Composable mode switch for component-based systems

Yin Hang, Hans Hansson
MRTC, Mälardalen University, Västerås, SWEDEN
Email: young.hang.yin@mdh.se

Etienne Borde
Telecom ParisTech, CNRS-LTCI, Paris, FRANCE

***Abstract*—Component based software development (CBD) reduces development time and effort by allowing systems to be built from pre-developed reusable components. A classical approach to reduce embedded systems design and run-time complexity is to partition the behavior into a set of major system modes. In supporting system modes in CBD, a key issue is seamless composition of multi-mode components into systems.**

In addressing this issue, we present a mode switch logic and algorithm for component-based multi-mode systems. The algorithm implements seamless coordination and synchronization of mode switch in systems composed of independently developed components. The paper provides formally defined semantics covering aspects relevant for mode switch, together with algorithms implementing mode switch rules for different types of components. The approach is illustrated by a simple example.

I. Introduction

Partitioning system behaviors into different operational modes is a frequently used approach to reduce complexity of adaptive systems design and verification, as well as to increase efficiency in system execution. Typically, for each mode different subsystems are executing, hence different software implementing different behaviors is executed.

For instance in a car, operational modes could be *off* and *engine on*. At runtime the system is initially running in the default mode (e.g., *off*) and when some condition changes or a particular event occurs, the system will switch to another pre-defined mode (e.g., *ignition on*). Mode switch analysis can be found in related ongoing researches, including mode switch protocols [1] and schedulability analysis during mode switch [2].

We will present a mode switch approach for component-based software. We consider component-based systems built by a set of hierarchically organized components. If multiple modes are supported, some components may reconfigure themselves during mode switch in order to provide different functionalities. Figure 1 illustrates the component hierarchy of a simple multi-mode system (used throughout this paper). The system supports two operational modes: M_1 and M_2. At top level, the system consists of components *a* and *b*. Component *a* consists of components *c*, *d* and *e*. However, Component *d* is deactivated (not in use; invisible) in mode M_2. Similarly, Component *b* has two subcomponents: *f* and *g* (deactivated in mode M_1). As *a* and *b* both have subcomponents, we call them *composite components*, and we call components that cannot be further decomposed (e.g., *c*, *d*) *primitive components*. A primitive component may have different behaviors due to different internal configurations in different modes. In Figure 1, Component *f* has one behavior in mode M_1 (distinguished by black color) and another behavior in mode M_2 (grey).

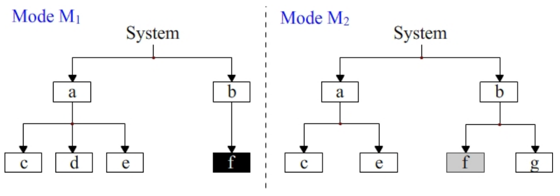

Fig. 1. Component hierarchy in different modes

In comparison with traditional component-based single-mode systems, we take mode switch into consideration and make rules for component reconfiguration throughout the system hierarchy. The main characteristic of Component-Based Development (CBD) is its focus on reuse. Hence, our mode switch mechanism should allow reuse of individual components in different systems. We will equip each component with mode-switch support that is independent of the context in which it is used and allows components to be composed without modification of the supporting mechanism.

Mode switch problems in CBD have been explored by various frameworks, including COMDES-II [3] and MyCCM-HI [4]. Besides, programming languages such as AADL [5] and programming models such as Giotto [6] and TDL [7], implemented in the Ptolemy II framework [8], provide semantics for the design of multi-mode systems. However, none of them comes up with a general mode switch logic (MSL) guiding the reconfiguration of hierarchically composed components. In this paper, we present a MSL for multi-mode systems, explaining how mode switch is triggered and propagated among related components and how component reconfiguration is implemented. Since our MSL is independent of applications, hardware, operating system, programming language or tools, it can be easily implemented in the CBD of most multi-mode systems, even though we in this paper assume a pipes-and-filters (control flow) type of component model.

