LUND UNIVERSITY

MÄLARDALEN UNIVERSITY
SWEDEN

MASTER OF SCIENCE THESIS

The Mode Switch Logic implementation in the ProCOM component model

By
Hongwan Qin
mas09hqi@student.lu.se

Supervised by
Mr. Hang Yin, Mälardalen University
Prof. Hans Hansson, Mälardalen University
Examined by Prof. Peter Nilsson, Lund University

MÄLARDALEN REAL-TIME RESEARCH CENTRE
MÄLARDALEN UNIVERSITY
&
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING, LTH, LUND UNIVERSITY

Lund 2012

©

The Department of Electrical and Information Technology
Lund University
Box 118, S-221 00 LUND
SWEDEN

This thesis is set in Times New Roman 11pt,
with the LaTeX Documentation System

Printed in Sweden by Tryckerite E-huset, Lund.
Nov 2012

Abstract

This thesis is based on a project which is provided by Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University. It stems from two separate existing works: ProCom and Mode Switch Logic (MSL). Both works are strongly related to Component-Based Software Engineering (CBSE).

Since traditional software development is becoming increasingly large and complex, to solve this problem, CBSE becomes a rapidly developing discipline and more efficient method compared with classical approaches to producing high quality software both in academia and industry. It is being more and more applied to industrial strength and mission-critical software. CBSE has already been endorsed by many industrial applications.

ProCom, the PROGRESS Component Model for real-time embedded systems, is developed at MRTC in the PROGRESS project funded by the Swedish Foundation for Strategic Research (SSF), focusing on component-based development of real-time embedded systems. ProCom targets the domains of vehicular systems, automation and telecom. It takes the advantage of both CBSE and Model-Driven Engineering (MDE). In terms of CBSE, it embodies component reusability. In terms of MDE, it supports automatic code generation and allows system analysis at an early stage. A system can be designed by reusable components, which can be mapped on a physical node in a subsequent deployment phase.

MSL, also developed at MRTC, handles the mode switch of component-based systems. In contrast to CBSE, an alternative to reduce system complexity is to partition the system behavior into different operational modes. A multi-mode system can switch between different modes when some condition changes. If a multi-mode system is component-based, its mode switch is not a trivial problem. MSL provides an effective mode switch mechanism for component-based systems.

The contribution of this thesis is that it presents how to implement MSL in the ProCom component model. ProCom does not support multiple modes and mode switch. Therefore, in order to implement MSL, ProCom must be extended. In this thesis, we present our solutions to achieving such an implementation.

Acknowledgement

I would like to express my heartfelt gratitude and acknowledgement to the people who have helped, encouraged and supported me while I was doing my master project and writing the thesis. I really want to say that it is truly hard to take the thesis project at night and at weekends together with my full time working in Sony Mobile since the beginning of this year, however, I am so lucky and appreciate that I have all of you around me till the final stage of this thesis.

Foremost, I am grateful to my examiner, Professor Peter Nilsson, at Lund University for his continuous care and support for my thesis. I not only enjoyed his lectures but also am so proud of being one of his master students. Without his imparting knowledge to us and what I have learned at Lund University, I would never have completed this thesis.

Special thanks to my another examiner Professor Hans Hansson, co-supervisor PhD student Hang Yin, and Doctor Jan Carlsson at Mälardalen Real-Time Research Centre at Mälardalen University, for their long-standing support, guidance and invaluable help during the whole period of this master project. They always point out the right direction and provide generous suggestions to me. I also want to say thanks to Professor Hans who allows me to take this thesis in parallel with my work at Sony Mobile.

Many thanks go to my colleagues in Companion Products of Sony Mobile in Lund, where I have ever worked for two years. My best regards go to my colleagues in the Department of Business Management, Business Control & Planning and Supply & Demand planning, where I serviced and supported these three departments. They treat me as an intimate friend and help me when I am in trouble, bringing joy into my life so that I could get my energy fully charged focus on my study after work. Also, I appreciate the director of Creation, Planning, Business and Global Marketing in Companion Products of Sony Mobile, Michael Henriksson for hiring me into this great company.

I also would like to give my thanks to my classmates at Lund University, and my friends from all over the world, who strongly believe in me and treasure our friendship. You make my life colorful and delightful in Sweden! My deepest gratitude to my parents and my twin brother, who always tell me that I have a warm and sweet home no matter what happens and the door is always open for me, and then my heart with pleasure fills.

Lund, November 6th, 2012
Hongwan Qin

Contents

List of Tables

List of Figures

List of Acronyms

CBSE	Component-Based Software Engineering
MSL	Mode Switch Logic
CBMMS	Component-Based Multi-Mode System
DDM	Dominant Default Mode
MMA	Mode Mapping Automata
MSS	Mode Switch Source
MSDM	Mode Switch Decision Maker
MSR	Mode Switch Request
MSI	Mode Switch Instruction
MSC	Mode Switch Completion
CBD	Component-Based Development
RTOS	Real-Time Operating System

Chapter 1

Introduction

This first chapter outlines the theoretical background of this thesis, including Component-Based Software Engineering (CBSE), the ProCom component model and the Mode Switch Logic (MSL) which serve as the input of this thesis. We then point out the methodology of this thesis. Finally, an overview of the thesis is introduced by briefly describing its structure.

1.1 Theoretical background

Embedded systems are computer systems dedicated to specific functionalities, often with limited constraints. Along with the progress of time and the technical development, embedded systems have spread themselves everywhere around our life and work. While embedded systems are providing more and more advanced functionalities, their software complexity has as a consequence been raised significantly. To handle this complexity, a typical approach is to partition the system behavior into different operational modes. Moreover, another approach to reducing software complexity is Component-Based Software Engineering (CBSE). Combining these two techniques, a multi-mode system can be developed in a component-based manner. A challenge of doing this is the mode switch handling. The mode switch of a system can be considered as the change of its configuration in one mode to a different configuration in another mode. The theoretical foundation of mode switch handling for such kind of systems has been built by the Mode Switch Logic (MSL) [1]. The goal of this thesis is to implement MSL in the ProCom component model. ProCom [2] is a component model for real-time and embedded systems, particularly targeting the domains of vehicular and telecommunication applications, developed at Mälardalen Real-Time Research Centre (MRTC) at Mälardalen University.

1.1.1 Component-Based Software Engineering

Component-Based Software Engineering (CBSE) has become recognized as a new technology of building software focusing on the component aspects of software development. This promising design paradigm is used for the development of complex systems from reusable software components. CBSE emerged from the failure of object-oriented development to support reuse effectively [3]. Components can be grouped with different explicit goals to allow them to be generalized and reused. A component is a software unit whose functionality and dependencies are completely defined by its interfaces. A component model defines a set of standards that component providers and composers should follow [4]. Apart from complexity management, CBSE can also increase productivity, reduce the time to market and improve the software quality.

1.1.2 The ProCom component model

Developed within the PROGRESS project, the ProCom component model aims for the software development of real-time and embedded systems, in particular, vehicular and telecommunication systems. ProCom has been supported by its development tool PRIDE [5] which can generate codes from components. ProCom is organized in two distinctive layers: the top layer ProSys and the lower layer ProSave. ProSys is used to model subsystems that can execute concurrently. The communication at the ProSys layer is realized by asynchronous message passing. In contrast, ProSave is dedicated to the detailed design of each subsystem. Components at the ProSave layer follow the pipe-and-filter architectural style. Data and control flows are clearly separated. All ProSave components follow the same execution pattern: (1) Read all input data when the associated trigger port is activated; (2) Become active and perform the computation; (3) Produce the output data and activate the associated output trigger port. And then the component becomes passive again. A ProSave component can have multiple services providing different functionalities, and a variety of connectors have been defined for the communication between ProSave components. Both the ProSys and ProSave layers are hierarchical and it is allowed to compose a ProSys component by ProSave components. Moreover, the behavior of a ProSave component at the bottom level is implemented as a C function.

Currently, ProCom does not support multi-mode components and mode switch. In this thesis, ProCom will be extended for the handling of mode switch.

1.1.3 The Mode Switch Logic (MSL)

A multi-mode system exhibits different behaviors in different operational modes. Such kind of system is supposed to switch to the most suitable mode when some condition changes. A mode switch can be triggered by a particular event or tim-

ing. An example of multi-mode system is the control software of an airplane which normally runs in *taxi* mode, *taking off* mode, *flight* mode and *landing* mode. So far not so much attention has been paid to the integration of multi-mode systems and CBSE. Traditional component models do not include handling of operational modes, and traditional handling of operational modes does not assume system built from reusable components. Combining both types of systems into one will introduce a new type of system, which we call Component-Based Multi-Mode System (CBMMS), i.e. a multi-mode system built by reusable components. In order to explain what a CBMMS is and how it works, an example is presented in Fig. 1.1, which illustrates the hierarchical component structure of a typical CBMMS in the left part. Meanwhile, the component connections of the same system is shown in the right part. This system is composed by three components: *a*, *b*, and *c*. Component *b* consists of two subcomponents: *d* and *e*. Since the component hierarchy has a tree structure, here *b* is the parent while *d* and *e* are the children of *b*. According to the terminology of CBSE, Component *a*, *c*, *d*, and *e* are primitive components, which are directly implemented by software codes and cannot be decomposed into other components; Component *Top* and *b* are composite components, which consist of other components. Furthermore, the system supports two modes: $m_{Top}1$ and m^2_{Top}. In $m_{Top}1$, Component *c* is not running and Component *a* is executing a mode-specific behavior (indicated by the black color in Fig. 1.1); in m^2_{Top}, *c* is activated but *e* becomes deactivated, and *a* changes its behavior (indicated by the grey color). In addition, the right part of Fig. 1.1 depicts the component connections further.

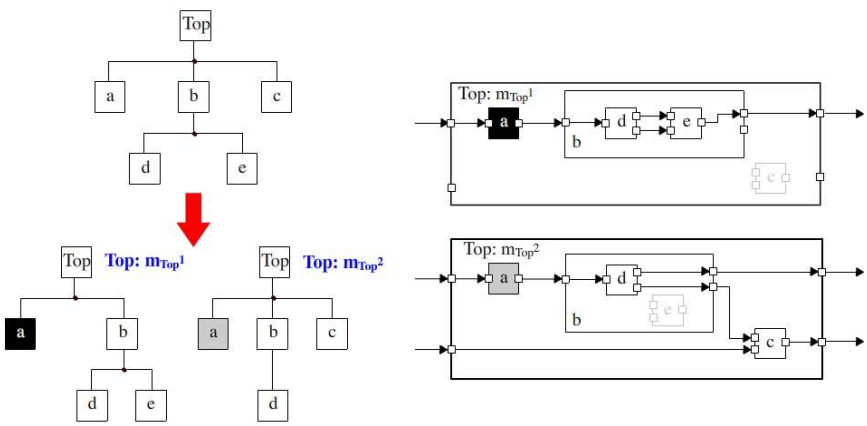


Figure 1.1: A component-based multi-mode system

The mode switch of a CBMMS is characterized by the joint mode switches of different components. Since the mode switches of different components can be either independent or correlated, the challenge comes from the synchronization and coordination of the mode switches of related components. The Mode Switch Logic

(MSL) [1], developed at Mälardalen Real-Time Research Center at Mälardalen University, intends to provide efficient solutions for the mode switch handling of CBMMSs. MSL proposes a mode-aware component model which enables the composition of multi-mode components and mode switch. Besides, a mode mapping mechanism is used to specify which mode each component should switch to once a mode switch is triggered. Furthermore, MSL includes a mode switch runtime mechanism that is able to efficiently handle the mode switch of a CBMMS at runtime.

1.2 Methodology

In this thesis, our prime aim is to provide the essential theories for the implementation of MSL in ProCom. First, our work starts from the reading of literatures related to both ProCom and MSL. This plays an important role in understanding the purpose of this thesis. After that we try to grasp the key features of both background works and find out the possibilities of extending ProCom to support MSL. Different solutions have been investigated and evaluated by trying them out on small examples. Moreover, regular discussions with Hang Yin, Jan Carlson and Hans Hansson from Mälardalen University have contributed a lot to the progress of this work.

1.3 Thesis layout

We divide this thesis into six parts. Chapter 1 provides the general theoretical background for this thesis. Chapter 2 gives a brief introduction of MSL. In Chapter 3, the ProCom component model is explained. As the main contribution of the thesis, Chapter 4 describes the implementation of MSL in ProCom in detail. The central ideas presented in Chapter 4 will then be demonstrated by a pedagogical example in Chapter 5. Finally, we summarize the thesis and discuss future work in Chapter 6.

