Chapter 1 Introduction

Modeling & Simulation approach [Zei00] has received increasing interest for its sound mechanism enabling fine representing of the discrete event dynamic systems. A general conceptual framework of Modeling & Simulation [Kim01; Zei02; Zei03] constitutes the three basic entities: the real system fitting in certain experimental framework, model, and simulator. The real system fitting in certain experimental framework represents a real or virtual environment in which the source data under analysis of interest to the modeler are collected. Model offers two facets of abstractions of the real system. The behavior of model is a set of input/output data comparable to that observable in the real system. The structure of model is the set of instructions to generate the data. Simulator executes the instructions of the model and really generates the behaviors of the model. Two kinds of relationships bridge the three basic entities. Modeling relation reflects the approximation of the model behaviors to the real system in a specified experimental framework; while simulation relation lying between a model and a simulator represents how faithfully the simulator carries out the instructions of the model. The M&S framework benefits from the separated concerns between modeling and simulation. On one hand, the same model can be simulated with different simulators, allowing portability and interoperability at a high level of abstraction. On the other hand, well-defined separation facilitates verifications of models and simulators independently and reusability in later combination with minimal re-verification.

Benefited from the precise mathematical specification and the underlying sound M&S framework, DEVS (Discrete Event System Specification) [Zei76, Zei00] has proved to be a universal modeling mechanism for discrete event dynamic systems. The DEVS

formalism provides a means of specifying a mathematical object called a system, in which a time base, inputs, states, outputs, and functions for determining next states and outputs given current states and inputs are defined. Certain constellations of such parameters render fine system abstractions and allow the possibility to analyze the system behaviors thoroughly.

DEVS-based systematic approach has gained popularity in the real time application due to the fact that it enables the smooth transformation from modeling to executing code in real time environment with the help of the RT-DEVS [Hon97], an extension of the original DEVS formalism. RT-DEVS allows DEVS models interact with surrounding environment, such as software components, hardware components or human operators, in real time. Aided by RT-DEVS, a real-time DEVS-based experimental framework (eCD++) [YuJ07] is devised to facilitate development of real-time embedded systems (RTS). eCD++ takes advantage of the hardware-in-the-loop technology [Gli04a] to establish a high level DEVS-based experimental environment for the real time embedded systems.

1.1. Problem Statement

eCD++ is a systematic toolkit assisting development of embedded real-time systems based on P-DEVS formalism [Cho94]. By permitting developing hybrid software and hardware systems and smooth transformation from the DEVS models to the hardware counterpart, eCD++ provides a DEVS-based real-time experimental framework, on which the embedded real-time systems can be designed and implemented effectively and safely.

As well known, embedded real-time systems [Kop00] are of critical timeliness and rigorous correctness of system behaviors. Moreover, most embedded real-time systems

are highly reactive artificial systems that deliver data from/to devices interacting with the surrounding environment (another artificial/natural system). Improper decisions may lead to catastrophic consequences for assets or lives. The traditional DEVS simulation approaches are too rigid to fit the varied requirements of embedded real-time systems, such as adjusting the system structure to respond to the changing environment, recovering from the fault automatically and self adaptability etc.

Due to the absence of dynamic structure, eCD++ fails to meet the challenges the real time embedded systems pose. Dynamic structure is a feasible solution to fitting the varied environments or recovering from errors automatically. Flexibility and reliability, therefore, could be reached by adjusting the structures of models dynamically.

Our work aims to introduce a Flexible Dynamic Structure DEVS algorithm (FDSDE) [Sha06] into eCD++. FDSDE defines a set of new message-passing algorithms [Sha07] to support the dynamic structure changes in RTS. The new experimental environment namely DS-ECD++ is developed equipped with an improved simulation engine that combines FDSDE with the P-DEVS real time simulation engine to adapt to not only the dynamic structure real-time simulation but also the real-time embedded system development. Dynamic structure DEVS, to some extent, makes it possible for the system designers and developers to improve the reliability and performance of the Real-Time embedded systems.

1.2. Contributions

The purpose of the thesis is to provide revised message-passing algorithms for each abstract simulator used in eCD++. The new message-passing algorithms are compatible

with the functionalities of eCD++ and are capable of conducting dynamic structural changes during the running of simulation. The major contributions of the thesis are list as the following:

- The message-passing algorithms for the existed abstract simulators in eCD++ are redesigned to allow processing both dynamic structural change and regular simulations. The redefined abstract simulators are also compatible with the major functionalities of eCD++.
- We identify the coupled models which are subject to experiencing structural changes as structure components. Structure agent is proposed to play as a structural representative executing structural changes on behalf of structure components. A set of structural change operations are specified structure agents and are invoked by the user-defined structure agents. Moreover, a new message-passing algorithm to process the behaviors of structure agents are presented.
- The basic structure change forms and the operation boundaries for the structural change operations are discussed.
- A variety of structural change scenarios are devised and further a couple of structural change cases are figured to verify the correctness of the functionalities of DS-eCD++.