II. Component Semantics for Mode Switch

In order to be compatible with our MSL, a component must be equipped with explicit interfaces related to mode switch and it must internally integrate certain rules to control its own mode switch process.

* This paper was accepted by the workshop APRES 2011 and some contents have been updated.

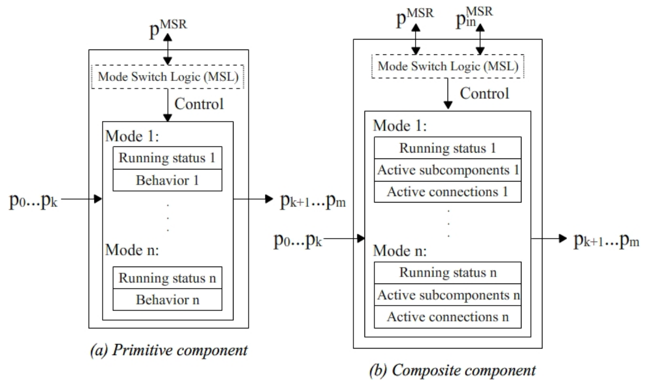

Fig. 2. Multi-mode component

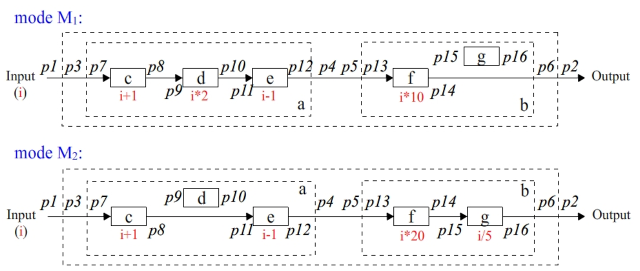

Fig. 3. System overview illustrating component connections

In the following we will concentrate on aspects related to mode switch, i.e., component behavior in a particular mode will not be covered.

Figure 2 illustrates multi-mode primitive and composite components, for which a few points need to be mentioned:

- A component typically has one or more input and output ports. Each port is externally connected to a neighboring component, its parent or subcomponents.
- For a primitive component, a *MSR*(Mode Switch Request) is received/sent via the input/output port p^{MSR}, dedicated to its mode switch related interactions with its parent. A composite component has the same type of port p^{MSR} and it also has another port p_{in}^{MSR} that is dedicated to the communication with its subcomponents during mode switch.
- The internal MSL defines how the component performs its own mode switch and also controls its own behavior. It will be described by algorithms in Section III.
- The configuration of a primitive component consists of its running status (*executing* or *deactivated*) and mode-specific behavior/code. The configuration of a composite component consists of its running status, executing subcomponents, connections in use between ports of its subcomponents and connections in use between its own ports and the ports of its subcomponents.

Each $c \in PC$ (the set of primitive components) is a tuple:

$$< P, M, B, MB, S, MSL >$$

where P is the set of ports of c; M is the set of operational modes supported by c; B is the set of behaviors of c; the function $MB : M \to B$ defines the behavior associated with a certain mode; the function $S : M \to \{Executing, Deactivated\}$ indicates if c is executing or not in a certain mode; and MSL is the mode switch logic integrated in c, described by Algorithm 1 in Section III.

Let CC denote the set of composite components. Each $c \in CC$ is a tuple:

$$< P, SC, Con, M, S, ESC, ACon, MSL >$$

where P is the set of ports of c partitioned into the disjoint subsets of input ports P_{in} and output ports P_{out}; SC is the set of subcomponents of c; $Con \subseteq (P_{sub} \cup P) \times (P_{sub} \cup P)$ is the set of connections between the subcomponents in SC and connections between c and SC, where P_{sub} is the set of ports of SC:

$$P_{sub} = \bigcup_{\substack{comp \in SC \\ p \in P_{comp}}} p;$$

M is the set of operational modes supported by c; the function $S : M \to \{Executing, Deactivated\}$ indicates for each mode $m \in M$ if c is executing or not in m; the function $ESC : M \to 2^{SC}$ defines the set of executing subcomponents in each mode; the function $ACon : M \to 2^{Con}$ defines the set of active connections (connections in use) in each mode; and MSL is the mode switch logic integrated in c, described by Algorithm 2 in Section III.