Chapter 2

The Mode Switch Logic

How can we handle the mode switch of a Component-Based Multi-Mode System (CBMMS)? One solution to this is the Mode Switch Logic (MSL) [1]. In this section we provide a detail introduction of MSL and its major elements, including the mode-aware component model, the mode mapping mechanism, and the mode switch runtime mechanism. Since MSL is still not mature enough, MSL will be extended to handle additional aspects in the future.

2.1 The mode-aware component model

The mode-aware component model defines the essential features that a component model should have to support multi-mode and mode switch. A multi-mode component should include a set of unique configurations associated with its unique behaviors in each mode. Different components should be able to exchange mode switch information with each other either directly or indirectly because the mode switch of one component may imply the mode switches of other components. Fig. 2.1 illustrates the mode-aware component model. In general, a multi-mode component supports multiple modes, each mode being associated with a configuration. The mode switch of such a component is realized by its reconfiguration, i.e. the switch from the configuration in the old mode to another configuration in the new mode. The mode switch is controlled by the mode switch runtime mechanism implemented in the component. The configuration and mode switch runtime of primitive components and composite components are different, and more details can be found in [1]. Just like most other port-based component models, the mode-aware component model defines a number of input ports and output ports which are used to communicated with other components. Besides, a primitive multi-mode component has a dedicated mode switch port (represented by p^{MSX} in Fig. 2.1) for exchanging mode switch information with its parent. A composite component multi-mode component has

two dedicated mode switch ports. Apart from p^{MSX}, p_{in}^{MSX} is used for a composite multi-mode component for exchanging mode switch information with its children. The port p_{in}^{MSX} in Fig. 2.1 is marked in grey to indicate that it is only included in a composite multi-mode component.

Since the mode-ware component model is not dependent on any existing component models, it can guide many existing component models for the mode switch extension.

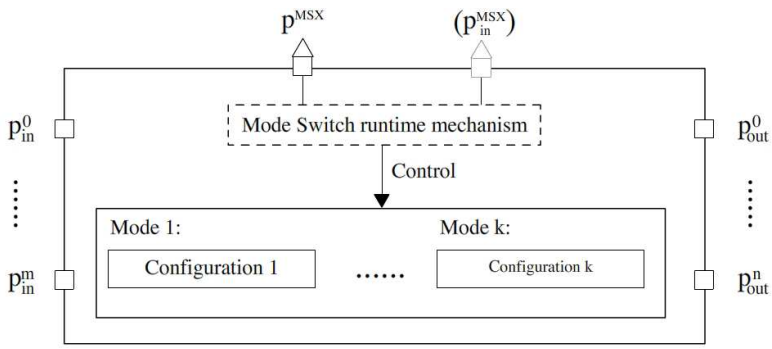


Figure 2.1: The mode-aware component model

2.2 The mode mapping mechanism

Usually a multi-mode component is independently developed without assuming the context where it will be used. When several multi-mode components are collected to compose a bigger component, it is most likely that their supported modes are different. This mode incompatibility problem is solved by the mode mapping mechanism of MSL [6]. Mode mapping has two major purposes: (1) To map the modes of a parent and its children; (2) To define the new modes of each component when it is asked to switch mode. The mode mapping mechanism is proposed by adhering to the following principles:

- Each component (primitive or composite) knows its supported modes, its initial mode and its current mode,but knows nothing about the mode information of other components in the system.
- Additionally, composite components know the entire mode information of their subcomponents, but they have no mode information of components at deeper nested levels.

According to the mode mapping mechanism, each composite component has a number of mode mapping rules for the mode mapping between itself and its subcomponents. These mode mapping rules can be further divided into static mode mapping

rules and dynamic mode mapping rules. Static mode mapping rules define the mode mapping in stable modes and they can be represented by the *mode mapping table* [1]. Dynamic mode mapping rules define the Dominant Default Modes (DDMs) for each component. When a component is asked to switch mode, it must know which new mode to switch to, and this new mode is called the DDM. In order to represent both static and dynamic mode mapping rules, Mode Mapping Automata (MMA) is designed. The mode mapping rules of a composite component are represented by a set of MMA including the MMA of the composite component and the MMA of its subcomponents. Each MMA has locations and transitions. Each location corresponds to a supported mode of the component. Each transition corresponds to a mode switch. A transition is triggered by an input signal and it can produce output signals. A signal can be either internal or external. An internal signal is used to synchronize different MMA and an external signal is used for a parent and its subcomponents to exchange mode switch information. Fig. 2.2 illustrates the mode mapping of Component *b* of the system introduced in Fig. 1.1. The mode mapping rules of *b* is represented by MMA_b, MMA_d and MMA_e, with all MMAs located in Component *b*. Figures 2.3 and 2.4 show MMA_b and MMA_d [7] which are internally synchronized. More information about MMA can be found in [7].

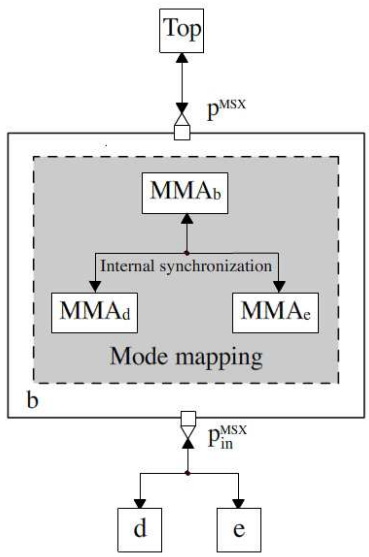


Figure 2.2: Mode mapping and Mode Mapping Automata (MMA)

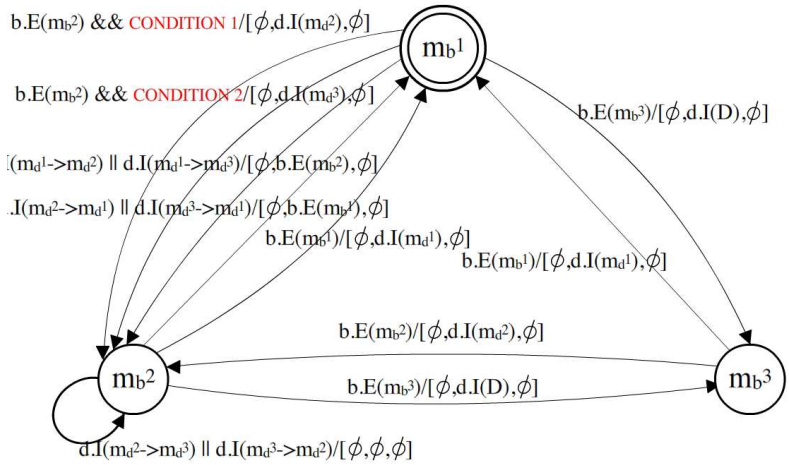


Figure 2.3: The Mode Mapping Automaton of *b*

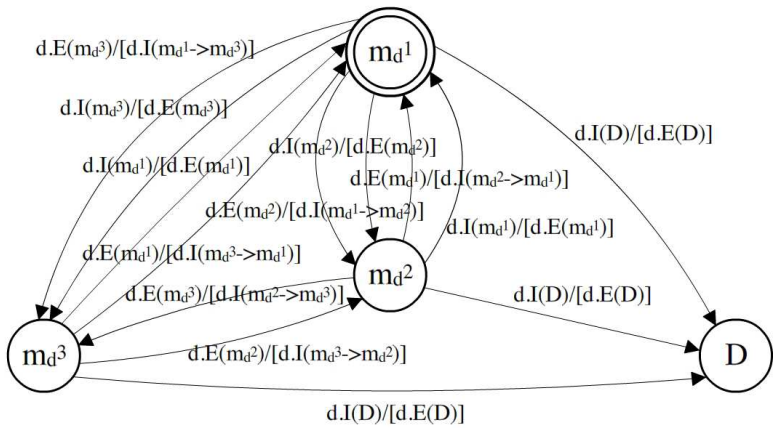


Figure 2.4: The Mode Mapping Automaton of *d*

2.3 The mode switch runtime mechanism

The mode switch runtime mechanism serves as the most important part of MSL. It handles the mode switch of the system and each component at runtime. In this thesis, we focus on the two most fundamental elements of the mode switch runtime mechanism: the MS propagation mechanism and the mode switch dependency rule.

2.3.1 Mode switch propagation

The Mode Switch (MS) propagation mechanism defines two special roles: the Mode Switch Source (MSS) and the Mode Switch Decision Maker (MSDM). An MSS

can actively detect a mode switch event and request to switch mode. It is up to the MSDM, which is usually another component at a higher level, to either approve or reject the request from the MSS. The purpose of the MS propagation mechanism is to propagate the mode switch request from an MSS to all the other components which must switch mode as a consequence.

Two primitives are introduced for the mode switch propagation: Mode Switch Request (MSR) and Mode Switch Instruction (MSI). An MSR is issued by an MSS (say c_i) as a mode switch event is detected and the MSS requests to switch mode. The MSR from the MSS is first propagated to its parent c_j. If the MSR implies no mode switch of c_j, c_j will be the MSDM and directly approve the MSR by issuing an MSI to its subcomponents which must switch mode. If the MSR implies the mode switch of c_j whose current state does not allow such mode switch, c_j will be the MSDM and directly reject the MSR by doing nothing. If the MSR implies the mode switch of c_j whose current state allows such mode switch, c_j will forward the MSR to its parent which will make further decisions. When an MSDM approves an MSR, the MSI from the MSDM will be propagated downstream to all the components which must switch mode. An MSI can never be rejected and it will trigger the mode switch of its recipient.

According to the MS propagation mechanism, the mode switch propagation is divided into two phases: the upstream MSR propagation and the downstream MSI propagation. If the top component happens to be an MSS, the first phase will be skipped as it can directly issue an MSI when it detects a mode switch event. Otherwise, if the MSR from an MSS is rejected by the corresponding MSDM, the second phase will be skipped.

Fig. 2.5 demonstrates the mode switch process of the system in Fig. 1.1, assuming Component *a* is an MSS. When *a* detects a mode switch event, it will issue an MSR to its parent *Top*, which approves the MSR by sending an MSI to its subcomponents *a*, *b*, and *c*. This indicates that the mode switch of *a* also implies the mode switches of *b* and *c*. Component *b* further propagates the MSI to its subcomponent *d*. Component *e* is not affected in this mode switch scenario, thus the MSI is not sent to *e*.

2.3.2 Mode switch dependency rule

The mode switch dependency rule guarantees the mode consistency between different components after each mode switch. It prevents the inconsistent mode problem that some component, which is supposed to run in the new mode after the system mode switch, is still running in the old mode.

After receiving an MSI (and propagating the MSI further if necessary), a component will start its reconfiguration. The mode switch dependency rule requires that a component having received an MSI from its parent must send a primitive Mode Switch Completion (MSC) back after completing its mode switch. The mode switch

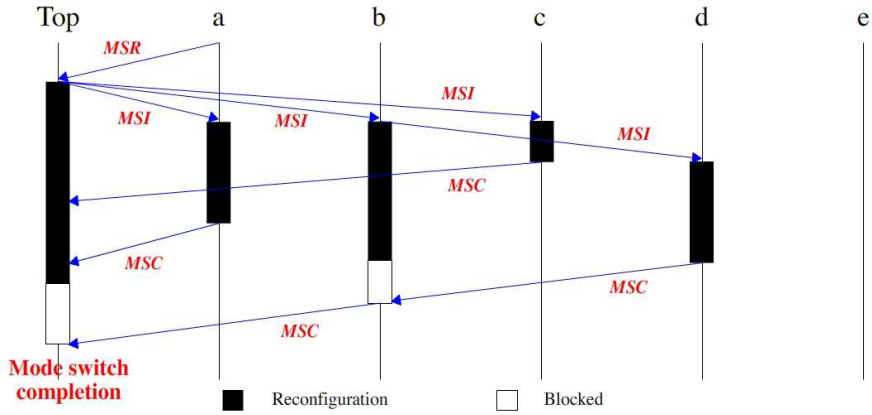


Figure 2.5: The mode switch process

of a composite component is completed only after its reconfiguration and the mode switch completion of all its subcomponents.

The mode switch dependency rule is also demonstrated in Fig. 2.5 where component reconfiguration is represented by black bars. An MSC must be sent in response to the MSI after mode switch. White bars mean that a composite component has completed its reconfiguration but has to wait for the MSC from its subcomponents, which temporarily blocks its mode switch. The system mode switch is completed when the MSDM, *Top*, completes its mode switch.

Chapter 3

The ProCom component model

According to Component-based Development (CBD), a component should comply with a component model. There are currently quite a lot of different component models [8], among which a number of component models are suitable for the development of embedded systems [9], including Rubus [10], Koala [11], AUTOSAR [12] and ProCom [2]. In this thesis, we select the ProCom component model as the target for the MSL implementation. In this chapter, we provide a general introduction of the ProCom component model.