1.3. Thesis Organization

The rest of the thesis is organized as follows:

Chapter 2 reviews the related literature. Firstly, the original DEVS formalism and the extensions of DEVS formalism including RT-DEVS and P-DEVS are introduced. Dynamic structure DEVS formalisms are listed as the underlying theoretical base of this

work. Moreover, the applications of dynamic structure DEVS are surveyed. Finally, we depict the two DEVS-based toolkits: Standalone CD++ and eCD++, which are the base of the thesis.

Chapter 3 depicts the FDSDE algorithm. The message-passing algorithms for each abstract simulator are exhibited. The typical message-passing scenarios and simulation phases are explained.

Chapter 4 addresses the software architecture of DS-eCD++. The implementing issues in each software component are explicitly described in the following. The functionalities of DS-eCD++ are also discussed in this chapter.

Chapter 5 puts forward the structural change scenarios. Two cases together with a series of experiments are conducted to verify the logic and the functionality of DS-eCD++.

Chapter 6 draws conclusions of the thesis and discusses the possible future work.

Chapter 2 Review of the State of The Art

This chapter presents a review of the state of the art in the filed of DEVS-based modeling and simulation technology. Especially, the dynamic structure DEVS and DEVS simulation in real-time domain are explored. The original DEVS formalism and two extended DEVS formalisms P-DEVS and RT-DEVS are illustrated in the first section. The DSDE and dynDEVS shown in the following section are the two extensions of DEVS to dynamic structure change. In addition, the dynamic structure DEVS and the applications are explored to demonstrate how powerful the dynamic structure change brings to the complex physical systems. Moreover, the researches of dynamic structure DEVS modeling and simulation in real-time embedded systems are surveyed. Finally, we introduce the standalone CD++ and ECD++, which are specific toolkits based on DEVS theory.

2.1. DEVS Formalisms

The set-theoretical definition of DEVS and its extensions are presented in this section. DEVS formalism is firstly introduced and it is a basis of other extended DEVS formalisms. P-DEVS improves the original DEVS formalism by eliminating serialization restrictions. RT-DEVS, based on P-DEVS, is a specified DEVS formalism executed in real time.

2.1.1. DEVS Formalism [Zei76, Zei00]

DEVS is a formal modeling and simulation framework based on systems theory. DEVS has well-defined concepts for coupling of components and hierarchical, modular model composition. DEVS defines a complex model as a composite of basic components (called

atomic model), which can be hierarchically integrated into coupled models. A DEVS atomic model is defined as:

$$M = <X, S, Y, \delta_{int}, \delta_{ext}, \lambda, ta>$$

Where X is a set of input events; Y is a set of output events of the atomic model; S is a set of partial states associated with the atomic model; ta represents the lifetime of each state in S; δ_{ext} is external transition function, this function is triggered when an input event in X is received by the atomic model; λ is output function; δ_{int} is internal transition function, if there is no external event comes, the current state will be kept for its lifetime ta, then output event might be triggered determined by λ and produce output event Y, at the same time internal state change will happen determined by internal transition function.

A DEVS coupled model is defined as:

$$CM = <X, Y, D, \{M_i\}, \{I_i\}, \{Z_{ij}\}, Select>$$

Coupled model is defined as a set of atomic models M_i ($i \in D$) which certain a set of interactions through their interface (X, Y). M_i is a basic DEVS model (atomic or coupled); I_i is the set of influencees of model i; for each $j \in I_i$, Z_{ij} is the i to j translation function to convert the output of M_i to the input of M_j. The property, closure under coupling, allows coupled model taken equally as atomic model, which enables model reuse. Select is the tie-breaking function for imminent components.

The definition of DEVS formalism may raise two types of ambiguity. One type of ambiguity may rise when multiple components in a coupled model are imminent at the

same time. DEVS formalism employs Select function to solve the ambiguity. By defining an order over the imminent components, only one imminent component in a coupled model is allowed to execute its internal transition function. Other imminent components are divided into two groups: the ones receiving an external event from this model or the remaining. The former group invokes the external transition function with e = ta(s); while the later group is imminent in the next simulation cycle and may need to be selected again to decide the execution sequence. The serializing execution produces message redundancy; therefore leads to potential executing bottleneck. The other ambiguity is caused when an atomic model receives an external event at exactly same time its internal transition is scheduled. The execution sequence is not specified in DEVS formalism. It is the DEVS software's responsibility to determine which function goes first. The serialization constraint, however, may not reflect the reality and cause errors.

2.1.2. P-DEVS Formalism

P-DEVS [Cho94] is an extended DEVS formalism eliminating the two types of ambiguity of the original DEVS formalism. A confluent function is added to the atomic model to dispose the transition collisions in atomic models. An atomic model is defined as:

$$M = <X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta>$$

Where

X: a set of input events

Y: a set of output events

S: a set of sequential states.

δ_{int} : S→S: internal transition function

δ_{ext}: Q x X^b → S: external transition function, X^b is a set of bags over elements in X,

$\delta_{ext}(s, e, \Phi) = (s, e)$.

δ_{con}: S x X^b → S: confluent transition function.

λ : S → Y^b: output function

ta : S → $R_0^+{}_{\to\infty}$: time advance function,

with Q = {(s, e) | s ∈S, 0 < e < ta(s)}, e is the elapsed time since last state transition.

The semantics of the P-DEVS definition introduce confluent transition function, which handles the collision behavior when an external event arrives at the same time of its internal transition, e = 0 or e = ta(s). The confluent transition function allows processing model behavior in parallel instead of serialization.

