Mode switch handling for the ProCom component model

Yin Hang*[1], Hongwan Qin[2], Jan Carlson[1], and Hans Hansson[1]

[1]Mälardalen Real-Time Research Centre, Mälardalen University, Västerås, Sweden
[2]Lund University, Lund, Sweden

February 26, 2013

Abstract

Component-Based Software Engineering has been deemed a suitable technique for the development of complex embedded systems, as component reuse makes it easier to manage software complexity. Another way of reducing software complexity is by partitioning system behavior into different operational modes. Such a multi-mode system can change its behavior by switching between modes. For a multi-mode system built by components, a challenge is its mode switch handling.

In this report, a novel approach is presented to integrate our mechanism for handling mode switch (the Mode Switch Logic), in ProCom, which is a component model designed for the development of real-time embedded systems. The outcome is a slightly extended version of ProCom which not only supports the development of multi-mode applications, but also is able to handle mode switch.

1 Introduction

The growing complexity of the software of embedded systems entails new techniques for the development of complex embedded systems, as traditional techniques are becoming less suitable. Component-Based Software Engineering (CBSE) [4] is a promising paradigm for developing complex systems by virtue of its benefits such as the management of software complexity, reduced time to market and improved software quality. CBSE allows a system to be built by reusable components which are independently developed so that the system does not have to be developed from scratch. The success of CBSE has been evidenced by a variety of component models proposed both in industry and academia [5] [12]. Among these component models, and in the focus of this report, ProCom [3] is a component model for real-time and embedded systems, particularly targeting vehicular, automation and telecommunication applications.

*Contact: young.hang.yin@mdh.se

In contrast to CBSE, another common approach to reducing software complexity of embedded systems is to partition system behavior into different operational modes. A multi-mode system can start running in a default mode and 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 *Top*, consists of three components: *a*, *b* and *c*. Component *b* is composed by *d* and *e*. Components *a*, *c*, *d* and *e* are primitive components because they cannot be further decomposed. Components *Top* and *b* are composite components because they are both compositions of other components. Since the component hierarchy has a tree structure, a composite component and its subcomponents have a parent-and-children relationship. For instance, *b* is the parent of *d* and *e*, which in turn are the children of *b*. Moreover, the system can run in two modes: m^1_{Top} and m^2_{Top}. When the system is in m^1_{Top}, Component *c* is deactivated (i.e. not running), shown in the component hierarchy in Figure 1 by not displaying *c* in mode m^1_{Top}. In contrast, when the system is in m^2_{Top}, *c* is activated whilst *e* is deactivated. Besides, Component *a* has different mode-specific behaviors represented by black and grey colors in Figure 1.

A key issue of a CBMMS is its mode switch handling. A mode switch may amount to the joint mode switches of many different components. For instance, a system mode switch from m^1_{Top} to m^2_{Top} in Figure 1 requires the activation of *c*, the deactivation of *e* and the behavior change of *a*. The mode switches of different components must be well synchronized and coordinated to guarantee a correct system mode switch. For that reason, we have developed the Mode Switch Logic (MSL) [7] [6], a mechanism for handling the mode switch of CBMMSs.

With the ProCom component model and MSL as two background techniques, this report provides a theoretical guidance for implementing MSL in ProCom. Currently, ProCom does not support multi-mode systems. However, the approach presented in this report realizes the development of CBMMSs together with their mode switch handling in ProCom. The remainder of the report is organized as follows: Section 2 introduces the ProCom component model. Section 3 gives a brief introduction of MSL. As the main contribution of the report, Section 4 describes how MSL is implemented in ProCom. In Section 5, an example is used to illustrate the major elements in Section 4. Related work is reviewed in Section 6. Finally, Section 7 concludes the report and discusses some future work.

2 The ProCom component model

ProCom [3] is a component model for the development of distributed real-time and embedded systems software. Compared with other existing component

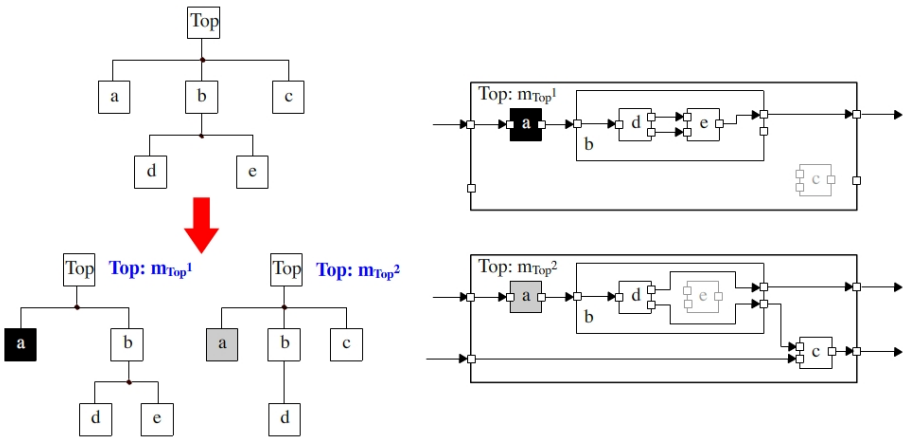


Figure 1: A conceptual component-based multi-mode system

models, the most distinctive feature of ProCom is its two layers: ProSave—the lower layer, and ProSys—the higher layer. With different concerns, these two layers allow a system to be modeled at different levels of granularity. Next we shall give a brief introduction of each layer.

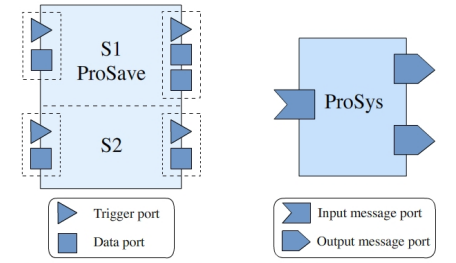
2.1 The ProSave layer

The ProSave layer is used to design subsystems allocated to a single physical node. It is based on a pipe-and-filter architectural style and has clear separation between control flow and data flow. A component belonging to this layer is called a ProSave component. A ProSave component can provide one or more services, each of which realizes a particular functionality. Each service has a single input port group and one or more output port groups. A port group consists of a trigger port and one or more data ports, with the trigger port dedicated to control flow and the data ports dedicated to data flow.

A ProSave component is passive in the sense that the execution of each of its services requires external activation. For a service S of a ProSave component, when the input trigger port is activated, S becomes active and performs computation based on its input data ports. After completing the computation, S writes the result to its output data ports, activates its output trigger port(s) and then becomes passive.

Figure 2(a) depicts a ProSave component with two services S_1 and S_2. Service S_1 has an input port group (consisting of an input trigger port and an input data port) and an output port group (consisting of an output trigger port and two output data ports). The ports of S_2 can be explained in the same way.

The communication between ProSave components is based on a single directional one-to-one connection between ports of the same types. An output trigger port of a ProSave component is directly connected to an input trigger port of another ProSave component. Similarly, an output data port of a ProSave component is directly connected to an input data port of another ProSave component. In addition, ProCom defines a couple of connectors for more advanced communication in ProSave. Figure 3 lists the most commonly used connectors:



(a) A ProSave component (b) A ProSys component

Figure 2: ProSave and ProSys components

- Control Or: It has at least two input trigger ports and one output trigger port. Its output trigger port is activated when any one of its input trigger ports is activated.
- Control Join: It has at least two input trigger ports and one output trigger port. Its output trigger port is activated only when all its input trigger ports are activated. It can also be presented by a small circle graphically.
- Control Fork: It has one input trigger port and at least two output trigger ports. When its input trigger port is activated, all its output trigger ports will be activated. It can also be presented by a thick dot graphically.
- Data Or: It has at least two input data ports and one output data port. The data arriving at any one of its input data ports is forwarded to its output data port.
- Data Fork: It has one input data port and at least two output data ports. The data arriving at its input data port will be duplicated and produced at all its output data ports. Just like Control Fork, it can also be presented by a thick dot graphically.
- Selection: It has an input trigger port, at least one input data port and at least two output trigger ports. When its input trigger port is activated, it will activate exactly one of its output trigger ports according to the data written to its input data port(s).

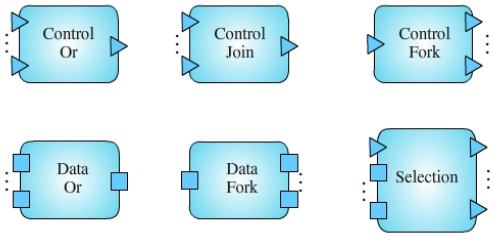


Figure 3: Typical connectors in ProSave

The ProSave layer is hierarchical, i.e. a composite ProSave component can be composed by other ProSave components.

2.2 The ProSys layer

The ProSys layer is used to construct distributed subsystems. A component belonging to this layer is called a ProSys component. A ProSys component has a number of input message ports and output message ports. Figure 2(b) depicts a ProSys component with one input message port and two output message ports. The communication between ProSys components is realized by asynchronous message passing. A message is sent from an output message port and received from an input message port via message channels. A message channel can be associated with multiple input and output message ports, enabling many-to-many communication.

A ProSys component is active, as it has its own threads. Therefore, concurrent execution is allowed in ProSys. Just like ProSave, ProSys is also hierarchical in the sense that a composite ProSys component can be composed by other ProSys components.

The integration of ProSys and ProSave is realized by building a ProSys component with ProSave components, illustrated in Figure 4. In order to map the pipe-and-filter architecture to message passing, a message port is internally treated as a pair of a trigger port and a data port. In addition, a special connector *Clock* can be used for the periodic activation of ProSave components composing the ProSys component.

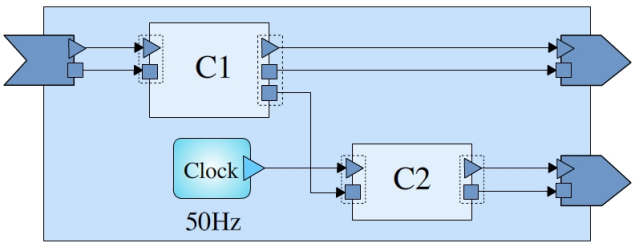


Figure 4: A ProSys component composed by ProSave components

3 The Mode Switch Logic

The Mode Switch Logic (MSL) [7] [6] is a systematic approach to the mode switch handling of CBMMSs. The major elements of MSL include a mode-aware component model, a mode mapping mechanism and a mode switch runtime mechanism. In the following, we shall briefly introduce these elements, which establish the foundation of our MSL implementation in ProCom.

3.1 The mode-aware component model

The mode-aware component model defines essential features that a component should possess in order to support both individual mode switch and cooperative mode switch with other components. As is illustrated in Figure 5, a component can support multiple modes and has a unique configuration defined for each mode. The mode switch of a component is realized by its reconfiguration, i.e. changing its configuration in the current mode to a new configuration in the

target mode. The mode switch runtime mechanism of MSL controls the mode switch behavior of a multi-mode component. 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_{in}^{MSX}, used to exchange mode related information with its subcomponents during a mode switch.

