a, then $m1c$, $m1d$ and $m1e$ should be the initial modes of c, d and e, respectively.

B. Local mode switch

When a MSTS triggers a *MSR*, the *MSR* will be propagated to all the other components. In this sense, any *MSR* has a global impact on the entire system. Some components may not switch mode upon receiving a *MSR*, but it is notified anyway. In the worst case, a component at the bottom level initiates a frequent *MSR* which only triggers its own mode switch and is ignored by all the other components, consequently lowering the MSR propagation efficiency. Since only some components are responsive to the *MSR*, MSR propagation becomes a local activity:

Definition 3. *A local mode switch is a mode switch that only takes place within a certain component excluding the top component. A local MSR is initiated within this component and this component should not propagate this MSR to its own parent. Components who do not receive this local MSR are unaware of this local mode switch.*

Now let's go back to the two types of mode incompatibility mentioned in Section III-A. The second type actually implies the existence of local mode switch. We can still use the mode mapping in Table II to explain local mode switch. The mode switch between $m2e$ and $m3e$ of Component e is a local mode switch, where e is the MSTS that sends a *MSR* to a and then a terminates the MSR propagation. Hence no other components are affected. Another local mode switch involves both c and e. $m2c$ is mapped to $m2e$ and $m3e$ while $m3c$ is mapped to $m4e$. When c switches between $m2c$ and $m3c$, e will also switch mode. At least one of them propagates a *MSR* which will be first sent to a, which will only propagate it to either c or e. This is considered to be a local mode switch of a.

The consideration of local mode switch can avoid unnecessary MSR transmission and ameliorate MSR propagation efficiency. Once the second type of mode incompatibility is detected, any composite component is able to distinguish any local mode switch activity based on its own local mapping. Then the composite component should not propagate the *MSR* to its parent. Similarly, the composite component can also decide not to propagate the *MSR* to some of its subcomponents or simply terminate MSR propagation.

C. Dominant Default Modes

The *Mode Mapping Table* is a representative expression of the mode mapping within a composite component. Nonetheless, it fails to address the unpredictable mode switch problem. When a switches to $m2a$, c has two potential target modes and it can either switch to $m2c$ or $m3c$. This unpredictable behavior is undesirable as it causes mode switch uncertainty. Therefore, we introduce Dominant Default Mode (DDM):

Definition 4. *The Dominant Default Mode (DDM) of a component during a mode switch is its default target mode out of multiple possible target modes upon receiving a MSR.*

DDM plays an essential role in eliminating unpredictable mode switch behaviors. The DDM of each component actually corresponds to one mode switch scenario. A mode switch scenario is specified by the mode switch of a MSTS x from a given mode $m1x$ to another given mode $m2x$. Different combinations of x, $m1x$ and $m2x$ give rise to different mode switch scenarios. Therefore, a component can have different DDMs associated with different mode switch scenarios and DDM declaration is an essential part of the mode mapping rules.

D. The mode mapping solution

A global mode mapping known by all components would be against the principles of CBD. Instead, our solution is to distribute the mode mapping mechanism to each composite component, which will manage its own local mode mapping. A composite component only knows the mode mapping between its subcomponents and itself. Figure 2 illustrates how the mode mapping mechanism functions within Component a. The local mode mapping is managed by a. During any *MSR* transmission between a and its subcomponents, the target mode in the *MSR* is converted to a new target mode according to the mode mapping rules and then sent to the target component. Therefore, the mode mapping mechanism guarantees that both a and its subcomponents can recognize their respective target modes upon receiving a *MSR*.

In [2], we define primitive and composite components as two different tuples based on the assumption that they support the same modes. Here we extend them by considering mode incompatibility, although we will here only include mode

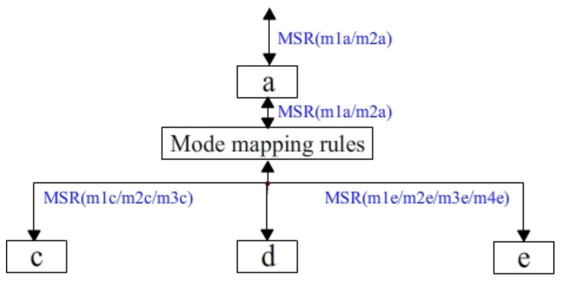


Fig. 2. The mode mapping mechanism of Component a

related items of the tuples. Let PC and CC denote the set of primitive and composite components, respectively. Then,

Definition 5. *Each Primitive Component $c \in PC$ is a tuple:*

$$< \mathrm{M}, m^0, \mathrm{m} >$$

where M is the set of supported modes of c; m^0 is the initial mode of c and m is the current mode of c.