The coupled model in P-DEVS presents the following structure:

CM = <X, Y, D, {M_i}, {I_i}, {Z_{ij}}>

Where

X is a set of input events,

Y is a set of output events

D is a set of components

For each I in D, M is a component

For each I in D U {self}, I_i is the influencees of i.

For each j in I_i, $Z_{i,j}$ is a function, the I-to-j output translation.

P-DEVS formalism furnishes two advantages over the original DEVS formalism by eliminating the two types of ambiguity. Each model is equipped with a message bag to handle the simultaneous events, by which the tie-breaking function, Select, is removed and all imminent components can be activated in parallel. Another type of ambiguity is eliminated by employing a confluent transition function in an atomic model. For those components experiencing internal and external transitions collide, the confluent transition function is invoked instead of either internal or external transition function to calculate the new state. The confluent transition function leaves the executing sequence of internal and external transitions to the modelers. It is reasonable for the modelers to determine the state transition of the models in the presence of collisions according to the system real requirement. Since P-DEVS formalism overcomes the deficiencies in DEVS definition, it enables more effective and more reasonable modeling of the target systems. The real time simulation engine in eCD++ complies with the P-DEVS principle.

2.1.3. RT-DEVS Formalism

The real time DEVS formalism [Hon97] is an extension of the DEVS formalism for the real time application. An atomic model in RT-DEVS is defined as:

RTAM = <X, S, Y, $\delta_{int,}$ δ_{ext}, δ_{con}, λ, ta, Ψ, A>

Where, ta is a time interval function given by an interval $ta(s)|_{min} \leq ta(s) \leq ta(s)|_{max}$, $s \in S$. Ψ: S→A is an activity mapping function. A is a set of activities $A = \{a| t(a) \in I^{+}_{0,\infty}$ and $t(a) \leq ta|_{max}\} \cup \Phi$.

A real time DEVS coupled model connects basic real time DEVS models together to form a new model. A real time DEVS coupled model is structured as:

RTCM = <D, $\{M_i\}$, $\{I_i\}$, $\{Z_{i,j}\}$>

Where, D is a set of components. For each i in D, M_i is a basic real time DEVS model, I_i is a set of influences of i. For each j in I_i, $Z_{i,j}$ is an i-to-j output translation.

The RT-DEVS formalism replaces virtual time advance in the DEVS formalism by real time advance. The time advance function is no more a fixed value. Instead, a time interval is defined. The RT-DEVS simulator checks a specified time advance of a RT-DEVS model against a real time clock. $Ta(s)|_{min}$ is an auxiliary parameter used to verify the time correctness during simulation. A set of activities associated with a state is defined in parametersΨ and A.

2.2. Dynamic Structure DEVS

Dynamic structure is also called variable structure. Zeigler coined the term “variable structure models” to describe models that contain in descriptions of their behavior the possibility of altering their own structure and, consequently, their behavior [Zei86; Zei89;

Zei91]. Structural changes might concern the model's behavior rules and attributes or, presupposing a compositional construction of models, might refer to a model's components and interactions. Dynamic Structure provides a desirable solution to capturing the dynamic nature of the discrete event dynamic systems and allows runtime simulation tuning.

2.2.1. Dynamic Structure DEVS Formalisms

The structure extensions to the DEVS formalisms have been made to regulate the dynamic structure definitions. DSDEVS [Bar95; Bar97] introduces a structural entity called network executive to conduct the structural changes for the network model in a centralized way. Pawletta [Paw96] employs a structure event condition and structure event action pair to represent a structure state of an atomic model or a coupled model. Uhrmacher [Uhr01] developed the Pawletta's algorithm by offering a complete definition in both structural and behavioral perspectives of an atomic model or a coupled model. Uhrmacher's algorithm captures the intrinsic reflective nature of variable structure model by offering a recursive definition. In this section, we will explain the formalisms of Barro's algorithm and Uhrmacher's algorithm.

DSDE Formalism

DSDE divides models into two groups: basic and network models. The basic models are atomic structure units which cannot be split. The network models are coupled components, composed of multiple basic structure models and interconnections that involve structural changes. A Network Executive is a modified basic model to conduct structural changes in network models. The Network Executive stores all possible states of structural changes and

their corresponding component sets in each structural state. The two parts are associated together through an index function in the Network Executive. A DSDE network is a 4-tuple [Bar01]

$$DSDEN_N = (X_N, Y_N, \chi, M_\chi),$$

Where X_N is the network input value set; Y_N is the network output value set; χ is the name of the dynamic Network Executive; M_χ is the model of the Network Executiveχ, which is a modified basic model and is defined by

$$M_\chi = (X_\chi, s_{0,\chi}, S_\chi, Y_\chi, \gamma, \Sigma^*, \delta_\chi, \lambda_\chi, \tau_\chi).$$