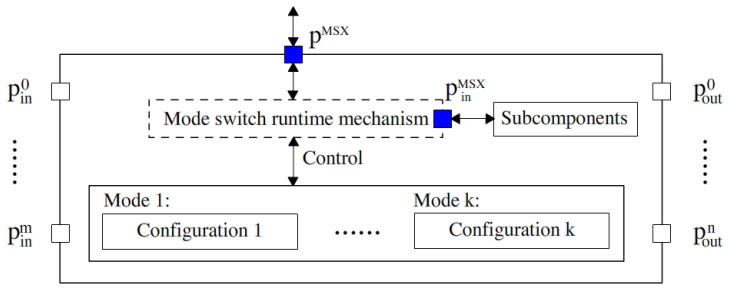


Figure 5: The mode-aware component model

3.2 Mode mapping

Usually a multi-mode component is independently developed without assuming the context where it will be used. For a multi-mode composite component c_i, composed by $c_j^1, c_j^2, \cdots, c_j^n$ $(n \in \mathbb{N})$, the mapping between the modes of c_i and its subcomponents must be properly specified. Among these components, when a component is running in one of its supported modes, c_i must be able to derive the current modes of itself and its subcomponents. Similarly, when a mode switch takes place, c_i should also be able to derive the new modes of itself and its subcomponents. This is called mode mapping, which can be handled by the mode mapping mechanism provided by MSL. The central idea of this mode mapping mechanism is to express the mode mapping of a composite component by a group of Mode Mapping Automata (MMAs). The relation between MMAs and mode mapping can be represented by Figure 6, where the mode mapping of c_i is expressed by a set of MMAs including MMA_{c_i} and $\text{MMA}_{c_j^k}$ $(k = [1, n])$. Each Mode Mapping Automaton (MMA) is associated with a specific component among c_i and c_j^k. All MMAs reside in c_i and are internally synchronized. An initial version of the mode mapping mechanism of MSL is introduced in [8] and refined in [6].

3.3 The mode switch runtime mechanism

The mode switch runtime mechanism handles the mode switch of a CBMMS and the mode switches of its components at runtime. In this report, we shall introduce its two most fundamental elements: the Mode Switch Propagation (MSP) protocol and the mode switch dependency rule. The MSP protocol specifies how a mode switch event is detected by an individual component and efficiently

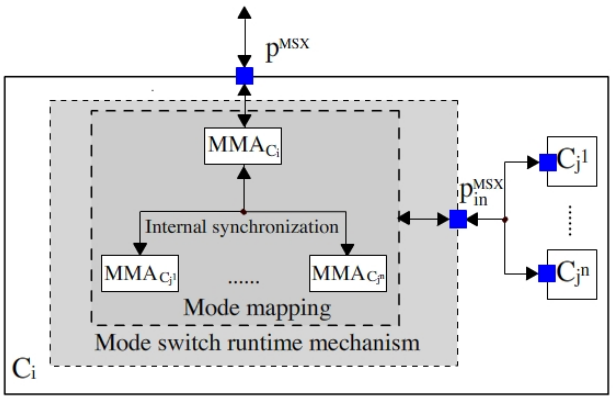


Figure 6: Mode mapping and Mode Mapping Automata (MMAs)

propagated to other related components. The mode switch dependency rule guarantees the mode consistency between a system and its components after each mode switch. Both elements of the mode switch runtime mechanism are based on the transmission of downstream and upstream primitives throughout the component hierarchy. A downstream primitive is sent from a composite component to its subcomponents via its dedicated mode switch port p_{in}^{MSX}. An upstream primitive is sent from a component to its parent via its dedicated mode switch port p^{MSX}.

3.3.1 The Mode Switch Propagation (MSP) protocol

MSL distinguishes different roles for the components of a CBMMS:

- Mode Switch Source (MSS): a component which can detect a mode switch event (e.g. the value of a sensor exceeds a threshold) and actively request to switch mode. When an MSS c_k requests to switch mode, a mode switch scenario is triggered, uniquely specified by c_k, its current mode $m_{c_k}^i$ and target mode $m_{c_k}^j$.
- Mode Switch Decision Maker (MSDM): a component which triggers a mode switch based on the mode switch request from an MSS. Since an MSDM has a higher authority than the corresponding MSS, the MSDM must be at a higher level in the component hierarchy.
- Type A component: a component which needs to switch mode as a consequence of the mode switch of an MSS.
- Type B component: a component not affected by the mode switch of an MSS.

The main purpose of the Mode Switch Propagation (MSP) protocol is to propagate the mode switch request of an MSS to all Type A components without disturbing Type B components. Let *Top* be the top component of a CBMMS. For a component c_i, let P_{c_i} be the parent of c_i. Let $T_{c_i} = A$ or $T_{c_i} = B$ denote c_i is a Type A or Type B component. A general description of the MSP protocol

is as follows:

The Mode Switch Propagation (MSP) protocol: *When an MSS c_i detects a mode switch event, it will request to switch mode by triggering a mode switch scenario. If $c_i \neq Top$, c_i will issue a primitive* `MSR` *(Mode Switch Request) which is propagated upstream and stepwise until it reaches the MSDM c_j. Let C_M be the set of vertically intermediate components between c_i and c_j in the component hierarchy. When $c_k \in C_M$ receives the* `MSR`*, c_k forwards it to P_{c_k} because $T_{c_k} = A$ and the current state of c_k allows a mode switch. Contrastly, c_j is the MSDM under three conditions; (1) $T_{c_j} = B$; (2) $T_{c_j} = A$ and the current state of c_j does not allow a mode switch; (3) $T_{c_j} = A$ and the current state of c_j allows a mode switch whereas $c_j = Top$. Then,*

- *In Condition (2), c_j will reject the* `MSR` *by doing nothing. Mode switch propagation is terminated and no component will switch mode.*
- *In conditions (1) and (3), c_j will approve the* `MSR` *by issuing a primitive* `MSQ` *(Mode Switch Query) that is propagated downstream and stepwise to all Type A components. After receiving an* `MSQ`*, a component c_k will check if its current state allows a mode switch and is required to reply to P_{c_k} with either a primitive* `MSOK` *or* `MSNOK`*. Component c_k only replies with an* `MSOK` *when its current state allows a mode switch (and all its Type A subcomponents reply with an* `MSOK` *if c_k is composite). Otherwise, if the current state of c_k does not allow a mode switch, c_k will reply with an* `MSNOK` *without propagating the* `MSQ` *downstream further. 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 has replied with an* `MSOK`*, c_j will trigger a mode switch by issuing a primitive* `MSI` *(Mode Switch Instruction) 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 a primitive* `MSD` *(Mode Switch Denial) 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 and the rest will be the same as the case when $c_i \neq Top$.*

The identification of the MSDM based on a specific `MSR`, and the approval or rejection of an `MSR` by the MSDM are determined by the mode mapping of the MSDM and its current state. Besides, mode mapping also identifies Type A and Type B components, and derives the new mode of each Type A component. This reveals the close cooperation between the MSP protocol and mode mapping.

3.3.2 The mode switch dependency rule

For a CBMMS, a system mode switch corresponds to the mode switches of all Type A components. The correctness of a system mode switch relies on not only the correct mode switch of each Type A component, but also on the mode consistency between the system and all Type A components. When a system

is in the process of switching mode, some components may complete mode switch earlier than some other components. This temporary mode inconsistency is tolerable, however, when a system completes a mode switch, all Type A components must be running in their new modes. In order to guarantee such mode consistency, a mode switch dependency rule is introduced in MSL. In general, the mode switch dependency rule is described as follows:

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. A primitive `MSC` (Mode Switch Completion) 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 MSP protocol and the mode switch dependency rule can be demonstrated in Figure 7, which shows a mode switch process based on the system introduced in Figure 1 under the guidance of the mode switch runtime mechanism of MSL. Component *b* is an MSS and *Top* is the corresponding MSDM. For this specific mode switch scenario, *Top*, *a*, *b*, *c* and *d* are all Type A components while *e* is a Type B component. Component *Top* approves the `MSR` by issuing an `MSQ` to its Type A subcomponents *b* and *c*. Then *b* further propagates the `MSQ` to its Type A subcomponent *d*. All Type A components reply with an `MSOK` after checking their current states upon receiving an `MSQ`. When *Top* receives an `MSOK` from *a*, *b* and *c*, it triggers a mode switch by issuing an `MSI` that follows the propagation trace of the `MSQ`. After the `MSI` propagation, a component will start reconfiguration that is represented by black bars in Figure 7. Finally, `MSC` is propagated bottom-up in the same fashion as `MSOK`. White bars mean that the mode switch of a composite component is blocked when it has completed reconfiguration but is still waiting for an `MSC` from one or more of its Type A subcomponents.

4 Implementing MSL in ProCom

In previous sections, the ProCom component model and MSL have been introduced separately. In this section, we describe our contribution in this report—implementing MSL in the ProCom component model. The basic idea of our approach is to integrate the key elements of MSL in ProCom with minimum modification to ProCom. First, a ProCom component must be made mode-aware to become consistent with the mode-aware component model and the

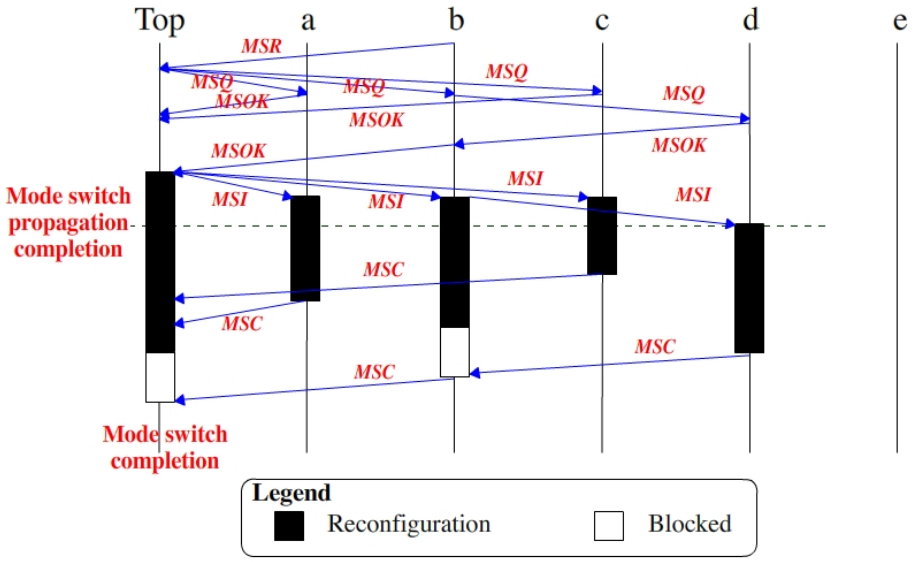


Figure 7: A complete mode switch process

mode mapping mechanism. Second, the mode switch runtime mechanism must be included in each ProCom component for its mode switch handling. Furthermore, since component connections may change during a mode switch, ProCom must be able to provide multiple versions of component connections and switch between them when necessary. Next we shall present our approach in terms of the definition of multi-mode ProCom components, the mode switch handling in ProCom, and the support of varied component connections in different modes. To simplify the presentation, two assumptions are made: (1) the execution of a component in ProCom can be immediately interrupted by a mode switch; (2) no new mode switch event is detected when a system is switching mode. The handling of atomic component execution which cannot be interrupted is presented in Chapter 5 of [6]. Without the second assumption, a conflict may occur due to multiple mode switch triggering. It is our ongoing work to provide handling of such conflicts.