Let's take Component *c* in Table I as an example. When in mode *m3c* and if *m1c* is the initial mode, then *c* is defined by the tuple:

$$< \{m1c, m2c, m3c\}, m1c, m3c >$$

Since mode mapping is managed by a composite component, the composite component must know the modes of itself and its children (i.e. its subcomponents at one level down in the hierarchy), as well as the mappings between these modes.

Definition 6. *Each Composite Component $k \in CC$ is a tuple:*

$$< \mathrm{M}, m^0, \mathrm{m}, \mathrm{SC}, M_{SC}, m^0_{SC}, m_{SC}, \mathrm{MM} >$$

where M is the set of supported modes of k; m^0 is the initial mode of k; m is the current mode of k; SC is the set of subcomponents of k; $M_{SC} : SC \rightarrow 2^{\mathcal{M}}$ is a function mapping each subcomponent of k to the corresponding supported modes ($\mathcal{M}$ is the set of all modes); the mapping between each subcomponent and its initial mode is defined by a function $m^0_{SC} : SC \rightarrow M_{SC}$, where for a subcomponent x, we require $m^0_{SC}(x)$ to be one of the supported modes of x, i.e. $m^0_{SC}(x) \in M_{SC}(x)$; likewise, $m_{SC}(y)$ returns the current mode of subcomponent y; MM is the mode mapping information within k, given by a set of synchronized Mode Mapping Automata (MMAs) each defined by a tuple

$$< S, s^0, \mathit{IL}, \mathit{OL}, T >$$

where S is a set of states, s^0 is an initial state, IL and OL are sets of input and output labels, and T is a set of transitions ($T \subseteq S \times \mathit{IL} \times 2^{\mathit{OL}} \times S$). For a composite component k*, there is one MMA handling the mapping at the interface with other parts of* k *and one MMA for each of the children of* k*. The states of these automata will correspond to the modes of the corresponding components, with the the initial mode corresponding to the initial state. The transitions of the automata will specify and implement the mode mapping by accepting and emitting MSR primitives and by synchronizing with the other automata (a transition can only be taken if it is taken jointly with a transition with the same label in one of the other automata). The formal semantics of this is straightforward to define, but omitted here due to space limitations.*

As an illustration, consider Component *a* in Table I defined by the tuple:

$$< M_a, m^0{}_a, m_a, SC_a, M_{SC_a}, m^0_{SC_a}, m_{SC_a}, MM_a >$$

where

$$\begin{aligned}
M_a &= \{m1a, m2a\} \\
m^0{}_a &= m1a \\
m_a &= m2a \\
SC_a &= \{c, d, e\} \\
M_{SC_a} &= \{c \rightarrow \{m1c, m2c, m3c\}, \\
&\quad\; d \rightarrow \{m1d\}, \\
&\quad\; e \rightarrow \{m1e, m2e, m3e, m4e\}\} \\
m^0_{SC_a} &= \{c \rightarrow m1c, d \rightarrow m1d, e \rightarrow m1e\} \\
m_{SC_a} &= \{c \rightarrow m3c, d \rightarrow m1d, e \rightarrow m4e\}
\end{aligned}$$

and MM_a is defined by the MMA in Figure 3 - 6.