As an illustration, Figure 3 extends the example in Figure 1 with component connections. The sample system gets data from the input, processes data and generates output. Using i to denote component input, the behavior of each component is a simple numerical calculation, e.g., Component f calculates $i * 10$ in mode M_1, and $i * 20$ in mode M_2. The flow of data is indicated by arrows. In Figure 3, Component f is defined by the tuple,

$$< P_f, M_f, B_f, MB_f, S_f, MSL_f >$$

where MSL_f follows Algorithm 1 and:

$$\begin{array}{lcl} P_f & = & \{p_{13}, p_{14}\} \\ M_f & = & \{M_1, M_2\} \\ B_f & = & \{B_f(M_1), B_f(M_2)\} \\ MB_f & = & \{M_1 \to \{B_f(M_1)\}, M_2 \to \{B_f(M_2)\}\} \\ S_f & = & \{M_1 \to \{Executing\}, M_2 \to \{Executing\}\} \end{array}$$

Similarly, Component a is defined by the tuple

$$< P_a, SC_a, Con_a, M_a, S_a, ESC_a, ACon_a, MSL_a >$$

where MSL_a follows Algorithm 2 and:

$$
\begin{array}{lcl}
P_a & = & \{p_3, p_4\} \\
SC_a & = & \{c, d, e\} \\
Con_a & = & \{(p_8, p_9), (p_{10}, p_{11}), (p_8, p_{11}), \\
 & & (p_3, p_7), (p_{12}, p_4)\} \\
M_a & = & \{M_1, M_2\} \\
S_a & = & \{M_1 \to \{\textit{Executing}\}, M_2 \to \{\textit{Executing}\}\} \\
ESC_a & = & \{M_1 \to \{c, d, e\}, M_2 \to \{c, e\}\} \\
ACon_a & = & \{M_1 \to \{(p_8, p_9), (p_{10}, p_{11}), (p_3, p_7), (p_{12}, p_4)\}, \\
 & & M_2 \to \{(p_8, p_{11}), (p_3, p_7), (p_{12}, p_4)\}\}
\end{array}
$$

Note that, each pair of connections (e.g., (p_8, p_9)) implies a data flow from first to second element (e.g., p_8 to p_9).

III. Mode Switch Process

After the static structures have been defined, it is time to consider the dynamic mode switch process. The local reconfiguration of a primitive component is quite straightforward, but to coordinate and synchronize the reconfigurations of all related components throughout the system hierarchy is more challenging. The mode switch must be performed such that severe problems and anomalies (e.g. mode or data inconsistency, and mode switch failure) are avoided. Our Mode Switch Logic (MSL) is designed to eliminate these potential problems during mode switch by applying appropriate rules, and is based on the following assumptions:

- In each mode, all components are well-formed, e.g., there are no inconsistencies due to mis-matched ports, and no circular sequences of connections between components.
- Component reconfigurations and communication between components related to mode switch are fault free.
- At any time, at most one *MSR* is processed.

Due to space limitations we will in this paper make the following additional simplifying assumptions:

- The execution of primitive components can be aborted at any time (to allow immediate response to a *MSR*).
- All components support the same modes (to avoid the need for a mode mapping mechanism).

Our MSL consists of a MSR propagation mechanism and dependency rules.

Mode switch request propagation: A *MSR* is initially issued by some component and then propagated to other components until all the components get notified. All composite components propagate the *MSR* to their subcomponents and parents. Only the primitive component which is the *MSR* source propagates the *MSR* to its parent.