Here γ: $S_\chi \rightarrow \Sigma^*$ is the structure function, Σ^* is the set of network structures. A structure $\Sigma_\alpha \in \Sigma^*$ associated with the executive partial state $s_{\alpha,\chi} \in S_\chi$ is given by $\Sigma_\alpha = \gamma(s_{\alpha,\chi})$ = $(D_\alpha \{M_{i,\alpha}\}, \{I_{i,\alpha}\}, \{Z_{i,\alpha}\})$, where D_α is the set of component names associated with the executive partial state $s_{\alpha,\chi}$; for all $i \in D_\alpha$, $M_{i,\alpha}$ is the model of the component i; for all $i \in D_\alpha \cup \{\chi, N\}$, $Z_{i,\alpha}$ is the set of component influencers of i; for all $i \in D_\alpha \cup \{\chi\}$, $Z_{i,\alpha}$ is the input function of the component i; and $Z_{N,\alpha}$ is the network output function. Changes of a basic model include structural changes within the basic model or changes on transition/output functions of this basic model. A Network Executive should be used together with the basic model to composite a network model (only the Network Executives can conduct structural changes).

DynDEVS Formalism

The dynamic DEVS formalism [Uhr01] does not introduce an extra component to conduct dynamic structural changes. Instead, ρ_α, a model transition function, is included. There are two kinds of dynamic DEVS models: dynDEVS (atomic) and dynNDEVS (coupled). The dynDEVS models are atomic structural components with the structure

$$dynDEVS =_{df} \langle X, Y, minit, \mathrm{M}(minit)\rangle$$

where X, Y are the structured sets of inputs and outputs; $minit \in \mathrm{M}(minit)$ is the initial model, where M(*minit*) is the least set having the structure{<*S*, *sinit*, δext, δint, $\rho\alpha$, λ, *ta*> }. dynNDEVS models are coupled structural components with the structure

$$dynNDEVS =_{df} \langle X, Y, ninit, \mathrm{N}(ninit)\rangle$$

where X, Y are the structured sets of inputs and outputs; $ninit \in \mathrm{N}(ninit)$ is the start configuration and N(*ninit*) is the least set having the structure {<*D*, ρ_N, {*dynDEVSi*}, {*I*}, {*Zi*, *j*}, *Select*>}. A model's state space, internal and external transition, output, time advance, and model transition functions are subject to change during simulation. A dynDEVS can be interpreted as a set of DEVS models with the same interface plus a transition function that determines which DEVS model succeeds the previous one. Agents associated with dynDEVS or dynNDEVS models hold the worldview knowledge of their corresponding models and environments. Agents are responsible for initiating structural changes and executing the structural change process.

The dynamic structure DEVS formalisms make it possible to represent discrete event dynamic systems more precisely; therefore, enable dynamic structure DEVS to represent dynamic structural behaviors in the DEVS simulation, complex systems design and developments and so on.

2.2.2. Dynamic Structure and the Applications

Dynamic structure DEVS provides a salient supplementary to the DEVS theory to represent and simulate the structural changes in interactions, composition, and behavior patterns. Dynamic structure DEVS promotes the Modeling & Simulation methodology [HuX03] at three levels. 1) It offers natural and effective way to model the complex systems which exhibit structural changes and behavioral changes to respond to different situations. Adaptive computer architecture [Bra94] is established using dynamic structure to achieve a desired computing performance. An ecological system [Uhr93] calls for dynamic structure to reflect the evolvements of the elements in the system. It is hard to model and simulate the structural changes in the above systems without dynamic structure. 2) Dynamic structure brings additional flexible to the systems design and development. The dynamic distributed robotic system [HuX03] exhibit dynamic reconfigurations as robots interact and make decisions in dynamic environments employing dynamic structure. A flexible manufacturing system [Her00] is able to switch among the different product processes online. Benefited from dynamic structure, the dynamic issues can be captured in the system development stage and embodied in the implementations; therefore, the flexibility and reliability of the system can be achieved. 3) The dynamic structure permits loading only a sub-set of the components for simulation. It is very useful in a very complex

system containing tremendous members as only the active components are loaded dynamically to conduct the simulation.

2.3. Modeling & Simulation Methodology in Real-Time Systems

Much research effort has been put in the development of real-time systems. In the real-time simulations, the simulation time should be synchronized as closely as possible to the clock time of the underlying computer system [Zei93]. The real time simulation frameworks, including DEVS-Scheme, The layered design approach for distributed real-time systems [Cho00] [Cho01a] and the real-time simulation framework based on RT-DEVS [Cho01b], are helpful attempts of applications of Modeling & Simulation methodology in the real-time field. Based on the real-time simulation frameworks, a series of methodologies are proposed to realize the transformation from the modeling stage to the design stage in real-time embedded systems, such as DEVS-on-a-Chip [HuX01], Robot-in-the-loop projects [HuX05a] and Hardware-in-the-loop [LiL03; Gli04a; YuJ07] etc. However, the static structure allowed in the frameworks makes it difficult to respond to the changes in the residing internal / external environments, which always call for dynamic structural or behavioural changes to maintain the flexibility and reliability in real-time embedded systems.