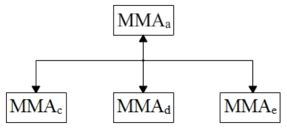
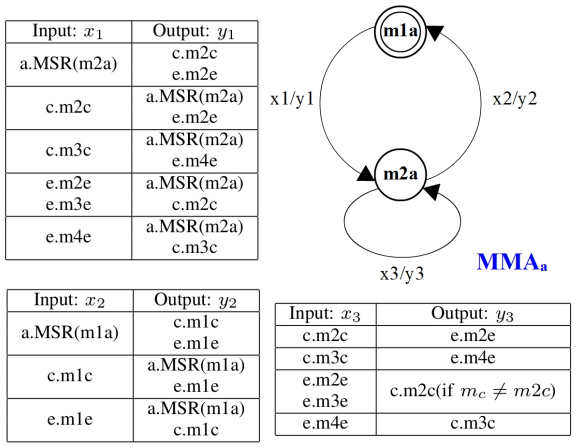


Fig. 3. The set of mode mapping automata of Component *a*



Input: x_1	Output: y_1
a.MSR(m2a)	c.m2c e.m2e
c.m2c	a.MSR(m2a) e.m2e
c.m3c	a.MSR(m2a) e.m4e
e.m2e e.m3e	a.MSR(m2a) c.m2c
e.m4e	a.MSR(m2a) c.m3c

Input: x_2	Output: y_2
a.MSR(m1a)	c.m1c e.m1e
c.m1c	a.MSR(m1a) e.m1e
e.m1e	a.MSR(m1a) c.m1c

Input: x_3	Output: y_3
c.m2c	e.m2e
c.m3c	e.m4e
e.m2e e.m3e	c.m2c(if $m_c \neq m2c$)
e.m4e	c.m3c

Fig. 4. The mode mapping automaton of Component *a*

Figure 3 shows the set of MMAs of Component *a*. The MMA structure is hierarchically organized in the same way as the corresponding components, although it does not include any potential substructuring of the subcomponents, and is fully contained in the composite component (*a* in this case). Figure 4, 5 and 6 illustrate the MMA of *a*, *c* and *e* respectively. MMA$_d$ is not described because Component *d* only supports

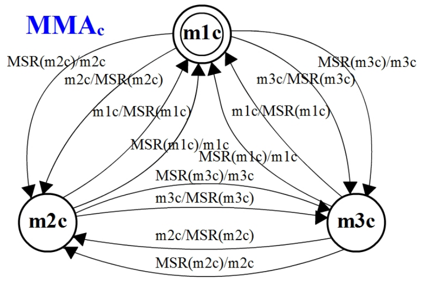


Fig. 5. The mode mapping automaton of Component c

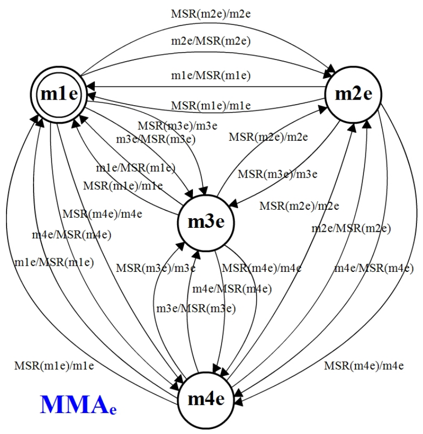


Fig. 6. The mode mapping automaton of Component e

a single mode and does not switch mode. For each MMA$_x$, each location (state) corresponds to one mode supported by x, with the initial mode marked by double circles. Each transition represents a mode switch and is associated with a condition in the form of *Input/Output*. The *Input* is the signal received by MMA$_x$ and the *Output* is the set of signals sent by MMA$_x$ in response to the *Input*. For both the *Input* and the *Output*, we distinguish two types of signals: the external signal, i.e. a *MSR*, and the internal signal used for the synchronization between different MMAs. For example, according to Figure 3, when a is in $m1a$ and receives *a.MSR(m2a)*, this is an "external" signal recieved by the mode mapping mechanism from a indicating that the new mode of a is $m2a$. Based on the mode mapping rules, MMA$_a$ then generates *e.m2c* and *e.m2e* as its *Output*, which will be received by MMA$_c$ and MMA$_e$ accordingly. Then MMA$_c$ will send *MSR(m2c)* to Component c (see Figure 5). Similarly, MMA$_e$ will send *MSR(m2e)* to Component e. $m2c$ and $m2e$ are actually the DDMs of Component c and e respectively in this scenario. A special case is the local mode switch between $m2e$ and $m3e$ of Component e which must be the MSTS for it. This phenomenon can also be observed from the transitions between $m2e$ and $m3e$ in Figure 6.