Let's demonstrate the MSR propagation mechanism using the example in Figure 3. Suppose the *MSR* is initiated from Component c in both modes. Component c is primitive and propagates the *MSR* to its parent a and itself to trigger mode switch. Once Component a receives the *MSR*, it propagates it in two directions: to its subcomponents rather than the triggering source Component c (d and e) and to its parent, which is the top level system. Since the subcomponents d and e are primitive, there is no further MSR propagation from them. The top level system has no parent, and will only propagate the *MSR* to its subcomponent b, which only needs to propagate the *MSR* to its subcomponents f and g. Since f and g are primitive, the MSR propagation is finally terminated. Once a component completes its MSR propagation, it can start its own mode switch process. The entire MSR propagation process is depicted in Figure 4.

Dependency rules: The mode switch completion of a component may have dependency on other components. To prevent mode inconsistency, unpredictable results or other related mode switch problems, we require the coordination of mode switches between different components to comply with the following dependency rules:

1) A composite component cannot complete its mode switch before the completion of the corresponding mode switch in its subcomponents.
2) A component cannot complete its mode switch before the mode switch completion of all components connected to its ingoing ports.
3) For components with parents, Rule 2 cannot be applied until the parent has updated the component connections for the new mode.

Components that cannot proceed with the mode switch due to a dependency rule are temporarily blocked until the corresponding condition is satisfied.

Algorithms for primitive and composite components: Before presenting the algorithms, we will introduce some notations.

- m_i denotes the initial mode of a component.
- $AP(m)$ denotes the set of ports in use, i.e. ports that in mode m are included in $ACon(m)$.
- $Wait$ and $Signal$ are blocking primitives for receiving input and sending output on a specific port.
- *MSR* is the mode switch request signal, carrying the identity of the new mode and the sending component.
- $Prepare\_for\_new\_mode(m_{old}, m_{new})$ changes running status and behavior for a primitive component. For a composite component, it changes running status and connection. It may also include some cleaning up in the old mode and preparation for the new mode.
- $c.p$ denotes the port p of Component c.
- top is a boolean variable only set to $true$ for the top level composite component.
- $last(c)$ evaluates to $true$ only if all ports in $AP^c_{out}(m_{new})$ are connected to ports of the enclosing composite component.
- $parentOK$ is a signal used to tell subcomponents that the parent reconfiguration is completed; a parameter indicates if a response is required.
- ms_done signals completion of mode switch.

We assume that the *MSR* originates from one of the primitive components and that component execution is aborted upon reception of a *MSR*. Details and description of this are outside the scope of this paper.

Algorithm 1 and 2 describe the mode switch processes of primitive and composite components respectively. These two algorithms have integrated mode switch request propagation mechanisms and dependency rules guiding a successful global

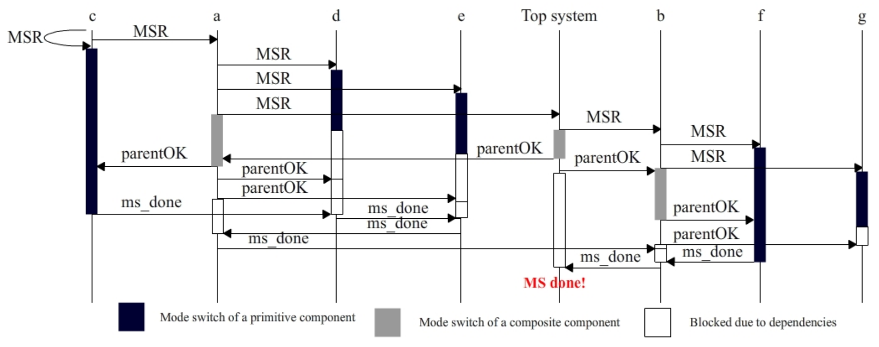

Fig. 4. MSR propagation and mode switch activity (from M_2 to M_1)