Modeling and Simulation with dynamic structure offers necessary modeling of the dynamic structural changes and behavioural changes. The dynamic structure can be applied in e-Commerce applications [Liu03] enabling a dynamic business process to meet the instant requirements, and scale to large and small business activities. In a manufacturing system, a routing for a product specifies a given sequence of manufacturing

workstations or machines. If some workstations or machines are replaced, then the routings requiring those machines must be updated accordingly. Dynamic structure is a desirable solution to update the routings online in a flexible manufacturing system [Her00]. Moreover, dynamic reconfiguration of some components in the real time systems realizes runtime simulation tuning [Mit06]. The rapid feedback cycle allows experimentation with parameters and structures and results in effective model configuration that is difficult to achieve when turnaround requires hours or days. The dynamic team formation in the distributed robotic system [HuX05d] is a meaningful attempt of dynamic reconfiguration of the components in a real time distributed system. Each robot is taken as an independent component and can be reconfigured by establishing the couplings between the robots; therefore a Leader-Follower match can be conducted. During this process, the couplings between the models can be added and removed, resulting in a variable structure system. Also, the real-time implementation enables an execution of hybrid software components and hardware components system; therefore promotes a smooth transformation from the simulation modeling to design of real time systems. [HuX04] and [HuX05b] present how the virtual robot models are replaced by the real robots while maintaining model continuity. By studying the cooperative robotic system [HuX05c], a stepwise incremental study process for development of the real-time embedded systems with dynamic structural changes is also proposed. All of those researches demonstrate that the dynamic structure is a desirable solution in the modeling & simulation methodology, especially in the development of the real time embedded systems applying the modeling & simulation methodology.

2.4. Introduction to the CD++ Simulation Toolkit

CD++ [Wai02] is a modeling and simulation software family based on the DEVS theory. In which atomic models are defined using a state-based approach (encoded in C++ or an interpreted graphical notation); while coupled models contain atomic models composition and interconnecting information of those atomic models. CD++ has been widely used in various applications from simple queuing systems to complex systems [Mac04] such as environmental systems [Wai06] or complex real-time systems [Gli04b]. CD++ employs the abstract simulator mechanism to exchange messages among the processors while simulation advances. A *Simulator* component is in charge of executing the behaviour of atomic models while a *Coordinator* component takes charge of the message processing of coupled models. The simulation evolves through message-passing, using six kinds of messages: I (Initialization), * (Internal), X (Inputs), Y (Output), @ (Collect) and D (Done). Different versions of CD++ have been developed to facilitate various applications. Stand alone CD++ implements DEVS and Cell-DEVS simulation. Parallel CD++ [Liu06; Liu07] is aiming to enhance the performance of Cell-DEVS simulation by distributing calculation of different cells over multiple processors. Distributed CD++ [Ma] is developed to facilitate the coordination of the different simulating engines in different sites through the standard distributed computing protocols. Real time embedded CD++ (eCD++) is a DEVS-based systematic developing tool constructed especially for Real-Time embedded system.

eCD++ [YuJ07] is a version of the CD++ software family that has been adapted for real time and embedded system applications. eCD++ employs the Model-Driven Architecture of real-time systems (MDA) [Wai05] to construct a high level experimental environment

for the development of real-time systems. The software is modularized as a group of components that have well-defined behaviors and have independent functionalities. Four major components are included: the Main Simulator, DEVS Modeling Subsystem, Simulation Subsystem and Messaging Subsystem. It is based on the P-DEVS formalism, which provides the modeling principles to characterize the structural and behavior aspects of real-time systems. Moreover, RT-DEVS enables eCD++ to simulate the hybrid software and hardware systems. Finally, eCD++ supports smooth transformations from simulation models to real components of the systems. The Flat Coordinator in eCD++ provides an alternative simulation fashion by eliminating the coordinators in the hierarchy and exchanging messages directly between the flat coordinator and simulators. The GGAD interpreter (Generic Graphical advanced environment for DEVS modeling and simulation) in eCD++ enables to specify atomic models graphically. It is an easier way for the non-expert users to build atomic models intuitively.

Chapter 3 the Flexible Dynamic Structure DEVS Algorithm

Flexible Dynamic Structure DEVS Algorithm is a new structural paradigm based on the DSDEVS formalism. The FDSDE supports various structural changes in the DEVS-based framework including changes of DEVS models composition, changes of the couplings among the DEVS models and changes of input/output ports of the coupled models. The structural changes are implemented dynamically during a simulation running according to the structural state variables.

In FDSDE, a conception of a structure component refers to a coupled model subject to the structural changes. A structure agent is introduced to execute the structural changes for a structure component. As defined in the DSDEVS formalism, the possible model structures of a structure component constitute the state space of a structure agent. Each model structure of a structure component is mapped into a structural state of its associated structure agent and is connected with a structural value. The structural state transitions of a structure agent are triggered by a structural change message. A new abstract simulator, RevSimulator, specifies the message-passing paradigm for the structure agent. The messages related to structural change are defined for the structural change processes.

Message Definitions

The simulation is advanced with exchanging different kinds of messages among the simulation processors. Two categories of messages are defined for simulation in eCD++: control messages and content messages. Control messages consist of the initialization message (I), the internal message (*), the collect message (@) and the done message (D); while content messages include the external message (x) and the output message (y). The

external messages and the output messages exchange simulation data between simulation models. The initialization message indicates the start of simulation. The collect message and the internal messages invoke the output functions and the state transition functions of the atomic models respectively. The timing information is carried through D messages for synchronization. The introduction of dynamic structure requires extra message types:

- D (sc)[#] Structural change request. This message is raised by an atomic model when the structural change conditions are satisfied. D (sc) brings the expected structural value to the parent coordinator. The structural value indicates the expected model structure of a structure component.
- *(sc)[#] Structural change message. This message is issued by the Root and passed down to each structure component. The structure agent of each structure component conducts the structural changes according to the structural value in the message.
- St Simulation resuming message. This message is sent by a structure component to the new models after the structural changes. This message is used to synchronize the models.