3.1 The ProCom development process

The ProCom development process [13], depicted in Fig. 3.1, is partitioned into the concerns of modelling and synthesis. Both concerns are further partitioned into four stages. The modelling addresses how to get and express deployment related design decisions, for example, how to distribute functionality over the nodes of the system. ProSave and ProSys in Fig. 3.1 are used to model the functional architecture of the system. ProSys is at a higher level than ProSave, as a ProSys component can be composed of ProSave components but not the other way round.

The deployment is performed in two steps. First ProSys subsystems are allocated to virtual nodes with the many-to-one mapping, defined as an intermediate level in the allocation of functional units to the physical nodes of the system. Then virtual nodes are allocated to physical nodes, also with the many-to-one mapping. The advantage of doing this is that virtual nodes preserve real-time properties and can be analyzed independently from the rest of the system. Please note that the four stages of the deployment process can be overlapping rather than being taken in a fixed order.

The synthesis is a process of generating concrete runnable representations of different modelling elements. Different from the deployment process, the synthesis

process must follow a specific order because each step requires the output from the previous step. As is shown in Figure 3.1, the synthesis starts with C files which implement primitive ProSave components and ends with runnable binary images on different physical nodes. ProSys runnables and runnable virtual nodes are intermediate artefacts.

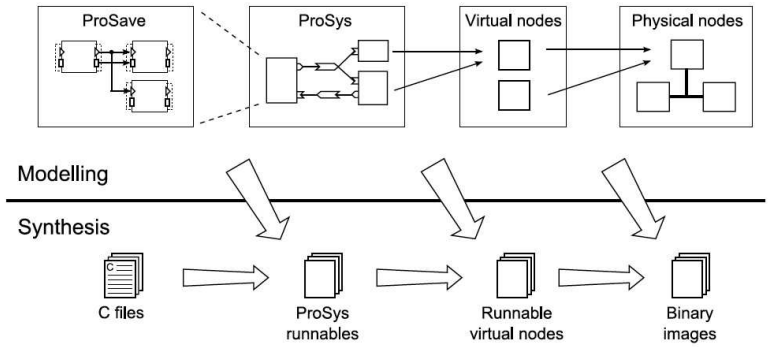


Figure 3.1: The ProCom development process

Fig. 3.2 is the typical system structure developed by ProCom. Hardware is at the bottom level. Above the hardware is the Real-Time Operating System (RTOS). The component layer is built on the RTOS, including ProSys and ProSave components. In the component layer, different applications can be built based on the reuse and composition of ProSys and ProSave components.

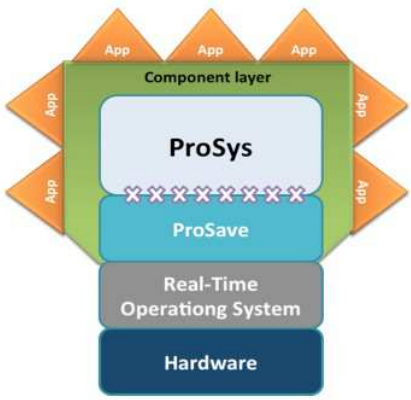


Figure 3.2: The ProCom development process

3.2 The ProCom component model overview

In this section we will provide an overview of the ProCom component model, including its application domains, ProSys and ProSave layers, port types, and connectors. ProCom uses two related but distinct layers to solve the different concerns at different levels of granularity, both layers built by different components whose information is stored in the repository, including requirements, textual documentation and models of the behavior and resource usage [2].

As the upper layer, ProSys allows the hierarchical composition of components as a ProSys component can be constructed from smaller ProSys components (a ProSys component is also called a ProSys subsystem in ProCom). A ProSys component is used to model a subsystem and has input and output message ports as its external interface. Fig. 3.3 depicts the external view of a ProSys subsystem with one input message port and two output message ports. The communication between ProSys components is realized by asynchronous message passing. The handling of a new message is flexible and depends on the receiving ProSys component. A message can be transmitted from one output message port of one ProSys component to one input message port of another ProSys component through a message channel which supports ”many-to-many” communication.

A ProSys component is active since it can have its own thread and no external activation is required to trigger its execution. Compared with ProSave, a ProSys component usually supports more complex functionality.

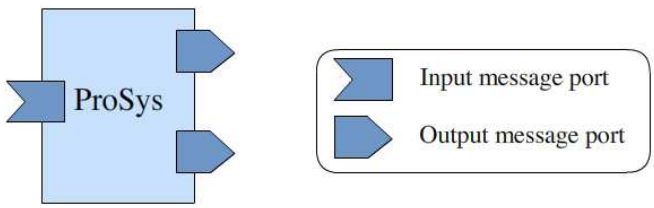


Figure 3.3: A ProSys component

ProSave is the lower layer of ProCom. Similar to ProSys, a composite ProSave component can be composed by smaller ProSave components. A primitive ProSave component is at the bottom level of the hierarchy and can be implemented in C language. Usually the functionality provided by a ProSave component is less complex compared with a ProSys component.

A unique feature of ProSave is that a ProSave component can provide one or more *services*, each of which corresponds to a particular functionality supported by the component. Fig. 3.4 depicts the external view of a ProSave component with two services $S1$ and $S2$. Each service is associated with a single input port group and a set of output port groups. Since control flow and data flow are separated in the

ProSave layer, each port group consists of one trigger port (denoted by dark blue triangles in Fig. 3.4) and a set of data ports (denoted by dark blue squares in Fig. 3.4). For instance, in Fig. 3.4, the service *S*1 has one input port group consisting of one input trigger port and one input data port, and one output port group consisting of one output trigger port and two output data ports. In contrast, *S*2 has one input port group and one output port group, both port groups consisting of one trigger port and one data port. Different from ProSys, a ProSave component is passive and needs external activation to trigger its execution. A ProSave component has very strict execution semantics: for each service, when the input trigger port is activated, the service will become active and the component will read input data from all its input data ports belonging to this service and performs its execution. After that it will produce output at its output data ports of this service and then activate the corresponding output trigger ports in an atomic manner.

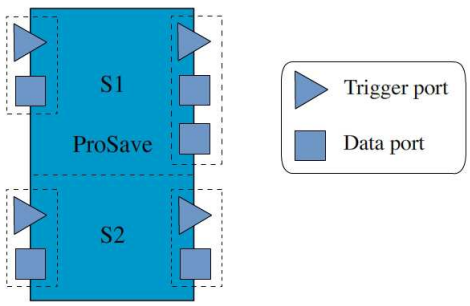


Figure 3.4: A ProSave component

The communicating between ProSave components is of pipe-and-filter style. One output trigger port of a ProSave component can be directly connected to one input trigger port of another ProSave component. Likewise, one output data port of a ProSave component can be directly connected to one input data port of another ProSave component. However, this direct connection can only be one-to-one. More advanced connection for ProSave components is achieved by the use of *connectors*. Fig. 3.5 lists the most commonly used connectors:

- 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 port will be 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 data ports. Just like Control Fork, it can also be presented by a thick dot graphically.
- Selection: It has one input trigger port, at least one input data port and at least two output trigger ports. When its input trigger port is triggered, it will activate exactly one of its output trigger ports according to the data written to its input data port(s).

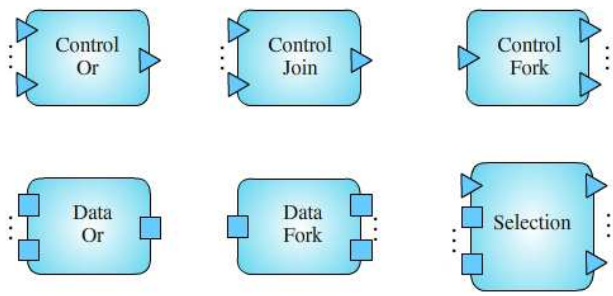


Figure 3.5: Common connectors for the communication between ProSave components

The ProSys and ProSave layers are integrated as a ProSys component that is internally composed by ProSave components (see Fig. 3.6). A unique element within such a special ProSys component is the *Clock* which provides periodical activation for its ProSave subcomponents or its output message ports.

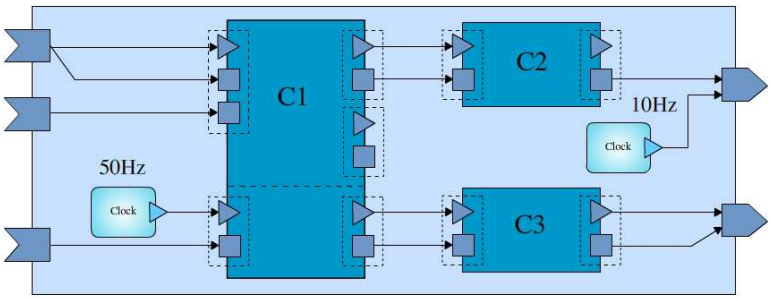


Figure 3.6: A ProSys component composed by ProSave components

Chapter 4

Implementing MSL in ProCom

In this chapter, our theoretical guidance of implementing MSL in the ProCom component model is explained in detail. The main purpose is to integrate MSL into ProCom with a minimum modification of ProCom. As is shown in Fig. 4.1, we realize our goal in three steps:

1. Extending ProCom components into multi-mode components
2. Integrating the mode switch runtime mechanism of MSL into ProCom
3. Merging component connections in different modes

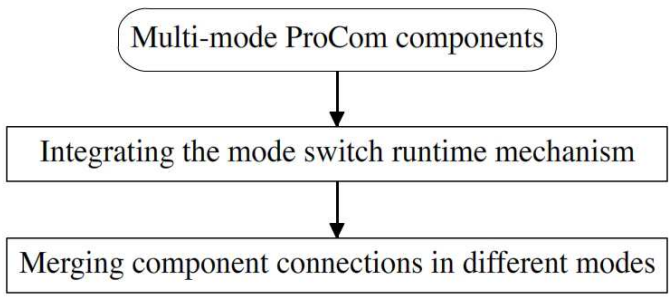


Figure 4.1: The automatic generation flow

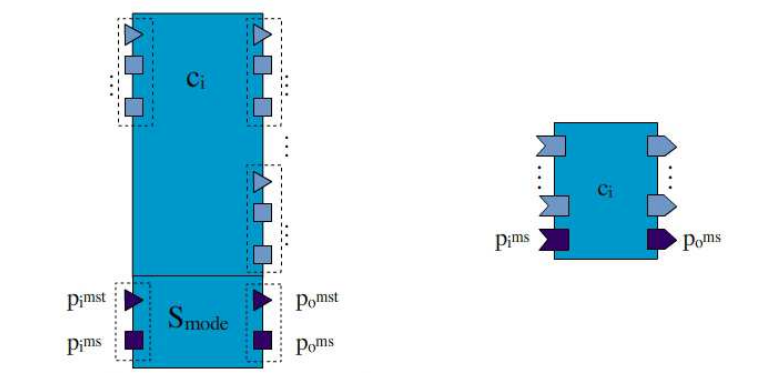
In the following sections, we shall explain these steps in detail.

4.1 Multi-mode ProSave and ProSys components

Currently, the ProCom component model does not support multi-mode and mode switch. Therefore, ProCom must be extended to be mode-aware, preferably with

minimum modification. As one of the contributions in this thesis, both ProSave and ProSys components are extended to support MSL without modification. According to the mode-aware component model of MSL, a multi-mode should have a clear separation between its behavior in each mode and its mode switch handling, and have dedicated mode switch ports to exchange mode switch information with the parent or the subcomponents.

By taking advantage of the support of multiple services of a ProSave component, we use a dedicate service called S_{mode} for the mode switch handling of a ProSave component. The service S_{mode} has one input port group consisting of one input trigger port p_i^{mst} and one input data port p_i^{ms}, and one output port group consisting of one output trigger port p_o^{mst} and one output data port p_o^{ms}. These two port groups function as the dedicated mode switch ports of the ProSave component. Fig. 4.2 (a) shows a typical multi-mode ProSave component c_i consisting of two services. The upper service is dedicated to its regular operation while the lower service S_{mode} is dedicated to the mode switch handling. The service S_{mode} has dedicated mode switch ports marked in purple. It should be noted that a ProSave component can have arbitrary number of regular services. The number of data ports of the input port group of each service can be manually defined. The number of output port groups of each service and the number of output data ports of each output port group can also be manually defined. In contrast, the ports of S_{mode} are fixed. The incoming and outgoing connections of S_{mode} will be introduced in later sections.