Algorithm 1 $AlgPC.mode\_switch(x \in PC, m_i \in M_x)$

```
$current\_mode := m_i;$
loop
  $Wait(p^{MSR}, MSR(m_{new}, origin));$
  $Prepare\_for\_new\_mode(current\_mode, m_{new});$
  $Wait(p^{MSR}, parentOK(resp\_required));$
  $\forall p \in AP^x_{in}(m_{new}) : Wait(p, ms\_done(m_{new}));$
  $\forall p \in AP^x_{out}(m_{new}) : Signal(p, ms\_done(m_{new}));$
  if $resp\_required$ then
    $Signal(p^{MSR}, ms\_done);$
  end if
  $current\_mode := m_{new};$
  if $S_x(m_{new}) = Executing$ then
    $exec(MB^x(m_{new}));$
  end if
end loop
```

Algorithm 2 $AlgCC.mode\_switch(x \in CC, m_i \in M_x, top)$

```
$current\_mode := m_i;$
loop
  $Wait(p^{MSR} \wedge p^{MSR}_{in}, MSR(m_{new}, origin));$
  if $origin = parent$ then
    $\forall c \in SC_x : Signal(c.p^{MSR}_{in}, MSR(m_{new}, x));$
  else
    $\forall c \in (SC_x \setminus \{origin\}) :$
    $Signal(c.p^{MSR}_{in}, MSR(m_{new}, x));$
    if $\neg top$ then
      $Signal(p^{MSR}, MSR(m_{new}, x));$
    end if
  end if
  $Prepare\_for\_new\_mode(current\_mode, m_{new});$
  $\forall c \in SC_x : Signal(c.p^{MSR}_{in}, parentOK(last(c)));$
  if $\neg top$ then
    $Wait(p^{MSR}, parentOK(resp\_required));$
  end if
  $\forall p \in (AP^x_{in}(m_{new}) \vee p^{MSR}_{in}) :$
  $Wait(p, ms\_done(m_{new}));$
  $\forall p \in AP^x_{out}(m_{new}) : Signal(p, ms\_done(m_{new}));$
  if $resp\_required$ AND $\neg top$ then
    $Signal(p^{MSR}, ms\_done(m_{new}));$
  end if
  $current\_mode := m_{new};$
end loop
```

mode switch. A few points deserve further explanation:

- Upon reception of a *MSR*, the currently executing component software is aborted and control is handed over to the MSL as defined by either Algorithm 1 or 2.
- If a composite component receives the *MSR* from its parent, it will propagate it to all its subcomponents. However, if the *MSR* comes from a child, it will propagate it to all other subcomponents rather than this child and to its own parent if it has one, because it is not efficient to propagate the same *MSR* to any component more than once. When a composite component propagates the *MSR* to its subcomponents, each subcomponent $c \in SC$ receives the *MSR* via the dedicated port p^{MSR}.
- $exec(MB^x(m_{new}))$ means that x starts to execute its behavior defined for mode m_{new}.

IV. Conclusion and Future Work

We have presented a Mode Switch Logic (MSL) for component-based systems with multiple operational modes. We first defined component semantics and configuration, and then proposed rules to guide the dynamic mode switch process, including a mode-switch request propagation mechanism and dependency rules. Most key concepts and rules are also demonstrated by a small example throughout the paper. We finally gave algorithms showing how to implement our MSL.

In future work we intend to generalize our results by lifting some of the simplifying assumptions, e.g., by restricting the interruptibility of components and by allowing different components to have different modes. We will additionally develop timing analysis for multi-mode systems, verify the correctness of proposed algorithms, as well as implement and evaluate the techniques in our component model ProCom [9], which is targeting real-time embedded systems.

Acknowledgment

The anonymous reviewers, Jan Carlson and Cristina Seceleanu are acknowledged for valuable feedback. This work is supported by the Swedish Research Council.

REFERENCES

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

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

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

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

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

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

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

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

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