4.1 Multi-mode ProCom components

Multi-mode components have not been considered by the current ProCom component model. However, we are able to define multi-mode ProCom components without extending ProCom. Since ProCom distinguishes ProSave and ProSys, multi-mode ProSave and ProSys components will be introduced separately in the following.

In ProSave, in order to separate the mode switch handling from the functional behavior of each component, a dedicated service S_{mode} is used for the mode switch handling of a multi-mode ProSave component. This service includes the definition of multiple modes, the configuration for each mode, mode mapping and the mode switch runtime mechanism. Furthermore, S_{mode} also has dedicated mode switch ports that correspond to p^{MSX} and p^{MSX}_{in} in the

mode-aware component model. The service S_{mode} consists of an input port group and an output port group. The input port group comprises an input trigger port p_i^{mst} and an input data port p_i^{ms}, while the output port group comprises an output trigger port p_o^{mst} and an output data port p_o^{ms}. Figure 8(a) shows a typical multi-mode ProSave component c_i with two services, the lower service being S_{mode}. The dedicated mode switch ports of S_{mode} are highlighted in purple.

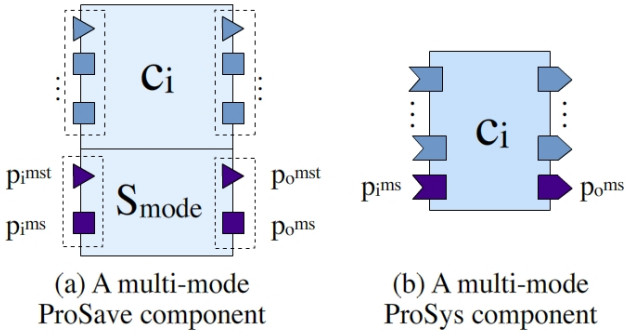


Figure 8: Multi-mode ProSave and ProSys components

In ProSys, no concept of service exists and concurrent execution is allowed in a ProSys component, hence a dedicated internal thread can be used for the mode switch handling of a multi-mode ProSys component. Similar to ProSave, a multi-mode ProSys component should also have dedicated mode switch ports. Since a ProSys component is equipped with message ports that integrate both control flow and data flow, the dedicated mode switch ports of a multi-mode ProSys component can be assigned to an input message port p_i^{ms} and an output message port p_o^{ms}. Figure 8(b) shows a typical multi-mode ProSys component c_i whose dedicated mode switch ports are highlighted in purple.

4.2 The mode switch handling in ProCom

Section 3 has stated that the mode switch of a CBMMS is handled by the mode switch runtime mechanism of MSL. In this subsection, we integrate this mode switch runtime mechanism in ProCom. For primitive multi-mode ProCom components, the mode switch runtime mechanism can be simply implemented in the code (see algorithms 1 and 2 in the appendix). In this report, our focus is on the mode switch handling of composite multi-mode ProCom components which requires a more elaborate approach in both ProSave and ProSys. Hereafter we by default imply multi-mode ProCom components while mentioning ProSave or ProSys components. The mode switch of a composite ProCom component is handled by dedicated subcomponents via its dedicated mode switch ports defined in Section 4.1.

4.2.1 The mode switch handling in ProSave

Since a composite ProSave component has no behavior and is just a composition of a set of enclosed ProSave components, a reasonable strategy is to introduce

additional subcomponents that are dedicated to its mode switch handling. The same strategy can be applied to a ProSys component composed by ProSave components.

For a composite component c_i, which is either a composite ProSave component or a composite ProSys component composed by ProSave components, we introduce two dedicated subcomponents of c_i for its mode switch handling: $MSL_{c_i}^A$ and $MSL_{c_i}^B$, both of which are primitive ProSave components. Components $MSL_{c_i}^A$ and $MSL_{c_i}^B$ interact with the S_{mode} service of each subcomponent of c_i and are synchronized with each other.

Let $c_i.p$ denote the port p of component c_i. Also, let $SC_{c_i} = \{c_j^1, c_j^2, \cdots, c_j^n\}$ $(n \in \mathbb{N})$ denote the set of subcomponents of c_i, excluding $MSL_{c_i}^A$ and $MSL_{c_i}^B$. Figure 9 illustrates the ports of $MSL_{c_i}^A$ and $MSL_{c_i}^B$, both of which are synchronized with each other via their synchronization ports p_i^{sync} and p_o^{sync}. Component $MSL_{c_i}^A$ has a single service with an input port group and an output port group. Apart from the synchronization ports, these port groups consist of the following ports:

- p_i^t: an input trigger port whose activation makes $MSL_{c_i}^A$ active.
- p_i^{msx}: an input data port for receiving a downstream primitive (e.g. `MSQ`, `MSI` or `MSD`) from the parent of c_i.
- p_o^t: an output trigger port activated after $MSL_{c_i}^A$ completes its current instance of execution.
- $P_o^{msx} = \{p_o^1, p_o^2, \cdots, p_o^n\}$ $(n = |SC_{c_i}|)$: a set of output data ports for sending a downstream primitive to SC_{c_i}.
- p_o^s: an output data port indicating the current mode of c_i.

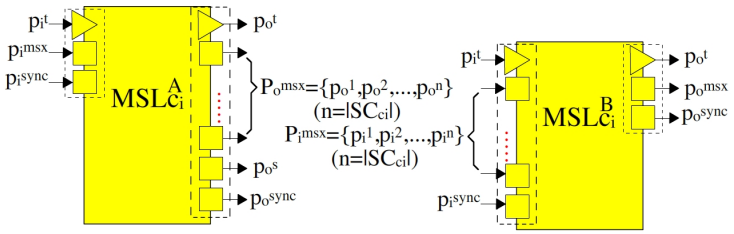


Figure 9: The pair of ProSave subcomponents of c_i for handling its mode switch

The ports of $MSL_{c_i}^B$ are quite symmetrical to $MSL_{c_i}^A$. Apart from the synchronization ports, $MSL_{c_i}^B$ also has the following ports:

- p_i^t: an input trigger port whose activation makes $MSL_{c_i}^B$ active.
- $P_i^{msx} = \{p_i^1, p_i^2, \cdots, p_i^n\}$ $(n = |SC_{c_i}|)$: a set of input data ports for receiving an upstream primitive (e.g. `MSR`, `MSOK`, `MSNOK`, or `MSC`) from SC_{c_i}.
- p_o^t: an output trigger port activated after $MSL_{c_i}^B$ completes its current instance of execution.

- p_o^{msx}: an output data port for sending an upstream primitive to the parent of c_i.

The connections around $MSL_{c_i}^A$ and $MSL_{c_i}^B$ are illustrated in Figure 10. The ports associated with services rather than S_{mode} of both c_i and SC_{c_i} have been omitted for simplicity. Components $MSL_{c_i}^A$ and $MSL_{c_i}^B$ are connected to both c_i and SC_{c_i}. Their connection with c_i is represented by the connection between $c_i.p_i^{ms}$ and $MSL_{c_i}^A.p_i^{msx}$ and the connection between $MSL_{c_i}^B.p_o^{msx}$ and $c_i.p_o^{ms}$. Their connection with SC_{c_i} is represented by the connection between $MSL_{c_i}^A.p_o^k$ $(k = [1, n])$ and $c_j^k.p_i^{ms}$ and the connection between $c_j^k.p_o^{ms}$ and $MSL_{c_i}^B.p_i^k$. A mode related control flow is established within c_i from $MSL_{c_i}^A$ to SC_{c_i} and then to $MSL_{c_i}^B$. A *Control Or* connector is used so that $MSL_{c_i}^B$ can be triggered by any subcomponent of c_i. This connection pattern is repeated within all composite ProSave components. For instance, $\forall c_j^k \in SC_{c_i}$ $(k = [1, n])$ which is composite, the internal connections of c_j^k will exhibit the same connection pattern as c_i. Such a connection pattern enables the transmission of both downstream and upstream primitives. For instance, a downstream primitive from c_i to c_j^k can be transmitted from $MSL_{c_i}^A.p_o^k$ to $c_j^k.p_i^{ms}$ and c_j^k can propagate the primitive further to lower levels if it is composite and wants to. Conversely, an upstream primitive from c_j^k to c_i can be transmitted from $c_j^k.p_o^{ms}$ to $MSL_{c_i}^B.p_i^k$ and then $MSL_{c_i}^B$ will forward this primitive to $MSL_{c_i}^A$ via their synchronization ports. Let c_l be the parent of c_i, if c_i wants to propagate this primitive further to c_l, $MSL_{c_i}^B$ can send the primitive to $c_i.p_o^{ms}$ which must be externally connected to $MSL_{c_l}^B$, the component dedicated to the mode switch handling of c_l.

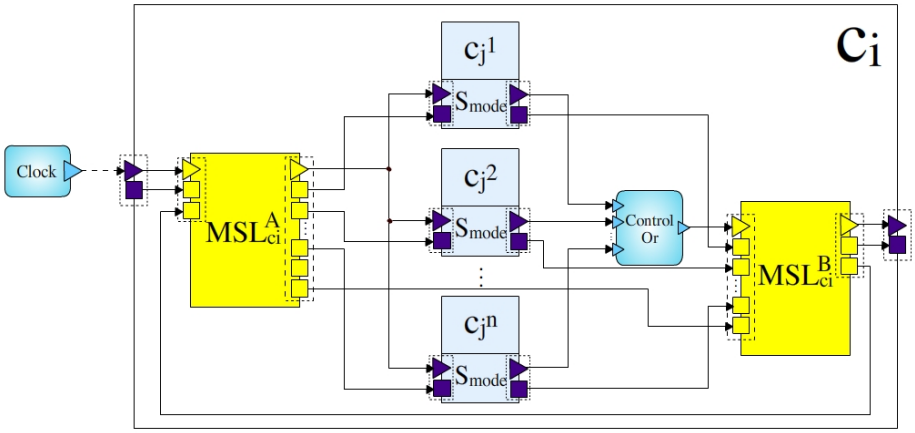


Figure 10: The connections around $MSL_{c_i}^A$ and $MSL_{c_i}^B$

Attention must be paid to the connector *Clock* in Figure 10. Since ProSave components are passive and require external activation, a common *Clock* must be placed at the top ProSave level, periodically triggering the mode related control flow in ProSave.