(a) A multi-mode ProSave component (b) A multi-mode ProSys component

Figure 4.2: Multi-mode ProSave and ProSys components

A multi-mode ProSys component is a bit different compared to the multi-mode ProSave component presented above. Since a ProSys component has no services and does not distinguish trigger and data ports, there is no need to introduce a dedicated service for handling mode switch. Instead, a dedicated internal thread can be used for handling mode switch and another pair of input and output message ports, p_i^{ms} and p_o^{ms}, can be added as the dedicated mode switch ports. Figure 4.2 (b) shows a

typical multi-mode ProSys component c_i which has a number of input message ports and output message ports, as well as a pair of input and output message ports dedicated to mode switch (marked in purple in the figure). The incoming and outgoing connections of its dedicated mode switch ports will be introduced in later sections.

Moreover, a multi-mode component should have unique configurations in different modes. For a multi-mode ProSave component, its configurations can be defined in its service S_{mode}; for a multi-mode ProSys component, its configurations can be defined in the dedicated internal thread. However, the main focus of component configuration in this thesis is on the running status of a component and component connections, which will be explained in later sections.

Our extension of ProSave and ProSys components can be easily implemented in the development environment of ProCom: PRIDE [5]. When a new component is built in PRIDE, one should specify if a ProSave or ProSys component should be single-mode or multi-mode. If multi-mode is specified, the dedicated mode switch ports and services should be automatically generated.

4.2 Integrating the mode switch runtime mechanism

The second contribution of this thesis is integrating the mode switch runtime mechanism of MSL in ProCom. Guided by MSL, the mode switch of a component is controlled by its mode switch runtime mechanism (and its local mode mapping if it is a composite component). For a primitive ProSave or ProSys component directly implemented by C code, its mode switch runtime mechanism has to be integrated in the code itself, interacting with its dedicated mode switch ports. We here focus more on integrating the mode switch runtime mechanism in a composite ProSave or ProSys component. Since a composite ProCom component has none of its own behavior, but is only a composition of its subcomponents, we propose the use of particular subcomponents of a composite ProSave or ProSys component for the integration of the mode switch runtime mechanism. This section also presents the integration of the mode mapping mechanism of MSL in a composite ProSave or ProSys component.

4.2.1 The MSL components at the ProSave level

For a multi-mode composite ProSave component or a multi-mode composite ProSys component composed by ProSave components, say c_i, we introduce two special subcomponents of c_i for its mode switch handling: $\text{MSL}^A_{c_i}$ and $\text{MSL}^B_{c_i}$, both of which can interact with the S_{mode} service of each $c_j \in SC_{c_i}$ and can be 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 the subcomponents of c_i, excluding $\text{MSL}^A_{c_i}$ and $\text{MSL}^B_{c_i}$. Fig. 4.3

illustrates the ports of $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$. Component $\text{MSL}_{c_i}^A$ has a single service and has an input port group consisting of the following input ports:

- p_i^t: the input trigger port whose activation makes $\text{MSL}_{c_i}^A$ active.
- p_i^{msx}: an input data port for receiving a downstream primitive (i.e. MSI according to the MS propagation mechanism of MSL) during the mode switch propagation.
- p_i^{sync}: an input data port connected to $\text{MSL}_{c_i}^B$ for synchronization.

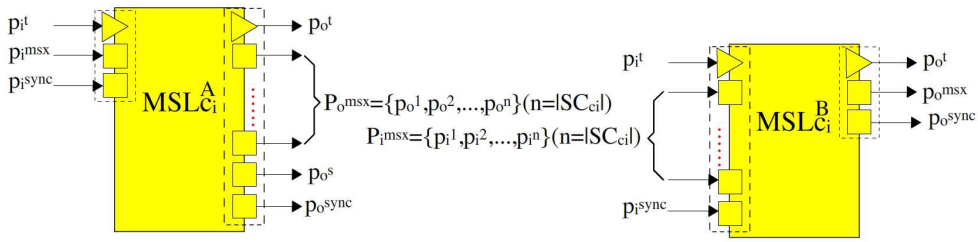


Figure 4.3: The pair of ProSave MSL subcomponents of c_i

Besides, $\text{MSL}_{c_i}^A$ also has an output port group consisting of the following output ports:

- p_o^t: the output trigger port activated after $\text{MSL}_{c_i}^A$ 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 where for each $p_o^k \in P_o^{msx}$ $(k = [1,n])$, p_o^k is connected to $c_j^k.p_i^{ms}$ $(c_j^k \in SC_{c_i})$.
- p_o^s: an output data port indicating the current mode of c_i. It is particularly used for merging component connections defined in separate modes and will be further explained later on.
- p_o^{sync}: an output data port connected to $\text{MSL}_{c_i}^B$ for synchronization.

The ports of $\text{MSL}_{c_i}^B$ is quite symmetrical to $\text{MSL}_{c_i}^A$. If the port p_o^s of $\text{MSL}_{c_i}^A$ is removed and then the input and output ports of $\text{MSL}_{c_i}^A$ are swapped, the resulting port layout will resemble $\text{MSL}_{c_i}^B$, which has an input port group consisting of the following input ports:

- p_i^t: the input trigger port whose activation makes $\text{MSL}_{c_i}^B$ 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 upstream primitives such as MSR or MSC, where for each $p_i^k \in P_i^{msx}$ $(k = [1,n])$, p_i^k is connected to $c_j^k.p_o^{ms}$ $(c_j^k \in SC_{c_i})$.

- p_i^{sync}: an input data port connected to $\text{MSL}_{c_i}^A$ for synchronization.

Besides, $\text{MSL}_{c_i}^B$ has an output port group consisting of the following ports:

- p_o^t: the output trigger port activated after $\text{MSL}_{c_i}^B$ completes its current instance of execution.
- p_o^{msx}: an output data port for forwarding an upstream primitive (i.e. MSR or MSC according to the MS propagation mechanism and the mode switch dependency rule of MSL) during a mode switch.
- p_o^{sync}: an output data port connected to $\text{MSL}_{c_i}^A$ for synchronization.

The component connections between $\text{MSL}_{c_i}^A$, $\text{MSL}_{c_i}^B$ and $c_j^k \in SC_{c_i}$ $(k = [1,n])$ can be demonstrated by Fig. 4.4. The ports of other services of c_i has been omitted for simplicity. The input trigger port $\text{MSL}_{c_i}^A.p_i^t$ is directly connected to $c_i.p_i^{mst}$ and it is eventually connected to a clock residing at the intermediate level between ProSave and ProSys. This clock can periodically trigger $\text{MSL}_{c_i}^A$, $\text{MSL}_{c_i}^B$ and S_{mode} of all ProSave components at all levels. We use such a dedicated clock because the mode switch handling should be separated from other regular services rather than S_{mode}. The input data port $\text{MSL}_{c_i}^A.p_i^{msx}$ is directly connected to $c_i.p_i^{ms}$. The output trigger port $\text{MSL}_{c_i}^A.p_o^{mst}$ is connected to all $c_j^k.p_i^{mst}$ $(c_j^k \in SC_{c_i}, k = [1,n])$. Likewise, for each output data port $\text{MSL}_{c_i}^A.p_o^k$ $(k = [1,n])$, $\text{MSL}_{c_i}^A.p_o^k$ is connected to the corresponding input data port $c_j^k.p_i^{ms}$ $(c_j^k \in SC_{c_i}$.

The input trigger port $\text{MSL}_{c_i}^B.p_i^t$ can be activated by any $c_k \in SC_{c_i}$ $(k = [1,n])$ and this is why a *Control Or* connector is used. For each input data port $\text{MSL}_{c_i}^B.p_i^k$ $(k = [1,n])$, p_i is connected to $c_j^k.p_o^{ms}$ $(c_j^k \in SC_{c_i})$. The output trigger port $\text{MSL}_{c_i}^B.p_o^t$ is directly connected to $c_i.p_o^{mst}$, while its output data port $\text{MSL}_{c_i}^B.p_o^{msx}$ is directly connected to $c_i.p_o^{ms}$. $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$ are connected via their synchronization ports, i.e. p_i^{sync} and p_o^{sync}.

Let P_{c_i} define the parent of a component c_i. Regarding c_i in Fig. 4.4, let's assume c_i receives an MSI from P_{c_i}. The MSI will arrive at its input data port p_i^{msx}. Then according to the MS propagation mechanism, c_i will propagate the MSI to $2^{SC_{c_i}}$ (any possible subset of SC_{c_i}) based on its mode mapping, i.e. by sending an MSI to the components among SC_{c_i} which need to switch mode. For $c_j^k \in SC_{c_i}$ $(k = [1,n])$, if c_j^k needs to switch mode for this mode switch scenario, an MSI will be sent from $\text{MSL}_{c_i}^A.p_o^k$ to $c_j^k.p_i^{ms}$ $(c_j^k \in SC_{c_i})$. If c_j^k is composite, it should have the same internal structure as c_i so that the MSI can be further propagated. Next let's consider an MSR coming from c_j^k $(k = [1,n])$, then the MSS must be within c_j^k. The MSR is sent from $c_j^k.p_o^{ms}$ will arrive at $\text{MSL}_{c_i}^B.p_i^k$ via the *Control Or* connector. If c_i decides to forward the MSR further to P_{c_i}, an MSR will be sent from $\text{MSL}_{c_i}^B.p_o^{msx}$ to $c_i.p_o^{ms}$. Component P_{c_i} should have the same internal structure as c_i, thus the MSR sent from c_i to P_{c_i} can be treated in the same manner as the MSR sent from c_j^k to c_i.

The reason why two MSL components are used to jointly handled the mode switch of a component is that ProCom prohibits the circular connection between two neighboring components. For instance, $\text{MSL}^A_{c_i}.p^t_o$ is connected to $\forall k = [1,n], c^k_j.p^{mst}_i$. This enables the downstream MSI propagation. To enable the upstream MSR propagation or MSC transmission, $c^k_j.p^{mst}_o$ is supposed to be connected to $\text{MSL}^A_{c_i}.p^t_i$. However, this is not allowed because $\text{MSL}^A_{c_i}$ and c^k_j are triggering each other. For this reason, $\text{MSL}^B_{c_i}$ is introduced to avoid mutual triggering. Since $\text{MSL}^A_{c_i}$ and $\text{MSL}^B_{c_i}$ are supposed to share the same mode information, they must be synchronized frequently and that is why they both have synchronization ports.

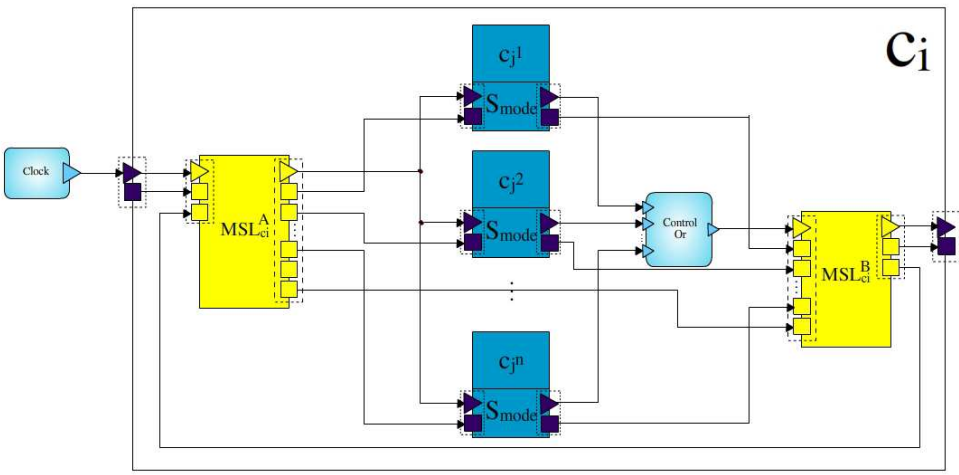


Figure 4.4: $\text{MSL}^A_{c_i}$, $\text{MSL}^B_{c_i}$ and component connections

Both $\text{MSL}^A_{c_i}$ and $\text{MSL}^B_{c_i}$ are primitive ProSave components and the mode switch runtime mechanism of MSL can be implemented by them. Algorithms 1 and 2 describe the mode switch handling of $\text{MSL}^A_{c_i}$ and $\text{MSL}^B_{c_i}$ respectively, where a few notations are explained as follows:

- *switching*: A boolean variable set to true when c_i is switching mode.
- *Mode_Mapping*: A function implementing the mode mapping of c_i based on the mode mapping mechanism of MSL.
- *Reconfiguration*: A function reconfiguring c_i to its configuration in the new mode. However, if the new mode of c_i is the same as its current mode, no reconfiguration will be taken.
- *DummyData*: A ProSave component must provide data at all its output data ports when the output trigger port in the same service is activated. Dummy

data can be written to those output data ports where no data is expected to be sent out.