E. The algorithm for mode mapping and MSR propagation

We have already introduced a set of mode mapping rules together with their formal expression. These rules can be used by any composite component during MSR propagation. In our previous work [2], we explained the MSR propagation mechanism and implemented it by an algorithm, yet without considering mode incompatibility. Here we extend the MSR propagation algorithm with these mode mapping rules as its input. Since mode mapping is always handled by a composite component, there is no need to implement an algorithm for primitive components. Algorithm 1 can be implemented in composite component x. Some notations and functions are explained as follows:

- *Wait* and *Signal* are primitives for receiving and sending a *MSR*. The *MSR* contains the target mode and the identity of the current sender.
- *convertMode* is a function which derives the new mode of a subcomponent y according to the given mode mapping information. It requires four parameters: the current and new modes of one component (either the parent of y or another subcomponent other than y), the current mode of y and the identity of y.
- *top* is a boolean variable only set to *true* for the top level composite component.

Algorithm 1 $AlgCC.MSR\_propagation(x \in CC, top)$

```
$Wait(MSR(m_x^{new}, origin))$;
if $origin = parent$ then
  $m^{new} = m_x^{new}$;
  for $i$ from 1 to $|SC_x|$ do
    $m^{new} = convertMode(m_x, m_x^{new}, m_{SC_x}(i), SC_x(i))$;
    if $m^{new} \neq m_{SC_x(i)}$ then
      $Signal(SC_x(i), MSR(m^{new}, x))$;
    end if
  end for
else
  $m_{origin}^{new} = m_x^{new}$;
  $m_x^{new} = convertMode(m_{origin}, m_x^{new}, m_x, x)$;
  $m^{new} = m_x^{new}$;
  for $i$ from 1 to $|SC_x|$ do
    if $SC_x(i) \neq origin$ then
      if $m_x^{new} \neq m_x$ then
        $m^{new} = convertMode(m_x, m_x^{new}, m_{SC_x}(i), SC_x(i))$;
      else
        $m^{new} = convertMode(m_{origin}, m_{origin}^{new}, m_{SC_x}(i), SC_x(i))$;
      end if
      if $m^{new} \neq m_{SC_x(i)}$ then
        $Signal(SC_x(i), MSR(m^{new}, x))$;
      end if
    end if
  end for
  if $m_x^{new} \neq m_x$ AND $\neg top$ then
    $Signal(parent, MSR(m_x^{new}, x))$;
  end if
end if
```

Algorithm 1 enables a composite component to decide which components it should propagate the *MSR* to and what are their target modes. The composite component x first identifies where the *MSR* comes from. If it is from its parent, it will derive the target modes of SC_x according to the local mode mapping. A *MSR* is sent to a subcomponent in SC_x only if its target mode is not equal to its current mode.

If the *MSR* is from a subcomponent y of *x*, *x* must first convert the target mode conveyed by the *MSR* to a target mode that it can recognize. Then *x* tries to convert the new mode of its other subcomponents. For each subcomponent *z* other than *y*, if its new mode is different from its current mode, it will receive a *MSR* from *x*. In addition, if the target mode of *x* is different from its current mode and it is not at the top level, it will also propagate the *MSR* to its own parent.

IV. Case study–A path finding car

In this section, our mode mapping mechanism will be implemented in a path finding car. This example is for demonstration purposes, and is substantially simpler than the real applications we have in mind.

The car is equipped with a simple CPU, one light sensor, and two motors. These hardware elements can also be regarded as components (the two motors are treated as one component), which run in different modes:

- As an essential component for the path finding function, the light sensor (called LightS for short) is able to distinguish black and white colors. It supports two modes: $Black$ and $White$, depending on the color it detects.
- The component Motors controls the moving direction of the car. It supports four modes: *Forward*, *Backward*, *Left* and *Right*, which will be simplified as mode *F*, *B*, *L* and *R*. Since the path finding function can be achieved even without moving backwards, we skip the B mode.
- The supported modes of the CPU is in line with the system itself, i.e. *Init*, *Follow*, and *Find*. Here we can consider it as the parent of LightS and Motors.