Moreover, our initial intention was to use a single dedicated component to handle the mode switch of a composite ProSave component. The reason why two such components are used is attributed to the rigorous execution semantics in ProSave, which prohibits mutual triggering between two neighboring ProSave

components. If a single component, say MSL_{c_i}, is used instead of $MSL^A_{c_i}$ and $MSL^B_{c_i}$, there must exist mutual triggering between MSL_{c_i} and SC_{c_i}. Consequently, the execution semantics of ProCom will be violated.

Since both $MSL^A_{c_i}$ and $MSL^B_{c_i}$ are primitive ProSave components, they can be easily implemented by following the mode mapping mechanism and mode switch runtime mechanism of MSL. Their detail mode switch behaviors can be found in the algorithms in the appendix of this report.

4.2.2 The mode switch handling in ProSys

The mode switch handling in ProSys is similar to that in ProSave.

For a composite ProSys component c_i, we introduce a dedicated subcomponent of c_i for its mode switching handling: MSL_{c_i} which plays an equal role as the pair of $MSL^A_{c_i}$ and $MSL^B_{c_i}$. However, message passing between ProSys components is more flexible than the pipe-and-filter communication style in ProSave. Two ProSys components can send messages to each other, therefore, a single subcomponent MSL_{c_i} is sufficient for the mode switch handling of c_i.

Still, let $SC_{c_i} = \{c^1_j, c^2_j, \cdots, c^n_j\}$ $(n \in \mathbb{N})$ denote the set of subcomponents of c_i, excluding MSL_{c_i}. Figure 11 illustrates the ports of MSL_{c_i}:

- p^{msx}_i: an input message port for receiving a downstream primitive from the parent of c_i.
- $P_i = \{p^1_i, p^2_i, \cdots, p^n_i\}$ $(n = |SC_{c_i}|)$: a set of input message ports for receiving an upstream primitive from SC_{c_i}.
- p^s_o: an output message port indicating the current mode of c_i.
- $P_o = \{p^1_o, p^2_o, \cdots, p^n_o\}$ $(n = |SC_{c_i}|)$: a set of output message ports for sending a downstream primitive to SC_{c_i}.
- p^{msx}_o: an output message port for sending an upstream primitive to the parent of c_i.

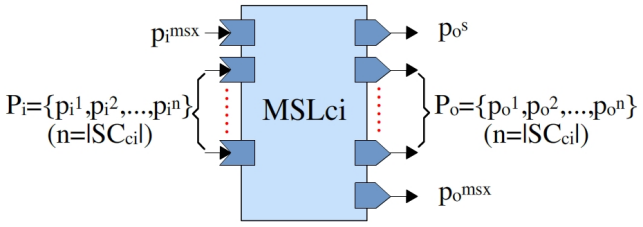


Figure 11: The ProSys subcomponent of c_i for handling its mode switch

The connections around MSL_{c_i} are illustrated in Figure 12, where the ports not related to the mode switches of both c_i and SC_{c_i} have been omitted for simplicity. Dark red shapes are message channels. Component MSL_{c_i} has direct communication with both c_i and SC_{c_i}. On the one hand, $MSL_{c_i}.p^{msx}_i$ is connected to $c_i.p^{ms}_i$ and $MSL_{c_i}.p^{msx}_o$ is connected to $c_i.p^{ms}_o$. On the other hand, $MSL_{c_i}.p^k_o$ $(k = [1, n])$ is connected to $c^k_j.p^{ms}_i$ and $c^k_j.p^{ms}_o$ is connected to $MSL_{c_i}.p^k_i$. No *Clock* is needed in ProSys, because ProSys components are

active and can execute without external activation. Additionally, since a message channel allows many-to-many communication, the *Control Or* connector in ProSave is removed. This connection pattern is repeated for all composite ProSys components while enabling the transmission of both downstream and upstream primitives.

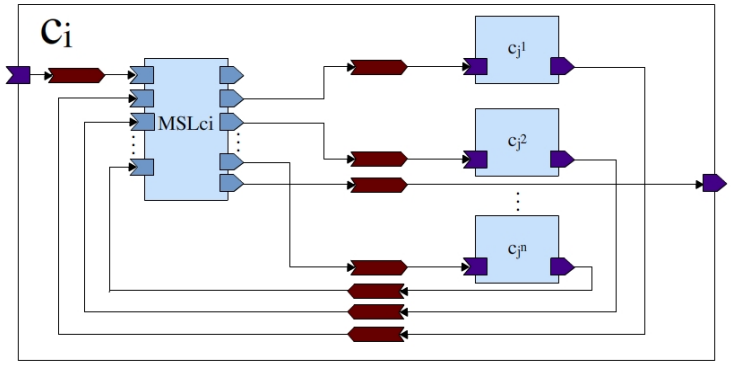


Figure 12: The connections around MSL_{c_i}

MSL_{c_i} is a primitive ProSys component where the mode switch runtime mechanism of c_i is implemented. Its detail mode switch behavior is described as algorithms in the appendix.

4.3 Managing the variability of ProCom component connections in multiple modes

Section 4.2 explains the mode switch handling of a ProCom component, yet without addressing how component reconfiguration is achieved during a mode switch in ProCom. Many properties of a component can be changed by reconfiguration, e.g. functional behavior and running status (activated or deactivated). Among these properties, our focus in this report is on the inner component connections of a composite ProCom component. As is indicated in Figure 1 at the very beginning of the report, the inner component connections of a composite component c_i can be different while c_i is in different modes. The inner component connections of c_i for each mode can be separately defined at design time and changed to each other when a mode switch occurs at runtime. In order to manage the variability of component connections in different modes in ProCom, we provide a solution which can automatically generate a complete view of inner component connections of each composite ProCom component based on its inner component connections separately defined for each mode. Depending on the current mode of a composite component, the activated subcomponents and corresponding inner component connections are selected.

4.3.1 Managing the variability of component connections in ProSave

Consider a composite ProSave component c_i, whose inner component connections are mode-dependent. The basic idea of managing the variability of inner component connections of c_i is to package each $c_j^k \in SC_{c_i}$ $(k = [1, n], n = |SC_{c_i}|)$

with additional connectors. Each connector integrates all the possible incoming or outgoing connections of a specific port for all modes and can select the correct connection based on the current mode of c_i.

Let $M_{c_i} = \{m^1_{c_i}, m^2_{c_i}, \cdots, m^q_{c_i}\}(q > 1)$ be the set of supported modes of c_i. Suppose the inner component connections of c_i for each mode $m^l_{c_i}$ $(l = [1, q])$ have been well-defined. Besides, for a ProSave component c, the following sets of ports are defined:

- P^t_i: the set of input trigger ports excluding $c.p^{mst}_i$.
- P^d_i: the set of input data ports excluding $c.p^{ms}_i$.
- P^t_o: the set of output trigger ports excluding $c.p^{mst}_o$.
- P^d_o: the set of output data ports excluding $c.p^{ms}_o$.

In order to merge the inner component connections of c_i into a complete view, connectors are automatically generated within c_i based on the following rules:

- For each p where $p \in c^k_j.P^t_i$ or $p \in c_i.P^t_o$, a *Control Or* connector A is generated, with a set of input trigger ports $P_i = \{p^{t1}_i, p^{t2}_i, \cdots, p^{tq}_i\}(q = |M_{c_i}|)$ and an output trigger port p^t_o. The incoming connection to $A.p^{tl}_i$ $(l = [1, q])$ follows the pre-defined connection while c_i is in mode $m^l_{c_i}$. The output trigger port $A.p^t_o$ is directly connected to p.
- For each p where $p \in c^k_j.P^d_i$ or $p \in c_i.P^d_o$, a *Data Or* connector B is generated, with a set of input data ports $P_i = \{p^{d1}_i, p^{d2}_i, \cdots, p^{dq}_i\}(q = |M_{c_i}|)$ and an output data port p^d_o. The incoming connection to $B.p^{dl}_i$ $(l = [1, q])$ follows the pre-defined connection while c_i is in mode $m^l_{c_i}$. The output data port $B.p^d_o$ is directly connected to p.
- For each p where $p \in c^k_j.P^t_o$ or $p \in c_i.P^t_i$, a *Selection* connector C is generated, with an input trigger port p^t_i, an input data port p^s_i and a set of output trigger ports $P_o = \{p^{t1}_o, p^{t2}_o, \cdots, p^{tq}_o\}(q = |M_{c_i}|)$. The input trigger port $C.p^t_i$ is directly connected to p. The input data port $C.p^s_i$ is connected to $MSL^A_{c_i}.p^s_o$ (see Section 4.2). The outgoing connection from $C.p^{tl}_o$ $(l = [1, q])$ follows the pre-defined connection while c_i is in mode $m^l_{c_i}$ according to the data from $C.p^s_i$: If the data returns $m^l_{c_i}$ $(l = [1, q])$, $C.p^{tl}_o$ will be triggered.
- For each p where $p \in c^k_j.P^d_o$ or $p \in c_i.P^d_i$, a *Data Selection* connector D is generated, with an input data port p^d_i, and another input data port p^s_i and a set of output data ports $P_o = \{p^{d1}_o, p^{d2}_o, \cdots, p^{dq}_o\}(q = |M_{c_i}|)$. The input data port $D.p^d_i$ is directly connected to p. The input data port $D.p^s_i$ is connected to $MSL^A_{c_i}.p^s_o$. The outgoing connection from $D.p^{dl}_o$ $(l = [1, q])$ follows the pre-defined connection while c_i is in mode $m^l_{c_i}$ according to the data from $D.p^s_i$: If the data returns $m^l_{c_i}$ $(l = [1, q])$, the data from $D.p^d_i$ will be forwarded exactly to $D.p^{dl}_o$.

The rules above also apply to a ProSys component composed by ProSave components by considering each message port as a port group consisting of a

trigger port and a data port. Among these four generated connectors, *Data Selection* does not exist in the current ProCom component model. Nonetheless, it can be easily developed as its execution semantics is fairly similar to *Selection*. This is the only extension of ProCom required by our approach. The above presented rules are illustrated in Figure 13.