- $Conf(c_i)$: The configuration of c_i in its current mode.
- *MSC_all*: A boolean variable set to true when $\text{MSL}_{c_i}^{B}$ has completed its MSC collection from SC_{c_i}.
- *UpdateConf*($Conf(c_i)$): A function updating the old configuration of c_i to $Conf(c_i)$. Compared with the function *Reconfiguration*, this function produces the same result, though in a different way (the new configuration is provided from the input synchronization port for this function, while the new configuration has to be additionally derived in the function *Reconfiguration*).

Please notice that these algorithms have abstracted many details for the sake of simplicity. For instance, $P_o^{msx} := MSI$ means that c_i sends an MSI to $2^{SC_{c_i}}$ based on its mode mapping. Hence, it implies that the MSI may not be sent to $\forall p_o^k \in P_o^{msx}$.

The two algorithms presented here implicitly require that a primitive multi-mode ProSave component should send dummy data via its port p_o^{ms} if it receives dummy data from its port p_i^{ms} while being triggered. Next let's demonstrate how the mode switch runtime mechanism works within c_i by a typical mode switch scenario. Suppose an MSR is issued from c_j^1 and c_i approves the MSR by issuing an MSI that is propagated downstream. Taking algorithms 1 and 2 and Fig. 4.4 into account, we get the following procedures:

1. An MSR is sent from $c_j^1.p_o^{ms}$ to $\text{MSL}_{c_i}^{B}.p_i^1$. Meanwhile, $\text{MSL}_{c_i}^{B}.p_i^t$ is activated by c_j^1 and $\text{MSL}_{c_i}^{B}$ will do its mode mapping and approves the MSR according to its mode mapping result and current state. Then $\text{MSL}_{c_i}^{B}$ will change its current mode to *switching* and send an MSI to $\text{MSL}_{c_i}^{A}$ via $\text{MSL}_{c_i}^{B}.p_o^{sync}$.
2. $\text{MSL}_{c_i}^{A}$ receives the MSI from its port p_i^{sync} and checks its mode mapping which should be equal to $\text{MSL}_{c_i}^{B}$. After that, $\text{MSL}_{c_i}^{A}$ will change its mode to *switching* and start its reconfiguration. After its reconfiguration, it will send an MSI to its port P_o^{msx} based on the mode mapping result and also send its new configuration to $\text{MSL}_{c_i}^{B}$ via its port p_o^{sync}.
3. For $c_j^k \in SC_{c_i}$ $(k = [1,n])$ which receives the MSI from $\text{MSL}_{c_i}^{A}$, if c_j^k is primitive, it can immediately start its reconfiguration and then send an MSC to $\text{MSL}_{c_i}^{B}$. If c_j^k is composite, its internal behavior will be the same as c_i. Eventually, c_j^k will send an MSC to $\text{MSL}_{c_i}^{B}$ which collects from MSC from SC_{c_i}. When $\text{MSL}_{c_i}^{B}$ completes its MSC collection, it will read the data from its port p_i^{sync} which must be the new configuration already sent from $\text{MSL}_{c_i}^{A}$. Thus $\text{MSL}_{c_i}^{B}$ will update its configuration, update its current mode from *switching* to the

Algorithm 1 $MSL_{c_i}^A$

loop
  if p_i^t **then**
    if $(p_i^{msx} = MSI \vee p_i^{sync} = MSI) \wedge m_{c_i} \neq switching$ **then**
      $Mode\_Mapping$;
      $m_{c_i} := switching$;
      $Reconfiguration$;
      $P_o^{msx} := MSI$;
      $p_o^s := DummyData$;
      $p_o^{sync} := Conf(c_i)$;
    end if
    if $p_i^{sync} = MSC\_all \wedge m_{c_i} = switching$ **then**
      $m_{c_i} := m_{c_i}^{new}$;
      $P_o^{msx} := DummyData$;
      $p_o^s := m_{c_i}^{new}$;
      $p_o^{sync} := DummyData$;
    end if
    if $p_i^{msx} = DummyData \wedge p_i^{sync} = DummyData$ **then**
      $P_o^{msx} := DummyData$;
      $p_o^s := m_{c_i}$;
      $p_o^{sync} := DummyData$;
    end if
    $p_i^t := false$;
    $p_o^t := true$;
  end if
end loop

Algorithm 2 $MSL_{c_i}^{B}$

loop
  if p_i^t **then**
    if $p_i^k = MSR \wedge m_{c_i} \neq switching\ (p_i^k \in P_i^{msx}, k = [1,n])$ **then**
      $Mode\_Mapping$;
      if *Approve* **then**
        $m_{c_i} := switching$;
        $p_o^{msx} := DummyData$;
        $p_o^{sync} := MSI$;
      else if *Reject* **then**
        $p_o^{msx} := DummyData$;
        $p_o^{sync} := DummyData$;
      else
        $p_o^{msx} := MSR$;
        $p_o^{sync} := DummyData$;
      end if
    end if
    if $p_i^k = MSC \wedge m_{c_i} = switching (p_i^k \in P_i^{msx}, k = [1,n])$ **then**
      if MSC_all **then**
        $UpdateConf(Conf(c_i))$;
        $m_{c_i} := m_{c_i}^{new}$;
        $p_o^{msx} := MSC$;
        $p_o^{sync} := MSC\_all$;
      else
        $p_o^{msx} := DummyData$;
        $p_o^{sync} := DummyData$;
      end if
    end if
    if $P_i^{msx} = DummyData \wedge p_i^{sync} = DummyData$ **then**
      $p_o^{msx} := DummyData$;
      $p_o^{sync} := DummyData$;
    end if
    $p_i^t := false$;
    $p_o^t := true$;
  end if
end loop

new mode, sending an MSC to P_{c_i} via its port p_o^{msx} and sending *MSC_all* to $\text{MSL}_{c_i}^A$ via its port p_o^{sync}.

4. $\text{MSL}_{c_i}^A$ receives *MSC_all* from its port p_i^{sync} and then changes its current mode to the new mode. Besides, $\text{MSL}_{c_i}^A.p_o^s$ is updated to the new mode as well. The mode switch of c_i is completed.

Due to the rigorous execution semantics of ProSave components, the mode switch scenario above requires at least four cycles of the clock in Fig. 4.4. No parallel reconfiguration among ProSave components are allowed as component reconfiguration must be taken one after another by following their triggering order. Algorithms 1 and 2 do not consider the case when c_i is the top component or an MSS. If c_i is the top component, it will never forward an upstream MSR upwards and we only need to remove the corresponding session of Algorithm 2. If c_i is an MSS but not the top component, it can actively issue an MSR to P_{c_i} when it detects a mode switch event and this can be performed by $\text{MSL}_{c_i}^B$. If c_i is an MSS and also the top component, it can actively issue an MSI that is propagated downstream when it detects a mode switch event. The MSI can be handled in the same way as the MSI when c_i receives from P_{c_i}.

4.2.2 The MSL component at the ProSys level

The mode switch runtime mechanism of MSL is implemented at the ProSys level in a similar way as ProSave level. For a multi-mode composite ProSys component c_i, we introduce a special subcomponent of c_i for its mode switching handling: MSL_{c_i} which plays an equal role as the pair of $\text{MSL}_{c_i}^A$ and $\text{MSL}_{c_i}^B$ described in the last subsection. The major difference includes:

- Message passing between ProSys components is more flexible than the pipe-and-filter communication pattern at the ProSave level. Circular connection is allowed, i.e. two neighboring components can send messages to each other via different ports, therefore, one MSL subcomponent of c_i is sufficient to handle its mode switch.
- A ProSys component does not distinguish control and data flow.
- Since a message channel allows many-to-one communication, the *Control Or* connector at the ProSave level can be removed.

Still, let $SC_{c_i} = \{c_j^1, c_j^2, \cdots, c_j^n\}$ $(n \in \mathbb{N})$ denote the set of the subcomponents of c_i, excluding MSL_{c_i}. Fig. 4.5 illustrates the ports of MSL_{c_i}, which has the following ports:

- p_i^{msx}: an input message port for receiving an MSI.

- $P_i = \{p_i^1, p_i^2, \cdots, p_i^n\}$ $(n = |SC_{c_i}|)$: a set of input message ports for receiving an MSR or MSC, where for each $p_i^k \in P_i^{msx}$ $(k = [1,n])$, p_i^k is connected to $c_j^k.p_o^{ms}$ $(c_j^k \in SC_{c_i})$.
- p_o^s: an output message port indicating the current mode of c_i. It will be further explained later on.
- $P_o = \{p_o^1, p_o^2, \cdots, p_o^n\}$ $(n = |SC_{c_i}|)$: a set of output message ports for sending an MSI to $2^{SC_{c_i}}$, where for each $p_o^k \in P_o$ $(k = [1,n])$, p_o^k is connected to $c_j^k.p_i^{ms}$ $(c_j^k \in SC_{c_i})$.
- p_o^{msx}: an output message port for forwarding an MSR or sending an MSC to P_{c_i}.

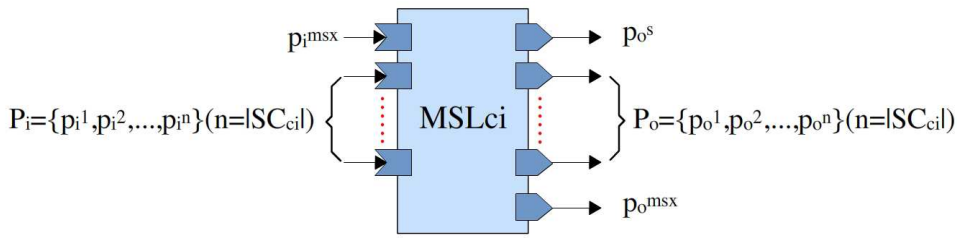


Figure 4.5: The ProSys MSL subcomponent of c_i

The component connections between MSL_{c_i} and $c_j^k \in SC_{c_i}$ $(k = [1,n])$ can be demonstrated by Fig. 4.6. The input message port $\text{MSL}_{c_i}.p_i^{msx}$ is directed connected to $c_i.p_i^{ms}$ for receiving an MSI. The input message ports $p_i^1, p_i^2, \cdots, p_i^n$ are connected to the p_o^{ms} of the corresponding subcomponents of c_i: $c_j^1, c_j^2, \cdots, c_j^n$ for receiving an MSR or MSC. The output message ports $p_o^1, p_o^2, \cdots, p_o^n$ are connected to p_i^{ms} of $c_j^1, c_j^2, \cdots, c_j^n$ for sending an MSI. In addition, The output message port $\text{MSL}_{c_i}.p_o^{msx}$ is directly connected to $c_i.p_o^{ms}$ for sending an MSR or MSC to P_{c_i}.

MSL_{c_i} is a primitive ProSys component where the mode switch runtime mechanism of c_i is implemented. The mode switch handling of MSL_{c_i} is described in Algorithm 3. A mode switch scenario can be used for demonstration. Still, let's assume that an MSR is issued from c_j^1 and c_i approves the MSR by issuing an MSI that is propagated downstream. Taking Algorithm 3 and Fig. 4.6 into account, we get the following procedures:

1. An MSR is sent from $c_j^1.p_o^{ms}$ to $\text{MSL}_{c_i}.p_i^1$.
2. MSL_{c_i} refers to its mode mapping, changes its mode to *switching* and then propagates an MSI to $2^{SC_{c_i}}$ based on the mode mapping result.