These components can be defined as the tuples introduced in Section III-D. Due to the limited space, we skip this tuple definition. Here we define $Init$, $White$, and F as the initial modes of CPU, LightS, and Motors. Based on the supported modes of different components, Table III provides the mode mapping, with the following explanations:

- The system (or CPU) is in mode *Init*. It takes three steps for the car to get aligned with the track for the first time (See Figure 7). We assume that the car is initially facing the track and that it is supposed to follow the track clockwise. The first step is to run straight towards the track. When the light sensor detects the black track for the first time, the car will cross the track and then turn left. After that it goes forward and will reach the track again. Since that moment, the system switches to mode $Follow$. During these three steps, the light sensor can be either in *Black* or *White*. The motors can be either in *F* or *L*. Actually the mode mapping between LightS and Motors is quite implicit, as the mode of Motors is not only based on the mode of LightS, but also on some other system parameters.
- The system is in mode *Follow*. As long as the light sensor keeps detecting the black color, the car will keep following the track by moving forward.
- The system is in mode *Find*. This means the car runs out of the track. The light sensor is in *White*. In order to go back onto the track, the car must either turn left or right. Therefore, the motors run either in *L* or *R*. Here we assume L is the DDM.

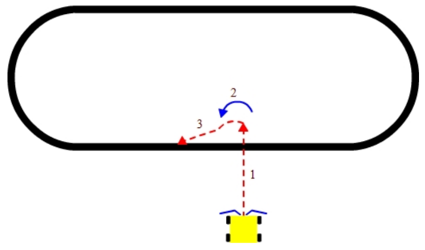


Fig. 7. Aligned with the track for the first time

Component	Mode mapping				
CPU	Init		Follow	Find	
LightS	Black	White	Black	White	
Motors	F ‖ L		F	L	R

TABLE III
THE MODE MAPPING OF THE PATH FINDING CAR

Both global and local mode switch activities can be found in this path finding car. A global mode switch is triggered by the CPU when the light sensor detects a sudden color change and some functional requirements are satisfied. A typical local mode switch is the mode switch between L and R of the Motors component when the system is in $Find$. The MMA of the CPU, the light sensor and the motors can be found in Figure 8 and 9. Regarding this set of MMAs, some points deserve additional explanation:

- The supported modes of the light sensor is based on its sampling value, thus its mode switch is always triggered by itself.
- Sometimes a MMA may do nothing upon receiving an *Input*. This is marked as "- -" for the *Output*.
- MMA$_{CPU}$ may generate multiple *Output* which are connected by "+".
- Some *Input* of MMA$_{CPU}$ contains *Conditionx*(x∈[1,4]). This comes from the system functionality, e.g. a counter recording how many times the car detects the black color. The CPU requires this information in order to derive the new modes of the motors. Therefore, sometimes it is quite challenging to entirely separate mode mapping from system functionalities.

V. Conclusion and future work

We have proposed a mode mapping mechanism as en extension of our Mode Switch Logic (MSL) for CBMMSs. This mechanism can be easily adopted by each composite

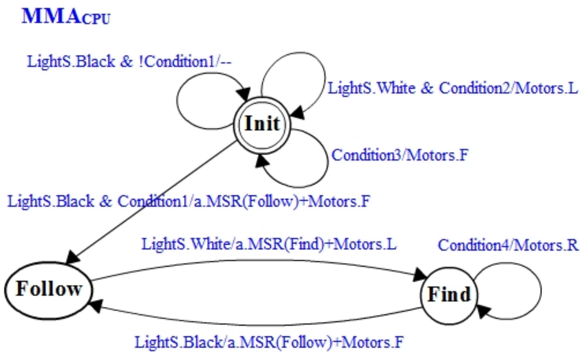

MMACPU
LightS.Black & !Condition1/--
LightS.White & Condition2/Motors.L
Init
Condition3/Motors.F
LightS.Black & Condition1/a.MSR(Follow)+Motors.F
LightS.White/a.MSR(Find)+Motors.L
Condition4/Motors.R
Follow
Find
LightS.Black/a.MSR(Follow)+Motors.F