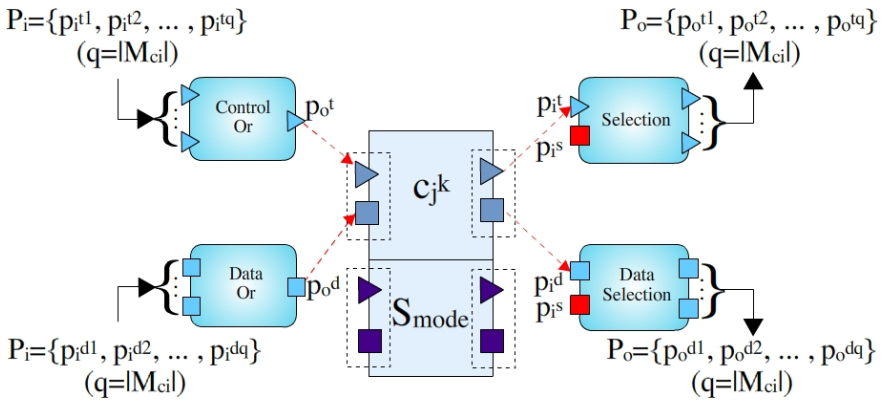


Figure 13: Managing the variability of ProSave component connections

4.3.2 Managing the variability of component connections in ProSys

In comparison with ProSave, the central idea of managing the variability of ProSys component connections is rather similar in that all the generated connectors in ProSave can be replaced with primitive ProSys components. However, since an input message port can receive messages from multiple senders, there is no need to generate ProSys components playing the role of *Control Or* or *Data Or*. Hence, the only ProSys component that needs to be generated is a *Selection* ProSys component which functions as both connectors *Selection* and *Data Selection*.

Let c_i be a composite ProSys component composed by ProSys components, with the set of supported modes $M_{c_i} = \{m^1_{c_i}, m^2_{c_i}, \cdots, m^q_{c_i}\}(q > 1)$ and the set of subcomponents $SC_{c_i} = \{c^1_j, c^2_j, \cdots, c^n_j\}$ $(n = |SC_{c_i}|)$. Suppose the inner component connections of c_i for each mode $m^l_{c_i}$ $(l = [1, q])$ have been well-defined. For a ProSys component c, let P_i be the set of input message ports excluding $c.p^{ms}_i$ and let P_o be the set of output message ports excluding $c.p^{ms}_o$. Then as is illustrated in Figure 14, for each p where $p \in c^k_j.P_o$ $(k = [1, n])$ or $p \in c_i.P_i$, a primitive ProSys component, called *Selection* and denoted as E, is generated, with two input message ports p_i and p_s and a set of output message ports $P = \{p^1_o, p^2_o, \cdots, p^q_o\}$ $(q = |M_{c_i}|)$. The port $E.p_i$ is directly connected to p while $E.p_s$ is connected to $MSL_{c_i}.p^s_o$ (see Section 4.2). The outgoing connection from $E.p^l_o$ $(l = [1, q])$ follows the pre-defined connection while c_i is in $m^l_{c_i}$ according to the data from $E.p_s$: If the data returns $m^l_{c_i}$ $(l = [1, q])$, the message sent to $E.p_i$ will be forwarded to $E.p^l_o$.

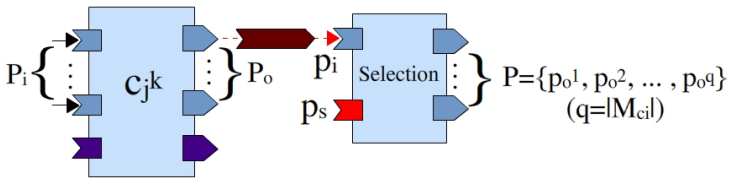


Figure 14: Managing the variability of ProSys component connections

5 An example

Section 4 has presented our principal ideas of implementing MSL in the ProCom component model. In this section, an example is used to illustrate this, covering all the key elements in Section 4 and demonstrating how a CBMMS can be developed in ProCom under the guidance of MSL.

5.1 System description

Consider a system to be developed in ProCom, with its component hierarchy given in Figure 1. Components d and e are ProSave components while the others are ProSys components. The system has two composite components, *Top* and b, whose basic mode mappings are given in tables 1 and 2. In each mode mapping table, the modes of different components belonging to the same column are mapped. For instance, when *Top* is running in m^1_{Top}, b is running in either m^1_b or m^3_b, and c is deactivated.

Component	Supported modes		
Top	m^1_{Top}		m^2_{Top}
a	m^1_a		m^2_a
b	m^1_b	m^3_b	m^2_b
c	Deactivated		m^1_c

Table 1: The basic mode mapping of *Top*

Component	Supported modes			
b	m^1_b	m^2_b		m^3_b
d	m^1_d	m^2_d	m^3_d	Deactivated
e	m^1_e			

Table 2: The basic mode mapping of b

It should be pointed out that tables 1 and 2 cannot present the complete mode mappings of *Top* and b which should also specify their new modes and the new modes of their subcomponents during a mode switch. The complete mode mapping can be described by Mode Mapping Automata (MMAs) which are presented in [6]. The mode mapping of a composite component must be

manually specified. The complete specification of mode mapping in ProCom requires the support of MMAs and this is left for future work.

Component b is a ProSys component composed by ProSave components. The inner component connections of b in different modes, illustrated in Figure 15(a), are treated in ProSave where control flow and data flow are separate. In contrast, Figure 15(b) illustrates the inner component connections of *Top* in m^1_{Top} and m^2_{Top} in ProSys.

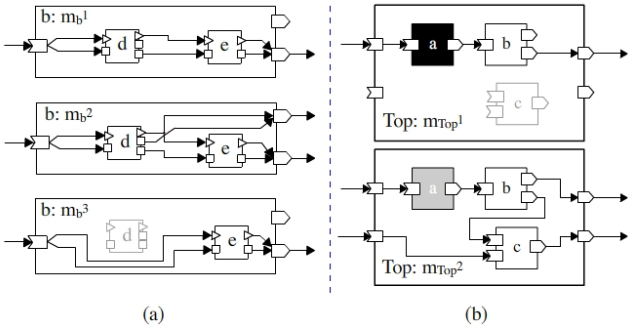


Figure 15: The inner component connections of b and *Top* in different modes

In order to develop such a CBMMS in ProCom, the first step is to define the multi-mode ProSave and ProSys components introduced in Section 4.1. Figure 16 shows the hierarchy of all multi-mode ProCom components. The dedicated mode switch ports of each component are marked in purple. Furthermore, as multi-mode ProSave components, d and e also have a dedicated service S_{mode}.

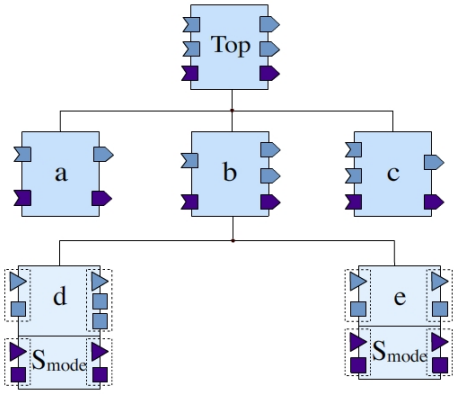


Figure 16: The component hierarchy in ProCom

5.2 The mode switch handling

In this example, a, c, d and e are primitive components, whose mode switch handling can be directly implemented as source code, following algorithms 1 and 2 in the appendix. As composite components, *Top* and b must use additional subcomponents to handle their mode switches.

Component b is a ProSys component composed by ProSave components, thus its mode switch can be handled by a pair of special subcomponents MSL_b^A and MSL_b^B, both of which can be automatically generated, given the mode mapping of b. Figure 17 presents MSL_b^A and MSL_b^B and their ports, which strictly conform to the definitions of $MSL_{c_i}^A$ and $MSL_{c_i}^B$ in Section 4.2. Please note that $MSL_b^A.P_o^{msx} = \{MSL_b^A.p_o^d, MSL_b^A.p_o^e\}$ and $MSL_b^B.P_i^{msx} = \{MSL_b^B.p_i^d, MSL_b^B.p_i^e\}$ as d and e are the subcomponents of b. Components MSL_b^A and MSL_b^B jointly handle the mode switch of b and their internal behaviors follow algorithms 3 and 4 in the appendix. Moreover, the connections around MSL_b^A and MSL_b^B are presented in Figure 19.

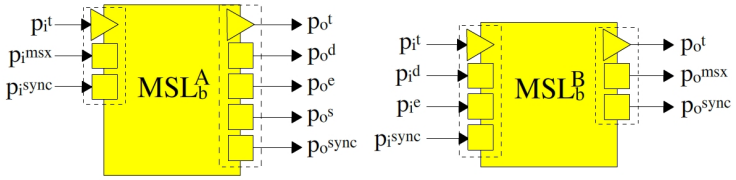


Figure 17: The port definition of MSL_b^A and MSL_b^B

Component *Top* is a ProSys component composed by ProSys components, thus its mode switch can be handled by a single special subcomponent MSL_{Top} that can be automatically generated, given the mode mapping of *Top*. Figure 18 presents MSL_{Top} and its ports, which strictly conform to the definition of MSL_{c_i} in Section 4.2. Since the subcomponents of *Top* are a, b and c, for MSL_{Top}, $P_i^{msx} = \{p_i^a, p_i^b, p_i^c\}$ and $P_o^{msx} = \{p_o^a, p_o^b, p_o^c\}$. Component MSL_{Top} handles the mode switch of *Top*. Component *Top* is a ProSys component at top level, hence the internal behavior of MSL_{Top} follows Algorithm 8 in the appendix. Moreover, the connections around MSL_{Top} are presented in Figure 20.

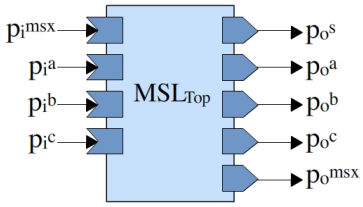


Figure 18: The port definition of MSL_{Top}

5.3 Managing the variability of component connections

Figure 15 indicates the variability of inner component connections of both b and *Top*. In order to manage such variability, we can merge their inner component connections in all modes into a complete view by adhering to the principles introduced in Section 4.3.

A complete view of the inner component connections of b is presented in Figure 19, automatically generated based on the inner component connections of b separately defined for each mode (see Figure 15(a)). This complete view

includes all the additional connectors introduced in Section 4.3, i.e. *Control Or*, *Data Or*, *Selection* and *Data Selection*. Moreover, a *Clock*, MSL_b^A and MSL_b^B, and a *Control Or* connector connected to $MSL_b^B.p_i^t$ are also generated for handling the mode switch of b. All connectors, directly connected to a port which is not dedicated to mode switch, have three input or output ports because b can run in three modes: m_b^1, m_b^2 and m_b^3. Each *Selection* or *Data Selection* has an input data port marked in red. This port is connected to $MSL_b^A.p_o^s$ which tells the current mode of b such that the correct outgoing connection is selected.