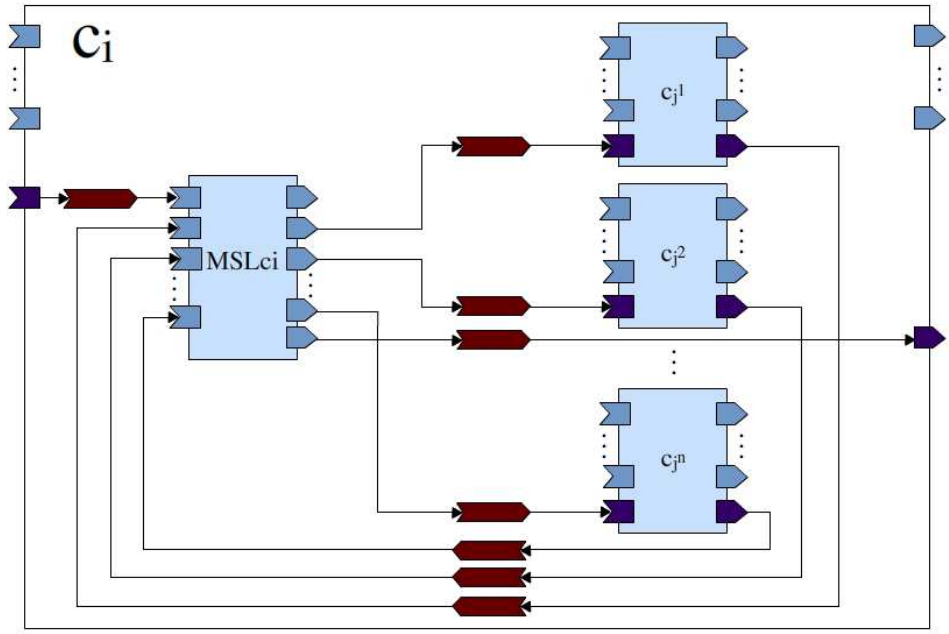


Figure 4.6: MSL_{c_i} and component connections

3. For $c_j^k \in SC_{c_i}$ $(k = [1,n])$ which receives the MSI from MSL_{c_i}, if c_j^k is primitive, it can immediately start its reconfiguration and then send an MSC to MSL_{c_i} via its port p_o^{ms}. If c_j^k is composite, its internal behavior will be the same as c_i. Eventually, c_j^k will send an MSC to MSL_{c_i} which collects from MSC from SC_{c_i}. When MSL_{c_i} completes its MSC collection, it will reconfigure itself if its new mode is different from its current mode. It will also change its current mode and its output data at port p_o^s to the new mode $m_{c_i}^{new}$. Finally, an MSC is sent to P_{c_i} via its port p_o^{msx}.

Since MSL_{c_i} requires no additional clock to trigger its execution and there is no synchronization problem between $\mathrm{MSL}_{c_i}^A$ and $\mathrm{MSL}_{c_i}^B$, the mode switch handling at the ProSys level is much easier than at the ProSave level.

4.2.3 Mode mapping

Algorithms 1-3 all include the function *Mode_Mapping*, which implements the mode mapping mechanism of MSL. The approaches proposed in this thesis allow the pair of ProSave component $\mathrm{MSL}_{c_i}^A$ and $\mathrm{MSL}_{c_i}^B$ and the ProSys component MSL_{c_i} to be automatically generated for a specific given component hierarchy. However, the mode mapping of c_i has to be manually specified to meet the expectation of the designer

Algorithm 3 MSL_{c_i}

loop
  if $p_i^{msx} = MSI \wedge m_{c_i} \neq switching$ **then**
    Mode_Mapping;
    $m_{c_i} := switching$;
    $P_o := MSI$;
  end if
  if $p_i^k = MSR \wedge m_{c_i} \neq switching$ $(p_i^k \in P_{SC}, k = [1,n])$ **then**
    Mode_Mapping;
    if *Approve* **then**
      $m_{c_i} := switching$;
      $P_o := MSI$;
    else if *Reject* **then**
      return ;
    else
      $p_o^{msx} := MSR$;
    end if
  end if
  if $p_i^k = MSC \wedge m_{c_i} = switching (p_i^k \in P_{SC}, k = [1,n])$ **then**
    if *MSC_all* **then**
      Reconfiguration;
      $m_{c_i} := m_{c_i}^{new}$;
      $p_o^s := m_{c_i}^{new}$;
      $p_o^{msx} := MSC$;
    end if
  end if
end loop

and customer. This section presents a *Mode Mapping Wizard* for guiding the mode mapping specification in ProCom. For each composite component c_i, the *Mode Mapping Wizard* consists of four steps: (1) Defining the mode mapping table of c_i; (2) Generating the skeleton of the Mode Mapping Automata (MMAs) of c_i and SC_{c_i}; (3) Editing the MMAs; and (4) MSS and mode switch scenario specification. Since the *Mode Mapping Wizard* can automatically generate the basic parts of MMAs in a graphical manner, the mode mapping specification of c_i can be much more convenient than using a particular specification language from scratch. Next let's briefly explain each step of the *Mode Mapping Wizard*.

Step 1: Mode Mapping Wizard
For a composite component c_i, the basic mode mapping between c_i and SC_{c_i} can be presented by a *mode mapping table*. The *Mode Mapping Wizard* is able to generate a blank *mode mapping table* shown in Table 4.1. The first column of Table 4.1 lists c_i and SC_{c_i}. Besides, Table 4.1 has K additional columns, where $n = \max\{|M_{c_i}|, |M_{c_j^k}|\}$ ($c_j^k \in SC_{c_i}$,$k = [1,n]$, n is the number of subcomponents of c_i, and M_{c_i} denotes the set of supported modes of c_i). Among these K columns, each cell should be filled with one supported mode of the component in the same row. This can be either manually input or selected from a mode selection list. For c_i, the mode selection list includes all its pre-defined supported modes, while for c_j^k, the mode selection list includes all its pre-defined supported modes plus "Deactivated", as c_i may deactivate certain subcomponents in certain modes. One can also merge cells horizontally to specify more complex mode mapping.

Component	Supported modes			
c_i			$\cdots$	
$c_j^1 \in SC_{c_i}$			$\cdots$	
$c_j^2 \in SC_{c_i}$			$\cdots$	
$\vdots$	$\cdots$			
$c_j^n \in SC_{c_i}$			$\cdots$	

Table 4.1: The mode mapping table of c_i

Step 2: Automatic generation of MMAs
As is introduced in Section 2.2, an MMA [7] consists of locations and transitions between locations. The locations, including the deactivated states, of each MMA can be automatically generated based on the *mode mapping table*. Transitions of each MMA can also be automatically generated by considering all possible mode switch scenarios. However, the definition of Dominant Default Modes (DDMs) is beyond the expression of *mode mapping table*. Therefore, some transition labels of an MMA may not be complete.

Step 3: Editing MMAs
The main task in this step is to complete the undefined transition labels generated in Step 2. This is realized by defining the DDMs for each component for all possible mode switch scenarios. The locations and transitions of an MMA can also be manually edited, added or deleted. Any operation violating the semantics of MMA will cause a syntax error that warns the designer so as to avoid incorrect mode mapping.

Step 4: MSS and mode switch scenario specification
In Step 2 and Step 3, all possible mode switch scenarios are considered for better component reuse. However, for a particular system, only some mode switch scenarios could happen. All Mode Switch Sources (MSSs) in the system should be defined and a mode switch scenario can be defined as: an MSS c_k requests to switch from $m^i_{c_k}$ to $m^j_{c_k}$ ($m^i_{c_k}, m^j_{c_k} \in M_{c_k}$).

The four steps above completes the mode mapping specification of c_i.

4.3 Multi-mode component connections

As is indicated in Fig. 1.1 at the very beginning of this thesis, the inner component connection of a composite component c_i can be different while c_i is in different modes. Of course it is quite easy to define the inner component connection of c_i separately for each mode of c_i, however, merging component connections in different modes becomes a tricky problem in ProCom. As the third contribution of this thesis, we provide a solution which can automatically generate the complete component connection based on component connections separately defined in different modes with minimum extension of ProCom. Depending on the current mode of a component, only activated components and active component connections are selected. Next our solution will be presented at the ProSave level and the ProSys level, respectively.

4.3.1 Merging component connection at the ProSave level

Consider a multi-mode ProSave component or a multi-mode ProSys component composed by ProSave components, say c_i, whose inner component connection is mode-dependent. The basic idea of automatically generating the merged component connection within c_i is to packaging each $c^k_j \in SC_{c_i}$ ($k = [1,n], n = |SC_{c_i}|$) with additional connectors. A connector is attached to each port of c^k_j except its dedicated mode switch ports of the service S_{mode}, i.e. p^{mst}_i, p^{ms}_i, p^{mst}_o and p^{ms}_o. General speaking, a *Control Or* connector is attached to an input trigger port; a *Data Or* connector is attached to an input data port; a *Selection* connector is attached to an output trigger port; a *Data Selection* connector is attached to an output data port. The *Data Selection* connector does not exist in the current ProCom component model, however, it can be easily developed as its execution semantics is very similar to *Selection*. A

Data Selection connector has two input data ports: p_i^d and p_i^s and at least two output data ports. Based on the value of the data at port p_i^s, this connector forwards the data from p_i^d to exactly one of its output data ports. The number of the input ports of *Control Or* and *Data Or*, and the number of the output ports of *Selection* and *Data selection* depend on the number of modes supported by the parent component c_i.

Let c_i be a composite multi-mode ProSave component with the set of supported modes $M_{c_i} = \{m_{c_i}^1, m_{c_i}^2, \cdots, m_{c_i}^q\} (q \geq 1)$. For each mode $m_{c_i}^k$ $(k = [1,q])$, the inner component connection of c_i has been provided separately. Component c_i has a number of subcomponents $c_j^1, c_j^2, \cdots, c_j^n$ $(n = |SC_{c_i}|)$. For each $c_j^k \in SC_{c_i}$ $(k = [1,n])$,

- Let P_i^t be the set of input trigger ports of c_j^k of all its services except S_{mode}.
- Let P_i^d be the set of input data ports of c_j^k of all its services except S_{mode}.
- Let P_o^t be the set of output trigger ports of c_j^k of all its services except S_{mode}.
- Let P_i^d be the set of output data ports of c_j^k of all its services except S_{mode}.

Let $\overline{p_i^t} \in P_i^t$ denote an arbitrary port belonging to P_i^t and the same is true of $\overline{p_i^d} \in P_i^d$, $\overline{p_o^t} \in P_o^t$ and $\overline{p_o^d} \in P_o^d$. Connectors are automatically generated around c_j^k based on the following rules:

- For each $\overline{p_i^t} \in P_i^t$ of c_j^k, a *Control Or* connector A is generated, with a set of input trigger ports $P_i^t = \{p_i^{t1}, p_i^{t2}, \cdots, p_i^{tq}\} (q = |M_{c_i}|)$ and one output trigger port p_o^t. The incoming connection to $A.p_i^{tl}$ $(l = [1,q])$ follows the pre-defined connection while c_i is in mode $m_{c_i}^l$. The output trigger port $A.p_o^t$ is directly connected to $c_j^k.\overline{p_i^t}$.
- For each $\overline{p_i^d} \in P_i^d$ of c_j^k, a *Data Or* connector B is generated, with a set of input data ports $P_i^d = \{p_i^{d1}, p_i^{d2}, \cdots, p_i^{dq}\} (q = |M_{c_i}|)$ and one output trigger port p_o^d. The incoming connection to $B.p_i^{dl}$ $(l = [1,q])$ follows the pre-defined connection while c_i is in mode $m_{c_i}^l$. The output data port $B.p_o^t$ is directly connected to $c_j^k.\overline{p_i^d}$.
- For each $\overline{p_o^t} \in P_o^t$ of c_j^k, a *Selection* connector C is generated, with one input trigger port p_i^t, one input data port p_i^s and a set of output trigger ports $P_o^t = \{p_o^{t1}, p_o^{t2}, \cdots, p_o^{tq}\} (q = |M_{c_i}|)$. The input trigger port $C.p_i^t$ is directly connected to $c_j^k.\overline{p_o^t}$. The input data port $C.p_i^s$ is connected to $\text{MSL}_{c_i}^A.p_o^s$ (see the previous section). The outgoing connection from $C.p_o^{tl}$ $(l = [1,q])$ follows the pre-defined connection while c_i is in mode $m_{c_i}^l$ according to the value of the data at $C.p_i^s$: If the data read from $C.p_i^s$ returns $m_{c_i}^l$ $(l = [1,q])$, $C.p_o^{tl}$ will be triggered.

- For each $\overline{p_o^d} \in P_o^d$ of c_j^k, a *Data Selection* connector D is generated, with one input data port p_i^d, and the other input data port p_i^s and a set of output data ports $P_o^d = \{p_o^{d1}, p_o^{d2}, \cdots, p_o^{dq}\}(q = |M_{c_i}|)$. The input data port $D.p_i^d$ is directly connected to $c_j^k.\overline{p_o^d}$. The input data port $D.p_i^s$ is connected to $\text{MSL}_{c_i}^A.p_o^s$ (see the previous section). The outgoing connection from $D.p_o^{dl}$ $(l = [1,q])$ follows the pre-defined connection while c_i is in mode $m_{c_i}^l$ according to the value of the data at $D.p_i^s$: If the data read from $D.p_i^s$ returns $m_{c_i}^l$ $(l = [1,q])$, the data read from $D.p_i^d$ will be forwarded exactly to $D.p_o^{dl}$.