Fig. 8. The mode mapping automaton of CPU

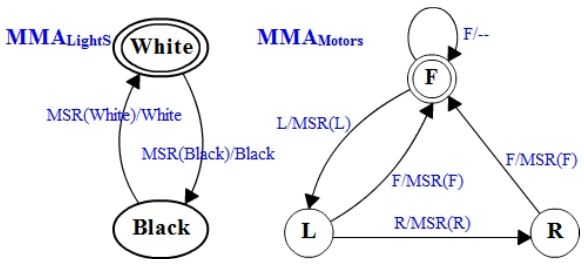


Fig. 9. The mode mapping automata of LightS and Motors

component during MSR propagation to handle different types of mode incompatibility, including local mode switch as a special case. The mode mapping is realized by a set of rules that can be formally expressed. We also extend our previous MSR propagation mechanism by considering mode mapping and provide an algorithm that takes the mode mapping rules as inputs. Our mode mapping mechanism is first demonstrated by a small example and finally it is implemented in a simple case study.

In our future work, we intend to further extend and generalize our mode switch mechanism, including considering *Independent mode switch*, i.e., mode switch of some components that are independent of the mode switch of other components, and *adaptive mode switch* in which different mode mappings can be selected based on on predefined conditions. In addition, the correctness of our mode mapping mechanism should be formally verified. A further issue is *atomic component executions*, i.e., lifting the current assumption that the execution of any component can be immediately interrupted by a *MSR* during the mode switch. Furthermore, when our MSL is mature enough, it is our ambition to implement it in the ProCom framework [24] that embodies the feature of component reuse very well.

ACKNOWLEDGMENT

This work is supported by the Swedish Research Council.

REFERENCES

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

[2] Y. Hang, E. Borde, and H. Hansson, “Composable mode switch for component-based systems,” in *APRES ’11: Third International Workshop on Adaptive and Reconfigurable Embedded Systems*, 2011, pp. 19–22.

[3] Y. Hang and H. Hansson, “Timing analysis for a composable mode switch,” in *The Work-in-Progress session of the 23rd Euromicro Conference on Real-Time Systems*, 2011, pp. 15–18.

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

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

[6] J. Real and A. Crespo, “Mode change protocols for real-time systems: A survey and a new proposal,” *Real-Time Systems*, vol. 26, no. 2, pp. 161–197, 2004.

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

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

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

[10] P. Pedro and A. Burns, “Schedulability analysis for mode changes in flexible real-time systems,” in *10th Euromicro Conference on Real-Time Systems*, 1998, pp. 172–179.

[11] B. Andersson, “Uniprocessor edf scheduling with mode change,” in *12th International Conference on Principles of Distributed Systems*, 2008, pp. 572–577.

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

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

[14] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan, “A multi-mode real-time calculus,” in *Real-Time Systems Symposium*, 2008, pp. 59–69.

[15] L. T. X. Phan, I. Lee, and O. Sokolsky, “Compositional analysis of multi-mode systems,” in *22nd Euromicro Conference on Real-Time Systems*, 2010, pp. 197–206.

[16] ——, “A semantic framework for mode change protocols,” in *17th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2011, pp. 91–100.

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

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

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

[20] T. A. Henzinger, B. Horowitz, and C. M. Kirsch, “Giotto: A time-triggered language for embedded programming,” in *PROCEEDINGS OF THE IEEE*, 2001, pp. 166–184.

[21] J. Templ, “TDL specification and report,” Department of Computer Science, University of Salzburg, Tech. Rep., Nov. 2003.

[22] P. D. S. Resmerita and W. Pree, “Timing definition language (TDL) modeling in ptolemy II,” Department of Computer Science, University of Salzburg, Tech. Rep., Jun. 2008.

[23] P. Pedro, “Schedulability of mode changes in flexible real-time distributed systems,” Ph.D. dissertation, University of York, Sep. 1999.

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