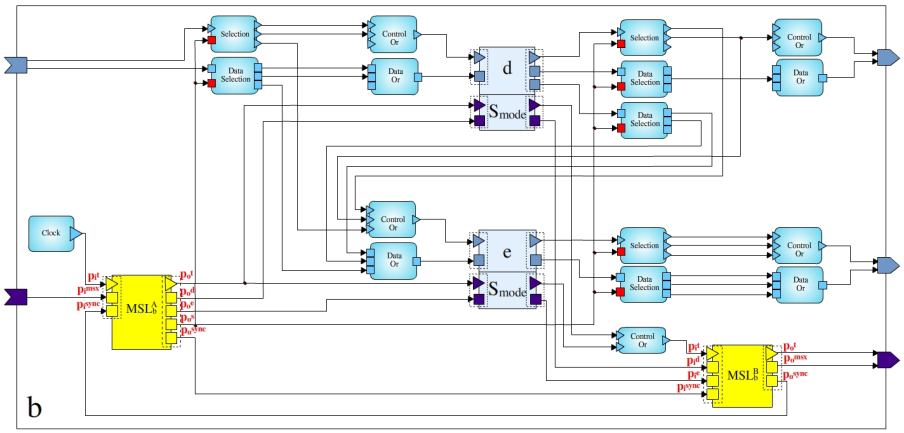


Figure 19: The complete view of inner component connections of b

A complete view of the inner component connections of *Top* is presented in Figure 20, automatically generated based on the inner component connections of *Top* separately defined for each mode (see Figure 15(b)). The complete view includes MSL_{Top} for the mode switch handling of *Top* and a couple of *Selection* ProSys components defined in Section 4.3. Each *Selection* component has two output message ports because *Top* can run in two modes: m_{Top}^1 and m_{Top}^2. Meanwhile, each *Selection* component also has a particular input message port marked in red. This port is connected to $MSL_{Top}.p_o^s$ which tells the current mode of *Top* such that the correct outgoing connection is selected.

What deserves extra attention is that the generated complete views of component connections in figures 19 and 20 are not optimized yet. By default, the generation rules assume that any connection associated with any port not dedicated to mode switch is different for different modes. However, some connections remain the same for all modes. For instance, according to Figure 15(a), the outgoing connection of e is never changed regardless of the current mode of b. Then the four generated connectors between e and the second output port of b in Figure 19 can actually be removed. Such optimization can be employed at both the ProSave and ProSys layers, thus substantially simplifying the generated complete view.

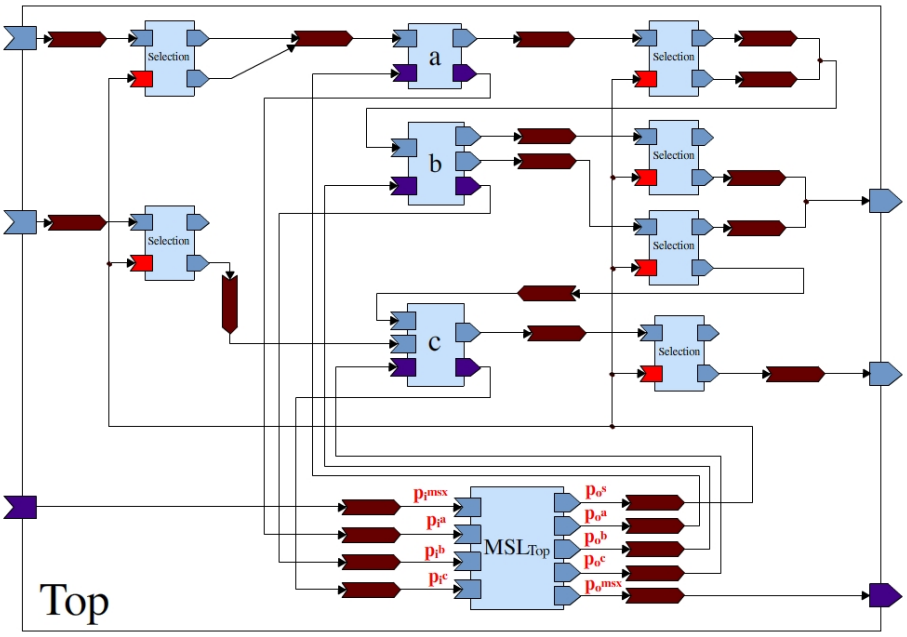


Figure 20: The complete view of inner component connections of *Top*

6 Related work

Apart from ProCom, many other component models have been proposed for the development of embedded systems, such as SaveCCM [10] (the predecessor of ProCom), COMDES-II [13] and MyCCM-HI [2], to name a few. There are also some other component models which have been commercialized, e.g. Koala [14] (targeting consumer electronics) and Rubus [9] (targeting ground vehicles). These component models have different notions about the mode switch handling. For instance, 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.

To the best of our knowledge, the extended MechatronicUML [11] by Heinzemann et al. is currently the most closely related work to our MSL. However, the extended MechatronicUML focuses more on component reconfiguration while mode is not addressed. It suggests that component reconfiguration is not only locally performed but also propagated through the component hierarchy. This is similar to the MSP protocol of MSL. In the extended MechatronicUML, the reconfiguration of a composite component is handled by two dedicated subcomponents: a *Manager* and an *Executor*, which play similar roles as the dedicated subcomponents of a composite ProCom component

here.

Another recent work related to MSL is the oracle-based approach [15] by Pop et al. concerning mode switch. 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.

7 Conclusion and future work

This report presents an approach to the mode switch handling of the ProCom component model guided by the Mode Switch Logic (MSL). It is shown that the mode switch of a Component-Based Multi-Mode System (CBMMS) can be properly handled after a slight extension of ProCom (i.e. the introduction of the *Data Selection* connector). Multi-mode ProSave and ProSys components are defined with reference to the mode-aware component model of MSL. Also, it is suggested that additional subcomponents can be used to handle the mode switch of each composite ProCom component. In order to manage the variability of component connections of a CBMMS, component connections in all modes are merged into a complete view with auxiliary elements generated at both the ProSave and ProSys layers. Thereby, each composite component is able to select the corresponding inner connections based on its current running mode. Finally, our approach is demonstrated by a simple example.

As future work, the theories presented in this report will be refined and implemented in PRIDE [1], a developing environment based on the ProCom component model. It is also envisioned that the extended PRIDE will provide a practical platform for the evaluation of both MSL and our implementation of MSL in ProCom.

References

[1] PRIDE. `http://www.idt.mdh.se/pride/?id=home`.

[2] 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*, pages 1160–1165, April 2009.

[3] T. Bureš, J. Carlson, I. Crnković, S. Sentilles, and A. Vulgarakis. ProCom - the Progress component model reference manual, version 1.0. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-230/2008-1-SE, Mälardalen University, June 2008.

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

[5] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37(5):593–615, October 2011.

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

[7] 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*, pages 183–188, June 2012.

[8] Y. Hang and H. Hansson. A mode mapping mechanism for component-based multi-mode systems. In *Proceedings of 4th Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, pages 38–45, November 2011.

[9] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K. Lundbäck. The Rubus component model for resource constrained real-time systems. In *Proceedings of 3rd International Symposium on Industrial Embedded Systems*, pages 177 –183, June 2008.

[10] H. Hansson, M. Åkerholm, I. Crnkovic, 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*, pages 627 – 635, September 2004.

[11] C. Heinzemann, C. Priesterjahn, and S. Becker. Towards modeling reconfiguration in hierarchical component architectures. In *Proceedings of 15th International ACM SIGSOFT Symposium on Component Based Software Engineering*, pages 23–28, June 2012.

[12] 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*, volume 6092 of *Lecture Notes in Computer Science*, pages 21–36. 2010.

[13] 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*, August 2007.

[14] R. V. Ommering, F. V. D. Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78 –85, March 2000.

[15] 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*, pages 93–102, June 2012.

Appendix A Algorithms for the mode switch handling in ProCom

Herein we provide a set of algorithms as a complement to Section 4.2 for describing the mode switch handling in ProCom. All these algorithms comply with the algorithms given in Chapter 6 of [6], which implement the mode switch runtime mechanism of MSL. A complete list of algorithms is provided in Table 3.

Algorithm index	Description
Algorithm 1	the mode switch handling of c_i, a primitive ProSave component
Algorithm 2	the mode switch handling of c_i, a primitive ProSys component
Algorithm 3	$MSL^A_{c_i}$; c_i is a composite ProSave component, $c_i \neq Top$
Algorithm 4	$MSL^B_{c_i}$; c_i is a composite ProSave component, $c_i \neq Top$
Algorithm 5	MSL_{c_i}; c_i is a composite ProSys component, $c_i \neq Top$
Algorithm 6	$MSL^A_{c_i}$; c_i is a composite ProSave component, $c_i = Top$
Algorithm 7	$MSL^B_{c_i}$; c_i is a composite ProSave component, $c_i = Top$
Algorithm 8	MSL_{c_i}; c_i is a composite ProSys component, $c_i = Top$

Table 3: List of algorithms

All these algorithms, except algorithms 1 and 2, describe the mode switch behaviors of $MSL^A_{c_i}$, $MSL^B_{c_i}$ or MSL_{c_i}, whose ports have been defined in Section 4.2. To simplify the presentation, we only use port name p for the associated component c_i instead of $c_i.p$. Besides, there are a number of notations deserving further explanation:

- *MSS*: a boolean variable set to true when c_i is a Mode Switch Source (MSS) (see the definition of MSS in Section 3.3).
- *MS_event*: a boolean variable set to true when c_i detects a mode switch event as an MSS.
- *Derive_new_mode*: a function returning the new mode of an MSS after a mode switch event is detected.
- $MSR(c_i, m_{c_i}, m^{new}_{c_i})$: an `MSR` from c_i, requesting to switch from m_{c_i} to $m^{new}_{c_i}$.
- $MSX(c_i, m^{new}_{c_i})$: a primitive rather than `MSR`, associated a mode switch of c_i to the new mode $m^{new}_{c_i}$.
- *Stop_running*(c_i, m_{c_i}): c_i stops running in m_{c_i}.
- *Start_running*(c_i, m_{c_i}): c_i starts running in m_{c_i}.
- *Resume*(c_i, m_{c_i}): c_i resumes execution in m_{c_i}.
- *Check_state*(c_i, m_{c_i}): c_i checks its current state while running in m_{c_i}. If its current state allows a mode switch, a boolean variable *MS_Ready* is set to true. Otherwise, *MS_Ready* is set to false.