The above presented rules are illustrated in Fig. 4.7 and should be applied to all subcomponents of c_i. Moreover, the input and output ports of c_i itself deserve special care. Let's reuse the definition P_i^t, P_i^d, P_o^t and P_o^d of c_j^k for c_i, then,

- For each $\overline{p_i^t} \in P_i^t$ of c_i, a *Selection* connector (the same as C defined above) is generated and connected to it within c_i, considering $c_i.\overline{p_i^t}$ as an output trigger port (not belonging to the service S_{mode}) of a subcomponent of c_i.
- For each $\overline{p_i^d} \in P_i^d$ of c_i, a *Data Selection* connector (the same as D defined above) is generated and connected to it within c_i, considering $c_i.\overline{p_i^d}$ as an output data port (not belonging to the service S_{mode}) of a subcomponent of c_i.
- For each $\overline{p_o^t} \in P_o^t$ of c_i, a *Control Or* connector (the same as A defined above) is generated and connected to it within c_i, considering $c_i.\overline{p_o^t}$ as an input trigger port (not belonging to the service S_{mode}) of a subcomponent of c_i.
- For each $\overline{p_o^d} \in P_o^d$ of c_i, a *Data Or* connector (the same as B defined above) is generated and connected to it within c_i, considering $c_i.\overline{p_o^d}$ as an input data port (not belonging to the service S_{mode}) of a subcomponent of c_i.

It is also important to note that some component connections may remain unchanged while c_i is switching mode, thus the merged component connections within c_i can be optimized by removing redundant generated connectors and redundant connections. In practice, the merged component connection may still look rather complex even after optimization, however, they are automatically generated by following simple rules. Therefore, the visual complexity will not be a problem. A desired function in PRIDE with MSL support would be to allow the user to view the component connection in a particular mode while hiding the component connections in other modes.

4.3.2 Merging component connection at the ProSys level

Like at the ProSave level, component connections in different modes can also be merged and automatically generated. The central idea is similar to that at the ProSave

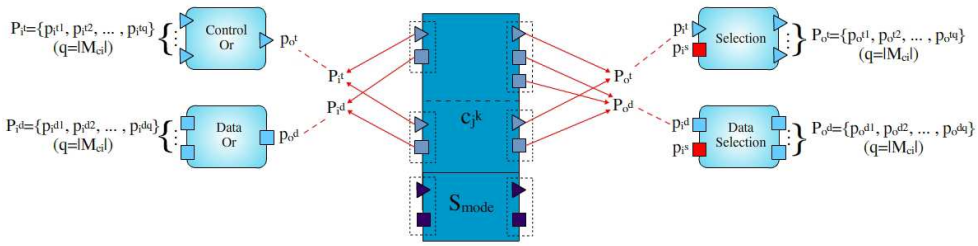


Figure 4.7: Merging component connections at the ProSave level

level. Since no connectors are supported at the ProSys level, we do not need to generate any connectors. Instead, we can generate primitive ProSys components functioning as the four types of connectors generated at the ProSave level. Since an input message port can receive messages from multiple message channels, there is no need to generate ProSys components with the same function as connectors *Control Or* or *Data Or*. Actually, we only need to generate a primitive ProSys component *Selection* that plays the same role as the combination of *Selection* and *Data Selection* at the ProSave level.

Let c_i be a composite multi-mode ProSys component composed by ProSys components. Component c_i has the set of supported modes $M_{c_i} = \{m^1_{c_i}, m^2_{c_i}, \cdots, m^q_{c_i}\}(q \geq 1)$. For each mode $m^k_{c_i}$ $(k = [1,q])$, the inner component connection of c_i has been provided separately. Component c_i has a number of subcomponents $c^1_j, c^2_j, \cdots, c^n_j$ $(n = |SC_{c_i}|)$. For each $c^k_j \in SC_{c_i}$ $(k = [1,n])$, let P_i be the set of input message ports of c^k_j except $c^k_j.p^{ms}_i$, and let P_o be the set of output message ports of c^k_j except $c^k_j.p^{ms}_o$. Then a primitive ProSys component called *Selection* and denoted as S is generated. Component E has two input message ports, p_i and p_s, and a set of output message ports $P_o = \{p^1_o, p^2_o, \cdots, p^q_o\}(q = |SC_{c_i}|)$. For each $\overline{p_o} \in P_o$ of c^k_j, $c^k_j.\overline{p_o}$ is connected to $E.p_i$. The input message port $E.p_s$ is connected to $\mathrm{MSL}_{c_i}.p^s_o$. 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 value of the data at $E.p_s$: If the data read from $E.p^s_i$ returns $m^l_{c_i}$ $(l = [1,q])$, the data read from $E.p_i$ will be forwarded to $E.p^l_o$.

The connection between c^k_j and E can be illustrated in Fig. 4.8. Moreover, let $c_i.P_i$ denote the set of input message ports of c_i except $c_i.p^{ms}_i$. Then for each $\overline{p_i} \in P_i$ of c_i, a primitive ProSys *Selection* component (the same as E defined above) is generated and connected to it within c_i, considering $c_i.\overline{p_i}$ as an output message port (excluding the port p^{ms}_o) of a subcomponent of c_i. The merged component connection at the ProSys level can also be optimized in the same way as at the ProSave level.

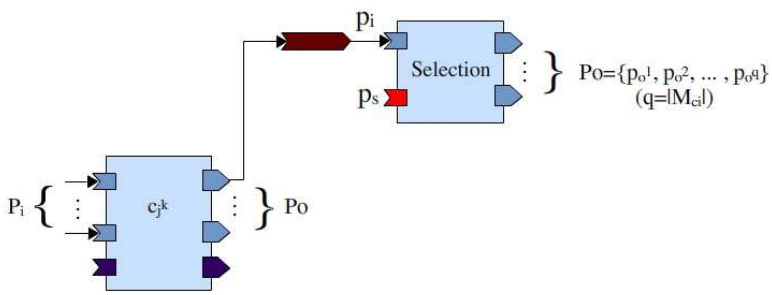


Figure 4.8: Merging component connections at the ProSys level

Chapter 5

A pedagogical example

This chapter demonstrates our approaches of implementing MSL in ProCom by a conceptual example for pedagogical purpose. First, the general system description of this example is provided. Then we show how the system can be developed in ProCom guided by MSL, including multi-mode ProSys and ProSave components, MSL components implementing the mode switch runtime mechanism, and merging component connections.

5.1 System description

The system discussed in this chapter has the same component hierarchy as the system in Fig. 1.1, introduced at the beginning of this thesis. The system, i.e. Component *Top*, consists of components *a*, *b*, and *c*. And Component *b* is further composed by *d* and *e*. However, the supported modes of different components and their connections are different from the system in Fig. 1.1.

The supported modes of each component and the basic mode mapping at each level are presented in tables 5.1 and 5.2. It has been stated that a more powerful expression of the mode mapping of *Top* and *b* is using Mode Mapping Automata (MMA). However, specifying the MMA is beyond the scope of this thesis and more details can be found in [7].

Fig. 5.1 shows the component connections of the system based on tables 5.1 and 5.2. Black and grey colors are used to represent different mode-specific behaviors. For instance, *a* has two mode-specific behaviors while *d* has three mode-specific behaviors, represented by white, black and grey, respectively in Fig. 5.1. Besides, *c* and *d* can be deactivated when their parents are in certain modes.

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 5.1: The mode mapping table 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 5.2: The mode mapping table of *b*

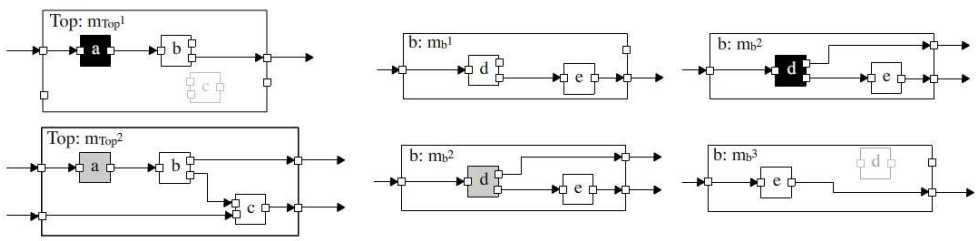


Figure 5.1: Component connections at all levels

5.2 Developing the system in ProCom

Now let's design the system introduced in the previous section in ProCom, where MSL has been implemented. In order to cover both ProSys and ProSave layers, we define *Top*, *a*, *b* and *c* as ProSys components, and define *d* and *e* as ProSave components. According to Section 4.1, the first step is to generate multi-mode components at both ProSys and ProSave levels based on the system specification.

Fig. 5.2 displays the ProCom component hierarchy of the system. All components have been developed as multi-mode components. Compared with the single-mode version, a multi-mode component has additional dedicated mode switch ports marked in purple in Fig. 5.2. Among the multi-mode ProSys components, including *Top*, *a*, *b* and *c*, each of them has an input message port p^{ms}_i and an output message port p^{ms}_o dedicated to mode switch. Among the multi-mode ProSave components, including *d* and *e*, each of them has a dedicated mode switch service S_{mode}, which has four dedicated mode switch ports: p^{mst}_i, p^{ms}_i, p^{mst}_o and p^{ms}_o. All components conform to the definition illustrated in Fig. 4.2.

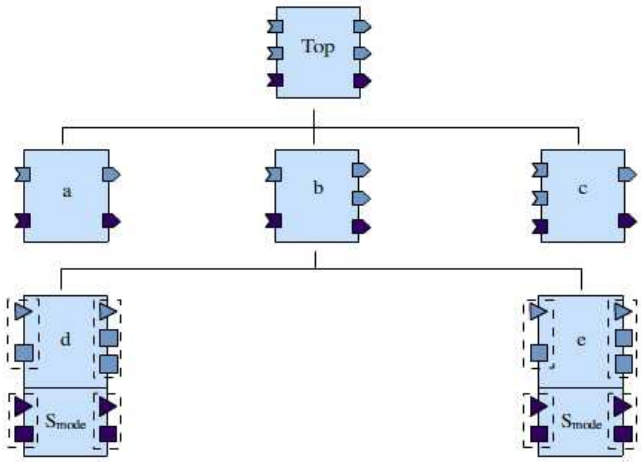


Figure 5.2: The ProCom component hierarchy of the system

In addition, referring to the inner component connections of *Top* while *Top* is in m^1_{Top} and m^2_{Top}, Fig. 5.3 shows the inner component connections of *Top* at the ProSys level. Similarly, Fig. 5.4 shows the inner component connections of *b* at the ProSave level, where the control flow and the data flow are separate. It is assumed that the inner component connection of a composite component for a specific mode can be independently specified without knowing the component connections in its other modes.

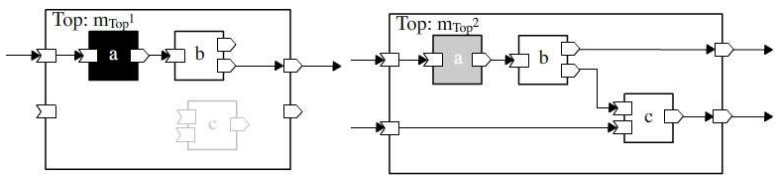


Figure 5.3: The inner component connections of *Top* at the ProSys level

5.3 Implementing the mode switch runtime mechanism

Section 4.2 has indicated that the mode switch runtime mechanism of MSL can be implemented by a pair of MSL subcomponents at the ProSave level, and an MSL subcomponent at the ProSys level. In this section, we shall apply the same ideas to composite components *Top* and *b* of this pedagogical system.

Since both *Top* and its subcomponents are ProSys components, the mode switch runtime mechanism of *Top* can be implemented by a primitive ProSys component

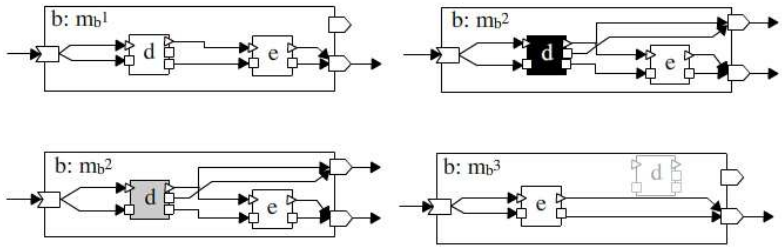


Figure 5.4: The inner component connections of *b* at the ProSave level

$\text{MSL}_{Top} \in SC_{Top}$, which can be automatically generated, given the mode mapping between *Top* and its subcomponents.

Fig. 5.5 presents MSL_{Top} as its ports explained as follows:

- p_i^{msx}: an input message port of *Top* for receiving an MSI. However, since *Top* has no parent, this port will not be used in this system.
- $P_i = \{p_i^a, p_i^b, p_i^c\}$: a set of input message ports of *Top* for receiving an MSR or MSC from its subcomponents *a*, *b* and *c*.
- p_o^s: an output message port of *Top* indicating its current mode. It will be used for merging the inner component connections of *Top* in m_{Top}^1 and m_{Top}^2.
- $P_o = \{p_o^a, p_o^b, p_o^c\}$: a set of output message ports of *Top* for sending an MSI to its subcomponents *a*, *b* and *c*.
- p_o^{msx}: an output message port for forwarding an MSR or sending an MSC to the parent. Since *Top* has no parent, this port will not be used in this system.