[#] The 'sc' in the parenthesis denotes the expected structural value in the message.

Structure Component and Structure Agent

Barros[Bar 1997] defines *dynamic structure system network* as a component that can change its structure dynamically. The dynamic structure system network is defined with a special component, the *network executive.* Since the network coupling information is located in the state of the executive, transition function can change this state and, in consequence, change the structure of the network. In FDSDE, a dynamic structure system

network, which is called *structure component*, is a coupled model subject to undergoing structural changes. The concept of network executive is represented by *structure agent*, in which the structure information of a structure component is located. The concepts of structure component and structure agent are illustrated.

Structure Component

In FDSDE, atomic models hold only model behaviours (Internal / External / Confluent transition function & output transition function) and no structure information is included; therefore atomic models are structure units and cannot be split in terms of structure. Instead, coupled models give a well-defined concept of system modularity and component couplings. That is to say, coupled models contain structure information. As a result, a structure component can be represented by a series of model structure sets including modules and couplings between the modules. If a model structure set is taken as a structure state of a structure component and connects to a structural change command (Scomm), the structure component can shift its model structure among the model structure sets according to the indicated structural value. The Fig. 1 demonstrates the relationship between the model structure sets and the structural states of a structure component.

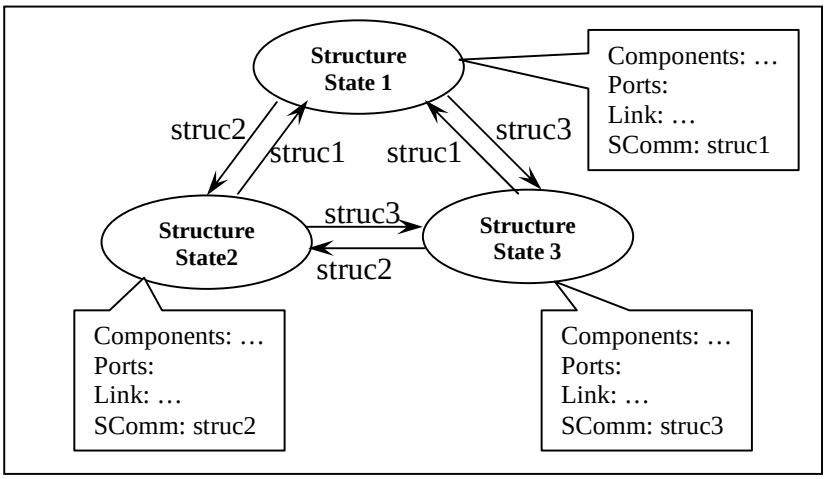


Fig 1. The Relationship between structure states and structure definitions of a structure component

Structure Agent

A structure agent defines possible structural states of a structure component and executes structural changes in its internal transition function for a structure component. As we have described, a structure component defines a series of model structures containing a group of modules and the couplings among the modules. Structure agent is employed to achieve the separated concerns between the model structure definitions and the structural change executions. Structure agent offers more flexibility to modellers who can generate the structural behaviours according to the real requirements.

Model Hierarchy and Processor Hierarchy

According to the DEVS theory, models are specified independently from the simulation mechanism. Two levels of hierarchies are presented in the DEVS-based simulation environment: model hierarchy and processor hierarchy. The DEVS model property, Closure under Coupling, carries the hierarchical nature of the models. A model of structure agent brought by dynamic structure is a leaf model of a structure component in the model

hierarchy. A general view of the model hierarchy is presented in Fig. 2. TOP is a structure component with a structure agent CEXEC. TOP is located at higher model hierarchical level than Coupled2.

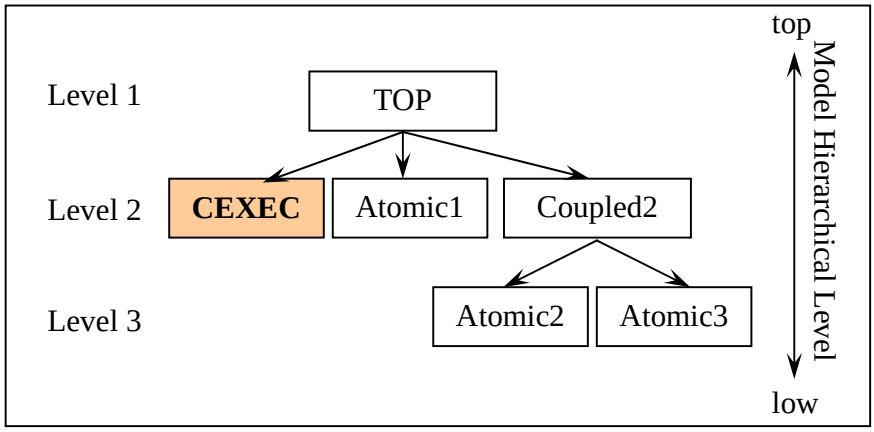


Fig 2. The Model Hierarchy

The straightforward processor hierarchy (Fig. 3) contains the similar structure with the corresponding model hierarchy. Root Coordinator is a global simulation governor standing at the top of the processor hierarchy.