- *Reconfiguration*$(c_i, m_{c_i}, m_{c_i}^{new})$: the reconfiguration of c_i from m_{c_i} to $m_{c_i}^{new}$.
- *Mode_Mapping*: a function performing the mode mapping of c_i.
- p_o^k: According to Section 4.2, $p_o^k \in MSL_{c_i}^A.P_o^{msx}$ $(k = [1, |SC_{c_i}|])$ and p_o^k is connected to $c_j^k \in SC_{c_i}$. In the algorithms, it is assumed that c_j^k is a Type A subcomponent of c_i (see the definition of a Type A component in Section 3.3), since c_i does not send any primitive to a Type B subcomponent.
- $\varnothing$: dummy data. When a ProSave component activates an output trigger port, it must provide data at all its data ports in the same output port group. Dummy data can be sent from an output data port if this port is not expected to send any data.
- $+$: Multiple elements are sent from a port.
- n: an integer counter initially set to 0.
- $|SC_{c_i}|_A$: the number of Type A subcomponents of c_i during a mode switch.
- $\tilde{p}$: the value of the data at port p kept from the previous activation period (only for a ProSave component).
- $p_1|p_2 := x$: data x is written to ports p_1 and p_2.
- *Get_data*(p_i^{sync}): $MSL_{c_i}^B$ gets data from its port p_i^{sync} if $p_i^{sync} \neq \varnothing$.
- *MSOK_all*: a boolean variable set to true when all Type A subcomponents of c_i reply with an `MSOK` in response to `MSQ`.

Moreover, please note that the execution status of c_i in ProSave layer is controlled by $MSL_{c_i}^A$ rather than $MSL_{c_i}^B$, e.g. by stoping or resuming the execution of c_i in its current mode or starting the execution of c_i in its new mode. This can be realized in different ways. For instance, one option is to share properties between $MSL_{c_i}^A$ and c_i. A concrete plan is out of the scope of this report.

A mode switch scenario based on the example in Section 5 can be used to demonstrate how these algorithms typically work. Let's reuse the scenario depicted in Figure 7, where b is an MSS which requests to switch mode by issuing a primitive `MSR` to *Top*. Components *Top*, *a*, *b*, *c*, *d* are Type A components for this scenario while *e* is a Type B component. Since *Top* is a ProSys component at top level, MSL_{Top} follows Algorithm 8. Since b is a non-top composite ProSys component composed by ProSave components, MSL_b^A follows Algorithm 3 and MSL_b^B follows Algorithm 4. Then taking figures 7, 19 and 20 into account, this scenario leads to the following procedures:

1. Component b detects a mode switch event as an MSS, hence an `MSR` is sent from $MSL_b^B.p_o^{msx}$ to $b.p_o^{ms}$, and then to $MSL_{Top}.p_i^b$.
2. According to Algorithm 8, MSL_{Top} will perform its mode mapping and realize that this `MSR` implies the mode switches of all its subcomponents and itself. After checking the current state of c_i which allows a mode switch, MSL_{Top} will send an `MSQ` to *a*, *b*, *c*. As is indicated in Figure 20, this `MSQ` is sent from $MSL_{Top}.p_o^a$ to $a.p_i^{ms}$, from $MSL_{Top}.p_o^b$ to $b.p_i^{ms}$, and from $MSL_{Top}.p_o^c$ to $c.p_i^{ms}$, respectively.

3. As primitive ProSys components, a and c both follow Algorithm 2. When a receives the `MSQ`, it stops its current execution and checks if its current state allows a mode switch. The checking result is positive, thus a sends an `MSOK` to *Top* as the feedback. The `MSOK` is sent from $a.p_o^{ms}$ to $MSL_{Top}.p_i^a$. Likewise, another `MSOK` is sent from $c.p_o^{ms}$ to $MSL_{Top}.p_i^c$.

4. Meanwhile, the `MSQ` from *Top* to b is propagated from $b.p_i^{ms}$ to $MSL_b^A.p_i^{msx}$. According to Algorithm 3, MSL_b^A will stop the execution of b and check its current state. Since the current state of b allows a mode switch, MSL_b^A performs the mode mapping of b and realizes that d is a Type A component and e is a Type B component. Therefore, MSL_b^A sends an `MSQ` to d (via $d.m_i^{mst}$ and $d.m_i^{ms}$) and also let MSL_b^B know $|SC_b|_A$ and m_b^{new} by sending them from $MSL_b^A.p_o^{sync}$ to $MSL_b^B.p_i^{sync}$. Simultaneously, dummy data is sent from $MSL_b^A.p_o^e$ to $e.p_i^{ms}$.

5. As primitive ProSave components, d and e both follow Algorithm 1. When d receives the `MSQ`, it stops its current execution and checks its current state. The checking result is positive, thus d sends an `MSOK` to b as the feedback. The `MSOK` is sent from $d.p_o^{ms}$ to $MSL_b^B.p_i^d$. Simultaneously, after receiving dummy data, e also sends dummy data to MSL_b^B via $MSL_b^B.p_i^e$.

6. According to Algorithm 4, after receiving an `MSOK` from d, MSL_b^B gets to know that it has only one Type A subcomponent from the data written to $MSL_b^B.p_i^{sync}$. Since b expects only one `MSOK`, b will also send an `MSOK` to *Top*. This corresponds to sending the `MSOK` from $MSL_b^B.p_o^{msx}$ to b_o^{ms} and then to $MSL_{Top}.p_i^b$.

7. According to Algorithm 8, *Top* has received all expected primitives `MSOK`. As a Type A component, *Top* will start its reconfiguration which is performed in MSL_{Top}, which also triggers a mode switch by sending an `MSI` to a, b and c. The propagation trace of this `MSI` is exactly the same as the `MSQ`.

8. After receiving the `MSI`, a and c will start their reconfiguration and then send an `MSC` to *Top*. When b receives the `MSI`, it starts its reconfiguration and sends another `MSI` to its Type A subcomponent d. This `MSI` is sent from $MSL_b^A.p_o^d$ to $d.p_i^{ms}$. Meanwhile, dummy data is sent from MSL_b^A to e.

9. Upon receiving the `MSI`, d will start its reconfiguration and then send an `MSC` to b (via $MSL_b^B.p_i^d$). This `MSC` will be forwarded from MSL_b^B to MSL_{Top}. When *Top* has received all expected primitives `MSC` and completed its reconfiguration, the system will complete its mode switch.

Algorithm 1 *Primitive_ProSave_MS*

```
loop
  if p_i^mst then
    if c_i = MSS ∧ MS_event then
      m_ci^new := Derive_new_mode;
      p_o^ms := MSR(c_i, m_ci, m_ci^new);
    else if p_i^msx = p_i^msx then
      p_o^ms := ∅;
    else
      if p_i^ms = MSQ then
        Stop_running(c_i, m_ci);
        Check_state(c_i, m_ci);
        if MS_Ready then
          p_o^ms := MSOK(c_i, m_ci^new);
        else
          p_o^ms := MSNOK(c_i, m_ci^new);
        end if
      end if
      if p_i^ms = MSI then
        Reconfiguration(c_i, m_ci, m_ci^new);
        p_o^ms := MSC(c_i, m_ci^new);
        Start_running(c_i, m_ci^new);
      end if
      if p_i^ms = MSD then
        Resume(c_i, m_ci);
      end if
    end if
    p_i^mst := false;
    p_o^mst := true;
  end if
end loop
```

Algorithm 2 *Primitive_ProSys_MS*

loop
  if $c_i = MSS \wedge MS\_event$ **then**
    $m_{c_i}^{new} := Derive\_new\_mode$;
    $p_o^{ms} := MSR(c_i, m_{c_i}, m_{c_i}^{new})$;
  else
    if $p_i^{ms} = MSQ$ **then**
      $Stop\_running(c_i, m_{c_i})$;
      $Check\_state(c_i, m_{c_i})$;
      if *MS_Ready* **then**
        $p_o^{ms} := MSOK(c_i, m_{c_i}^{new})$;
      else
        $p_o^{ms} := MSNOK(c_i, m_{c_i}^{new})$;
      end if
    end if
    if $p_i^{ms} = MSI$ **then**
      $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
      $p_o^{ms} := MSC(c_i, m_{c_i}^{new})$;
      $Start\_running(c_i, m_{c_i}^{new})$;
    end if
    if $p_i^{ms} = MSD$ **then**
      $Resume(c_i, m_{c_i})$;
    end if
  end if
end loop

Algorithm 3 $MSL^{A}_{c_i}(c_i \neq Top)$

loop
  if p_i^t **then**
    if $p_i^{sync} = MSR$ **then**
      $Mode\_mapping$;
      if $m_{c_i}^{new} \neq m_{c_i}$ **then**
        $Stop\_running(c_i, m_{c_i})$;
        $p_o^s := \varnothing$;
      else
        $p_o^s := m_{c_i}$;
      end if
      $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
      $p_o^{sync} := \varnothing$;
    end if
    if $p_i^{sync} = MSI$ **then**
      if $m_{c_i}^{new} = m_{c_i}$ **then**
        $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
      end if
      $\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
      $p_o^s := \tilde{p_o^s}$;
      $p_o^{sync} := \varnothing$;
    end if
    if $p_i^{sync} = MSD$ **then**
      $\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
      $p_o^s := m_{c_i}$;
      $p_o^{sync} := \varnothing$;
    end if
    if $p_i^{sync} = MSC$ **then**
      if $m_{c_i}^{new} = m_{c_i}$ **then**
        $p_o^s := m_{c_i}$;
      else
        $Start\_running(c_i, m_{c_i}^{new})$;
        $p_o^s := m_{c_i}^{new}$;
      end if
      $P_o^{msx} | p_o^{sync} := \varnothing$;
    end if
    if $p_i^{msx} = MSQ$ **then**
      $Stop\_running(c_i, m_{c_i})$;
      $Check\_state(c_i, m_{c_i})$;
      if MS_Ready **then**
        $Mode\_mapping$;
        $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
        $p_o^s := \varnothing$;
        $p_o^{sync} := |SC_{c_i}|_A + m_{c_i}^{new}$;
      else
        $Resume(c_i, m_{c_i})$;
        $P_o^{msx} | p_o^s := \varnothing$;
        $p_o^{sync} := MSNOK(c_i, m_{c_i}^{new})$;
      end if
    end if

if $p_i^{msx} = MSI$ **then**
$Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
$\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
$p_o^s | p_o^{sync} := \varnothing$;
end if
if $p_i^{msx} = MSD$ **then**
$Resume(c_i, m_{c_i})$;
$\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
$p_o^s := m_{c_i}$;
$p_o^{sync} := \varnothing$;
end if
if $p_i^{msx} = \tilde{p_i^{msx}} \ \wedge p_i^{sync} = \tilde{p_i^{sync}}$ **then**
$p_o^s := \tilde{p_o^s}$;
$p_o^{msx} | p_o^{sync} := \varnothing$;
end if
$p_i^t := false$;
$p_o^t := true$;
end if
end loop