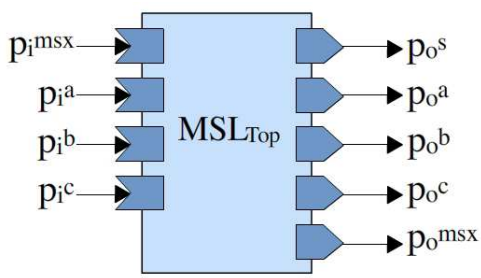


Figure 5.5: The MSL subcomponent of *Top*

The internal mode switch behavior of MSL_{Top} will follow Algorithm 3 described in Section 4.2.2.

At one level down, since b is a multi-mode ProSys component composed by ProSave components, its mode switch runtime mechanism can be implemented by a pair of MSL subcomponents of b, i.e. MSL_b^A and MSL_b^B, both of which can be automatically generated, given the mode mapping between b and its subcomponents.

Fig. 5.6 presents MSL_b^A and MSL_b^B and their ports. MSL_b^A has the following ports:

- p_i^t: the input trigger port whose activation makes MSL_b^A active.
- p_i^{msx}: an input trigger port for receiving an MSI from *Top*.
- p_i^{sync}: an input data port connected to MSL_b^B for synchronization.
- p_o^t: the output trigger port activated after MSL_b^A completes its current instance of execution.
- $P_o^{msx} = \{p_o^d, p_o^e\}$: a set of output data ports for sending an MSI to the subcomponents of b, i.e. d and e.
- p_o^s: an output data port indicating the current mode of c_i. It is used for merging the inner component connections of b when b is in m_b^1, m_b^2 and m_b^3.
- p_o^{sync}: an output data port connected to MSL_b^B for synchronization.

MSL_b^B has the following ports:

- p_i^t: the input trigger port whose activation makes MSL_b^B active.
- $P_i^{msx} = \{p_i^d, p_i^e\}$: a set of input data ports for receiving an MSR or MSC from the subcomponents of b, i.e. d and e.
- p_i^{sync}: an input data port connected to MSL_b^A for synchronization.
- p_o^t: the output trigger port activated after MSL_b^B completes its current instance of execution.
- p_o^{msx}: an output data port for forwarding an MSR or sending an MSC to *Top*, i.e. the parent of b.
- p_o^{sync}: an output data port connected to MSL_b^A for synchronization.

The internal mode switch behaviors of MSL_b^A and MSL_b^B will follow algorithms 1 and 2 described in Section 4.2.1.

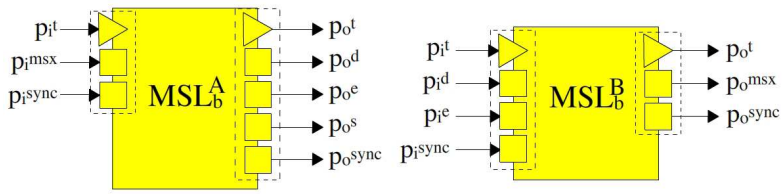


Figure 5.6: The pair of MSL subcomponents of *b*

5.4 Merging component connections

In Section 4.3, we have introduced our solution to merging component connections in different modes at both ProSys and ProSave levels. In this section, we demonstrate our approach by merging the inner component connections of *Top* and *b* respectively.

Fig. 5.7 illustrates the automatically generated view of merging the inner component connections of *Top* in m^1_{Top} and m^2_{Top} which are specified separately in Fig. 5.3. The ports of MSL_{Top} have been described in Fig. 5.5, and are thus not displayed here. It can be observed that MSL_{Top} and six primitive ProSys *Selection* components have been automatically generated. For each subcomponent of *Top*, each regular output message port (excluding the dedicated output message put) has an outgoing connection to a generated *Selection* component. Besides, Each input message port (excluding the dedicated input message port) of *Top* also has an outgoing connection to a generated *Selection* component. All *Selection* components have two output message ports because *Top* supports two modes. Furthermore, all *Selection* components have an input message port, marked in red in Fig. 5.7, that has in incoming connection from $\text{MSL}_{Top}.p^s_o$. Based on the current mode of *Top*, indicated by the message from $\text{MSL}_{Top}.p^s_o$, each *Selection* component will forward the data received from its preceding component to the corresponding output message port. Except $\text{MSL}_{Top}.p^s_o$, all the other ports of MSL_{Top} are connected to the dedicated mode switch message ports of *Top* and its subcomponents, strictly following the connection pattern described in Fig. 4.6.

Similarly, Fig. 5.8 illustrates the automatically generated view of merging the inner component connections of *b* in m^1_b, m^2_b and m^3_b which are specified separately in Fig. 5.4. The ports of the pair of MSL^A_b and MSL^B_b have been described in Fig. 5.6, thus not displayed here. Apart from MSL^A_b and MSL^B_b, a number of connectors have been automatically generated. For each subcomponent of *b*, each regular input trigger port has an incoming connection to a generated *Control Or* connector; each regular input data port has an incoming connection to a generated *Data Or* connector; each output trigger port has an outgoing connection to a generated *Selection* connector; and each output data port has an outgoing connection to a generated *Data Selection* connector. Besides, each input trigger port of *b* (excluding $b.p^{mst}_i$) has an outgoing connection to a generated *Selection* connector and each input data port of

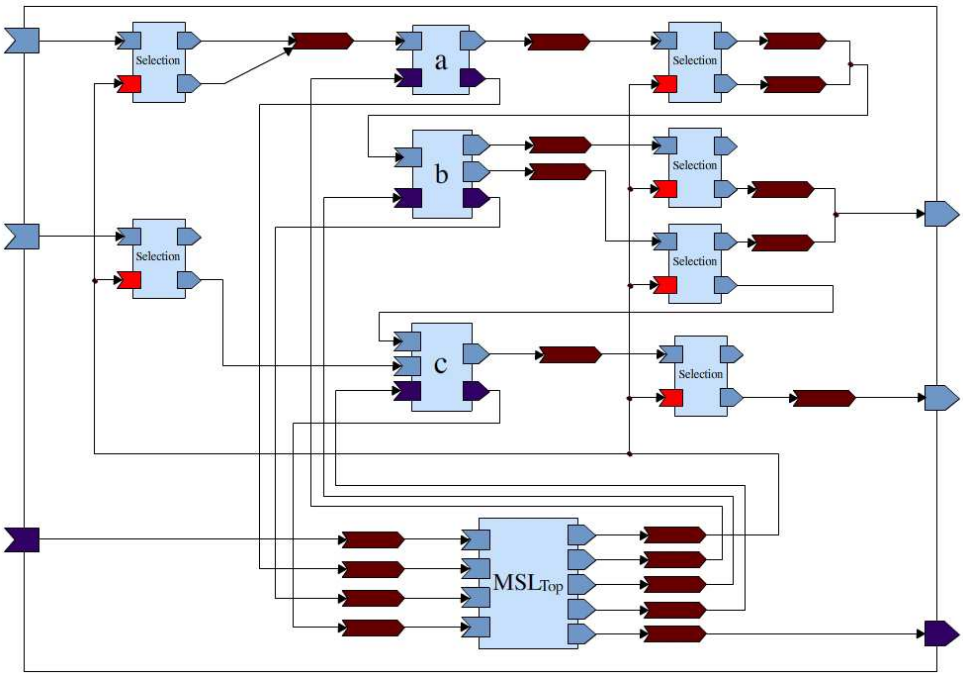


Figure 5.7: The merged inner component connections within *Top*

b) (excluding $b.p_i^{ms}$) has an outgoing connection to a generated *Data Selection* connector. All *Control Or* and *Data Or* have three input ports and all *Selection* and *Data Selection* have three output ports, because *b* supports three modes. Besides, all *Selection* and *Data Selection* have a special input data port, marked in red in Fig. 5.8, that has in incoming connection from $\text{MSL}_b^A.p_o^s$. Based on the current mode of *b*, indicated by the data from $\text{MSL}_b^A.p_o^s$, each *Selection* will activate the corresponding output trigger port, and each *Data Selection* will forward the data received from its preceding component to the corresponding output data port. The incoming and outgoing connections of MSL_b^A and MSL_b^B strictly follow the connection pattern described in Fig. 4.4.

Furthermore, since *b* is a ProSys component composed by ProSave components, a clock dedicated to mode switch is used to periodically trigger all ProSave components.

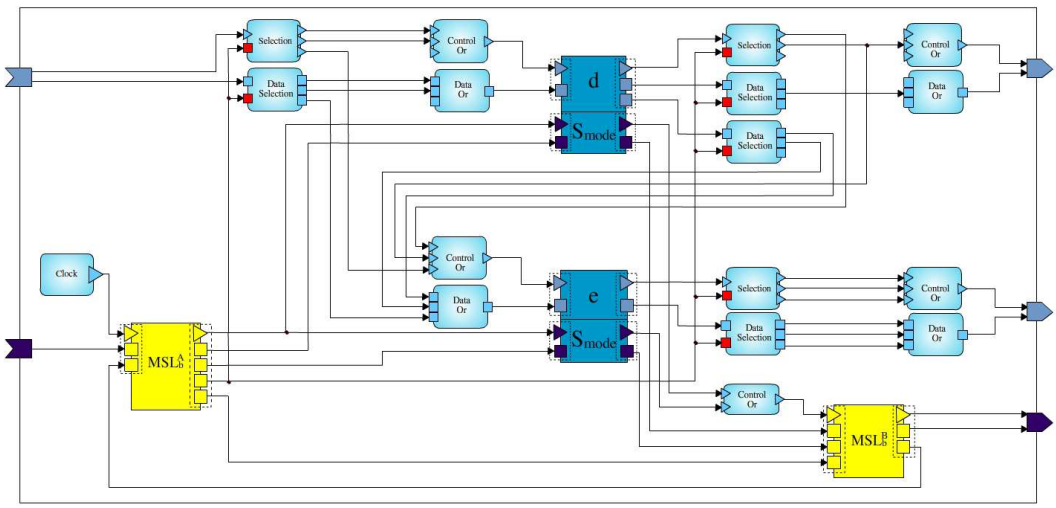


Figure 5.8: The merged inner component connections within *b*

Chapter 6

Conclusions

This thesis has presented an approach for implementing the Mode Switch Logic (MSL) in the ProCom component model. Both MSL and ProCom are developed by Mälardalen Real-Time Research Center (MRTC) at Mälardalen University. In Chapter 1, a basic introduction of the background and motivation of this thesis is provided. Chapters 2 and 3 describe the essentials of MSL and ProCom respectively. Then Chapter 4, the core of the thesis, explains our central ideas of implementing MSL in ProCom, including three major contributions: (1) the definition of multi-mode ProCom components; (2) implementing the mode switch runtime mechanism of MSL in ProCom; and (3) merging the connections between ProCom components in different modes. Since ProCom has two distinguished layers, i.e. ProSys and ProSave, which are quite different, each contribution considers both ProSys and ProSave layers which must be treated differently. Our approach only requires a minor modification (the introduction of the *Data Selection* connector) of the ProCom model in order to support MSL. Finally, in Chapter 5, our approach is further demonstrated by a pedagogical example.

As future work, our approaches of implementing MSL in ProCom shall be applied to the ProCom development tool PRIDE. Since our approach allows most of the MSL-related parts to be automatically generated, the development of Component-Based Multi-Mode Systems (CBMMSs) in PRIDE is expected to be relatively convenient and straightforward.

Bibliography

[1] 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 (CBSE'12)*, pages 183–188, June 2012.

[2] 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.

[3] V. K. Sharma and N. P. Gupta. Component-based software development. *IJCSNS International Journal of Computer Science and Network Security*, 10(11):132–134, November 2010.

[4] I. Sommerville. *Software Engineering (9th Edition)*. Addison Wesley, March 2010.

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

[6] 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 (CRTS'11)*, pages 38–45, November 2011.

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

[8] 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.

[9] 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.

[10] 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 (SIES'08)*, pages 177 –183, June 2008.

[11] 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.

[12] Autosar GbR: Autosar-technical overview. Technical report, AUTOSAR GbR. `http://www.autosar.org/index.php?p=3&up=1&uup=0`.

[13] J. Carlson, J. Feljan, J. Mäki-Turja, and M. Sjödin. Deployment modelling and synthesis in a component model for distributed embedded systems. In *Proceedings of 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA'10)*, September 2010.