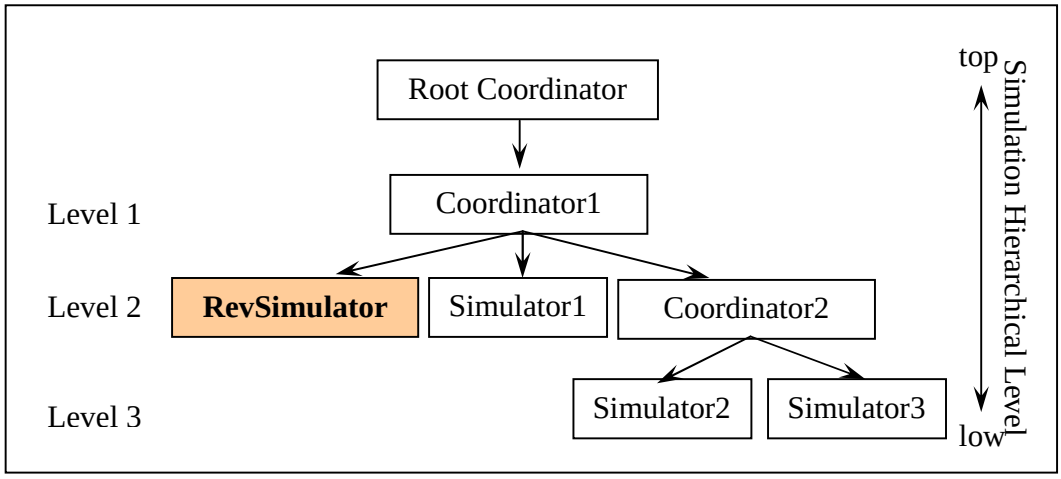


Fig 3. The Processor Hierarchy

Although the hierarchical processor structure reflects the nature of the DEVS model hierarchy, it performs ineffectively with the deeper model hierarchical complexity for communication overheads are unavoidably increased. A flat coordinator technique is a more effective processor hierarchy by eliminating the coordinators in the hierarchy and by making direct messaging communications between the flat coordinator and the simulators (Herny's thesis). The flattened processor hierarchy is shown in Fig. 4

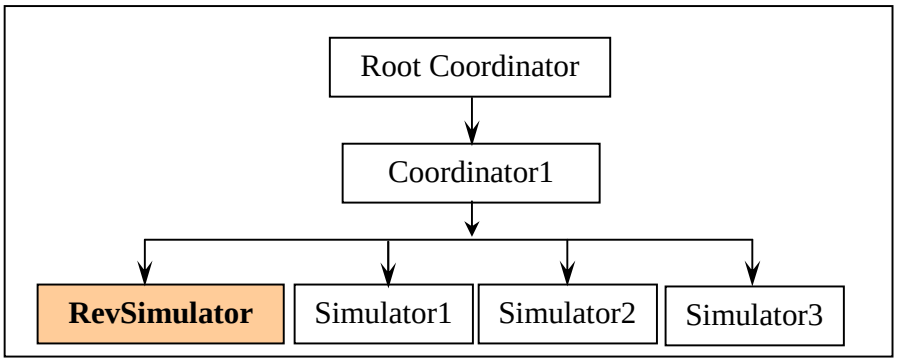


Fig 4. Flattened Processor Hierarchy

Message-Processing Algorithms

Message-processing algorithm defines a series of receive functions for each message type in each abstract simulator. Four kinds of abstract simulators namely Root Coordinator, Coordinator, Simulator and RevSimulator are used in DS-eCD++. Root Coordinator, Coordinator and Simulator which are used in eCD++ are amended to adapt to both the dynamic structural changes and the regular simulation. RevSimulator, a new kind of abstract simulator, is devised for structure agents. In the following, the message-processing mechanism of each abstract simulator is described.

Simulator

The simulator is capable of processing initial message (I), collect message (@), internal message (*), and external message (X). In DS-eCD++, no changes are made in the receive functions for collect message and external message. A structural change variable is initialized in the initial message. The structural change request detection mechanism is applied to the internal message.

```
1   When a (I, 0) is received from the parent Coordinator
2           tL = 0; tN = Infinity
3           Initialize the variable of the atomic model
4           struc_var = 0 // struc_var : structural change variable
5           send (D, tN) to the parent Coordinator
6   end when
```

Fig 5. Simulator Algorithm for (I, 0)

The simulator receives (I, 0) at the beginning of the simulation (Fig. 5). Two timing variables, tL and tN, are initialized in (I, 0). The associated atomic model is initialized by calling initial function (line 3). Moreover, the structural change variable which is used to keep track of the structural state of a structure component is initialized in the following. Finally, a (D, tN) is sent to its parent Coordinator with tN, absolute next time, indicating the time for next state transition of the atomic model.

```
1.  When a (@, t) is received from the parent Coordinator
2.          if t = tN then
3.              y = λ(s)
4.              send (y, t) to the parent Coordinator
5.              send (D, t) to the parent Coordinator
6.          end if
7.          else raise error
8.  end when
```