Algorithm 4 $MSL^B_{c_i}(c_i \neq Top)$

loop
  if p_i^t **then**
    if $c_i = MSS \wedge MS\_event$ **then**
      $m_{c_i}^{new} := Derive\_new\_mode$;
      $p_o^{msx} := MSR(c_i, m_{c_i}, m_{c_i}^{new})$;
    else
      if $\exists p_i^k = MSR$ **then**
        $Mode\_mapping$;
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $p_o^{msx} := \varnothing$;
          $p_o^{sync} := MSR(c_j^k, m_{c_j^k}, m_{c_j^k}^{new})$;
        else
          $Check\_state(c_i, m_{c_i})$;
          if MS_Ready **then**
            $p_o^{msx} := MSR(c_i, m_{c_i}, m_{c_i}^{new})$;
          else
            $p_o^{msx} := \varnothing$;
          end if
          $p_o^{sync} := \varnothing$;
        end if
      end if
      if $\exists p_i^k = MSOK \vee \exists p_i^k = MSNOK$ **then**
        $Get\_data(p_i^{sync})$;
        $n++$;
        if $n = |SC_{c_i}| \wedge MSOK\_all$ **then**
          if $m_{c_i}^{new} = m_{c_i}$ **then**
            $p_o^{msx} := \varnothing$;
            $p_o^{sync} := MSI$;
          else
            $p_o^{msx} := MSOK(c_i, m_{c_i}^{new})$;
            $p_o^{sync} := \varnothing$;
          end if
          $n := 0$;
        end if
        if $n = |SC_{c_i}| \wedge \neg MSOK\_all$ **then**
          if $m_{c_i}^{new} = m_{c_i}$ **then**
            $p_o^{msx} := \varnothing$;
            $p_o^{sync} := MSD$;
          else
            $p_o^{msx} := MSNOK(c_i, m_{c_i}^{new})$;
            $p_o^{sync} := \varnothing$;
          end if
          $n := 0$;
        end if
      end if

if $\exists p_i^k = MSC$ **then**
$n++$;
if $n = |SC_{c_i}|_A$ **then**
if $m_{c_i}^{new} = m_{c_i}$ **then**
$p_o^{msx} := \varnothing$;
$p_o^{sync} := MSC$;
else
$Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
$p_o^{msx}|p_o^{sync} := MSC(c_i, m_{c_i}^{new})$;
end if
$n := 0$;
end if
end if
if $p_i^{sync} = MSNOK$ **then**
$p_o^{msx} := MSNOK(c_i, m_{c_i}^{new})$;
$p_o^{sync} := \varnothing$;
end if
if $p_i^{msx} = p_i^{\tilde{m}sx} \wedge p_i^{sync} = p_i^{s\tilde{y}nc}$ **then**
$p_o^{msx}|p_o^{sync} := \varnothing$;
end if
end if
$p_i^t := false$;
$p_o^t := true$;
end if
end loop

Algorithm 5 $MSL_{c_i}(c_i \neq Top)$

loop
  if $c_i = MSS \wedge MS\_event$ **then**
    $m_{c_i}^{new} := Derive\_new\_mode$;
    $p_o^{msx} := MSR(c_i, m_{c_i}, m_{c_i}^{new})$;
  else
    if $\exists p_i^k = MSR$ **then**
      $Mode\_mapping$;
      if $m_{c_i}^{new} = m_{c_i}$ **then**
        $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
      else
        $Check\_state(c_i, m_{c_i})$;
        if MS_Ready **then**
          $p_o^{msx} := MSR(c_i, m_{c_i}, m_{c_i}^{new})$;
        end if
      end if
    end if
    if $p_i^{msx} = MSQ$ **then**
      $Stop\_running(c_i, m_{c_i})$;
      $Check\_state(c_i, m_{c_i})$;
      if MS_Ready **then**
        $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
        $p_o^s := \varnothing$;
      else
        $p_o^{msx} := MSNOK(c_i, m_{c_i}^{new})$;
      end if
    end if
    if $\exists p_i^k = MSOK \vee \exists p_i^k = MSNOK$ **then**
      $n++$;
      if $n = |SC_{c_i}| \wedge MSOK\_all$ **then**
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
        else
          $p_o^{msx} := MSOK(c_i, m_{c_i}^{new})$;
        end if
        $n := 0$;
      end if
      if $n = |SC_{c_i}| \wedge \neg MSOK\_all$ **then**
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
          $Resume(c_i, m_{c_i})$;
          $p_o^s := m_{c_i}$;
        else
          $p_o^{msx} := MSNOK(c_i, m_{c_i}^{new})$;
        end if
        $n := 0$;
      end if
    end if

if $p_i^{msx} = MSI$ **then**
$\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
$Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
end if
if $\exists p_i^k = MSC$ **then**
$n++$;
if $n = |SC_{c_i}|_A$ **then**
if $m_{c_i}^{new} = m_{c_i}$ **then**
$p_o^s := m_{c_i}$;
else
$p_o^{msx} := MSC(c_i, m_{c_i}^{new})$;
$p_o^s := m_{c_i}^{new}$;
$Start\_running(c_i, m_{c_i}^{new})$;
end if
$n := 0$;
end if
end if
if $p_i^{msx} = MSD$ **then**
$\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
$p_o^s := m_{c_i}$;
$Resume(c_i, m_{c_i})$;
end if
end if
end loop

Algorithm 6 $MSL^{A}_{c_i}(c_i = Top)$

loop
  if p_i^t **then**
    if $c_i = MSS \wedge MS\_event$ **then**
      $m_{c_i}^{new} := Derive\_new\_mode$;
      $Mode\_mapping$;
      $Stop\_running(c_i, m_{c_i})$;
      $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
      $p_o^s | p_o^{sync} := \varnothing$;
    else if $p_i^{msx} = p_i^{\tilde{m}sx} \ \wedge p_i^{sync} = p_i^{s\tilde{y}nc}$ **then**
      $p_o^s := \tilde{p_o^s}$;
      $P_o^{msx} | p_o^{sync} := \varnothing$;
    else
      if $p_i^{sync} = MSR$ **then**
        $Mode\_mapping$;
        if $m_{c_i}^{new} \neq m_{c_i}$ **then**
          $Stop\_running(c_i, m_{c_i})$;
          $p_o^s := \varnothing$;
        else
          $p_o^s := m_{c_i}$;
        end if
        $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
        $p_o^{sync} := \varnothing$;
      end if
      if $p_i^{sync} = MSI$ **then**
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
        end if
        $\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
        $p_o^s := \tilde{p_o^s}$;
        $p_o^{sync} := \varnothing$;
      end if
      if $p_i^{sync} = MSD$ **then**
        $\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
        $p_o^s := m_{c_i}$;
        $p_o^{sync} := \varnothing$;
      end if
      if $p_i^{sync} = MSC$ **then**
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $p_o^s := m_{c_i}$;
        else
          $Start\_running(c_i, m_{c_i}^{new})$;
          $p_o^s := m_{c_i}^{new}$;
        end if
        $P_o^{msx} | p_o^{sync} := \varnothing$;
      end if
    end if
    $p_i^t := false$;
    $p_o^t := true$;
  end if
end loop

Algorithm 7 $MSL^{B}_{c_i}(c_i = Top)$

loop
  if p_i^t **then**
    if $\exists p_i^k = MSR$ **then**
      $Mode\_mapping$;
      if $m_{c_i}^{new} = m_{c_i}$ **then**
        $p_o^{msx} := \varnothing$;
        $p_o^{sync} := MSR(c_j^k, m_{c_j^k}, m_{c_j^k}^{new})$;
      else
        $Check\_state(c_i, m_{c_i})$;
        if MS_Ready **then**
          $p_o^{sync} := MSR(c_j^k, m_{c_j^k}, m_{c_j^k}^{new})$;
        else
          $p_o^{sync} := \varnothing$;
        end if
        $p_o^{msx} := \varnothing$;
      end if
    end if
    if $\exists p_i^k = MSOK \vee \exists p_i^k = MSNOK$ **then**
      $Get\_data(p_i^{sync})$;
      $n++$;
      if $n = |SC_{c_i}| \wedge MSOK\_all$ **then**
        $p_o^{msx} := \varnothing$;
        $p_o^{sync} := MSI$;
        $n := 0$;
      end if
      if $n = |SC_{c_i}| \wedge \neg MSOK\_all$ **then**
        $p_o^{msx} := \varnothing$;
        $p_o^{sync} := MSD$;
        $n := 0$;
      end if
    end if
    if $\exists p_i^k = MSC$ **then**
      $n++$;
      if $n = |SC_{c_i}|_A$ **then**
        if $m_{c_i}^{new} \neq m_{c_i}$ **then**
          $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
        end if
        $n := 0$;
      end if
    end if
    if $p_i^{msx} = p_i^{\tilde{m}sx} \ \wedge p_i^{sync} = p_i^{s\tilde{y}nc}$ **then**
      $p_o^{msx} | p_o^{sync} := \varnothing$;
    end if
    $p_i^t := false$;
    $p_o^t := true$;
  end if
end loop

Algorithm 8 $MSL_{c_i}(c_i = Top)$

loop
  if $c_i = MSS \wedge MS\_event$ **then**
    $m_{c_i}^{new} := Derive\_new\_mode$;
    $Mode\_mapping$;
    $Stop\_running(c_i, m_{c_i})$;
    $p_o^s := \varnothing$;
    $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
  else
    if $\exists p_i^k = MSR$ **then**
      $Mode\_mapping$;
      if $m_{c_i}^{new} = m_{c_i}$ **then**
        $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
      else
        $Check\_state(c_i, m_{c_i})$;
        if MS_Ready **then**
          $\forall p_o^k := MSQ(c_j^k, m_{c_j^k}^{new})$;
          $p_o^s := \varnothing$;
        end if
      end if
    end if
    if $\exists p_i^k = MSOK \vee \exists p_i^k = MSNOK$ **then**
      $n++$;
      if $n = |SC_{c_i}| \wedge MSOK\_all$ **then**
        if $m_{c_i}^{new} \neq m_{c_i}$ **then**
          $Reconfiguration(c_i, m_{c_i}, m_{c_i}^{new})$;
        end if
        $\forall p_o^k := MSI(c_j^k, m_{c_j^k}^{new})$;
        $n := 0$;
      end if
      if $n = |SC_{c_i}| \wedge \neg MSOK\_all$ **then**
        $Resume(c_i, m_{c_i})$;
        $\forall p_o^k := MSD(c_j^k, m_{c_j^k}^{new})$;
        $p_o^s := m_{c_i}$;
        $n := 0$;
      end if
    end if
    if $\exists p_i^k = MSC$ **then**
      $n++$;
      if $n = |SC_{c_i}|_A$ **then**
        if $m_{c_i}^{new} = m_{c_i}$ **then**
          $p_o^s := m_{c_i}$;
        else
          $p_o^s := m_{c_i}^{new}$;
          $Start\_running(c_i, m_{c_i}^{new})$;
        end if
        $n := 0$;
      end if
    end if
  end if
end loop