Fig 6. Simulator Algorithm for (@, t)

```
1.  When a (q, t) is received from the parent Coordinator
2.          lock the bag
3.          Add event q to the bag
4.          unlock the bag
5.  end when
```

Fig 7. Simulator Algorithm for (q, t)

Fig. 6 is the simulator algorithm for (@, t). If the scheduled time tN arrives (t = tN), the simulator executes output function (λ) of the atomic model and sends the output to the parent coordinator upon receiving (@, t). A (done, t) is sent to the parent coordinator indicating the completion of the execution. If a (q, t) arrives (Fig. 7), the Simulator add the input event to its message bag.

```
1.  When a (*, t) is received from the parent Coordinator
2.          case tL ≤ t < tN and bag is not empty
3.                  e = t – tL
4.                  s = δext(s, e, bag)
5.                  if no structural change request is raised
6.                      empty the bag
7.                  end if
8.          end case
9.          case t = tN and bag is empty
10.                 s =δint (s)
11.                 tL = t
12.                 tN = tL + ta(s)
```

```
13.              end case
14.              case t = tN and bag is not empty
15.                       s =δcon (s, bag)
16.                       if no structural change request is raised
17.                            empty the bag
18.                       tL = t
19.                       tN = tL + ta(s)
20.              end case
21.              case t > tL or t < tN
22.                       raise error
23.               end case
24.               if structural change request is raised
25.                    struc_var = sc
26.                    send (D, tN, sc) to the parent Coordinator
27.                    // sc: new structural value
28.               else send (D, tN) to the parent Coordinator
29.               end if
30.  end when
```

Fig 8. Simulator Algorithm for (*, t)

Simulator uses one (*, t) message to synchronize three different transition functions (internal transition function, external transition function and confluent transition function) of the atomic model. The one of the three transition functions is executed according to the status of the message bag in the atomic model and the timing point when the message is received (Fig. 8). The external messages in the message bag would not be consumed if a structural change request is raised. Therefore, the external messages in the message bag would not be removed. If a structural change request is raised, the new structural value indicating the expected model structure is sent to the parent coordinator. The tN in the

structural change request indicates the expected structural change time. Otherwise, a (D, tN) is sent indicating the completion of the internal message.

```
1. When a (St, t) is received from parent coordinator
2.         reset the structural change variable
3.         initialize the variable for the atomic model
4.         send (D, tN) to parent coordinator
5. end when
```

Fig 9. Simulator Algorithm for (St, t)

If the structural change causes the addition of models, St messages are received (Fig. 9) in the new models. The structural change variable is reset and the variable used in the atomic model are initialized in (St, t). Scheduled tN is sent out with the D message to the parent coordinator for the next simulation cycle.

RevSimulator

RevSimulator defines the message-passing mechanism for structure agents. A structure agent would not receive content messages for it is absent from input / output ports. Moreover, since a structure agent receives a structural change message passively and stays at the structural state until next structural state is indicated, it would not be an imminent child of the associated structure component. Hence, a structure agent does not receive a collect message. A structure agent would not be a receiver of a St message. A structure agent is a receiver of an initial message and a structural change message. Initial message is used to initialize a structure agent at the beginning of simulation. Structural change

messages bring the expected structural values to a structure agent and indicate it to conduct the structural changes for the structure component.

During the initialization stage, RevSimulator (Fig. 10) sets the tN as infinity and initializes the structure agent by invoking its initial function. RevSimulator notifies the completion of initialization by sending a (D, t) to the parent coordinator.

When a structural change message arrives at the RevSimulator (Fig. 11), the timing period is checked first. The internal transition function of the corresponding structure agent is executed. After that, the (D, t) is sent out to the parent coordinator.

```
1.  When receive a (I, 0) from parent coordinator
2.          tN = inf
3.          Initialize Structure Agent by calling the initfunction
4.          send (D, t) to parent coordinator
5.  end when
```

Fig 10. RevSimulator Algorithm for (I, 0)

```
1.  When receive a (*, t) (sc) from parent coordinator
2.          if t < tL or t > tN then raise error
3.          else if (message value is not 0) then
4.                  tN = inf
5.                  tL = t
6.                  invoking the internal function of the structure agent
7.                  send (D, t) to parent coordinator
8.           end if
9.  end when
```

Fig 11. RevSimulator Algorithm for (*, t) (sc)

Coordinator

The Coordinator is in charge of the messages between the parent coordinators and the child simulators. The coordinator is able to process the following messages:

- @ message from the parent coordinator
- Y message from the child simulator
- Q message from the parent coordinator
- * message from the parent coordinator
- *(sc) message from the parent coordinator
- D message from the child simulator
- D(sc) message from the child simulator

@, Y and Q messages follow the mechanisms used in eCD++. A *(sc) is delivered by a * message and distinguishes itself by a non-zero structural value (sc). Similarly, a structural change request appends an expected structural value to a D message.

```
1.  When a (@, t) is received from the parent Coordinator
2.        if t = tN then
3.           tL = t
4.           for all imminent child processors i with minimum tN
5.                 send (@, t) to child i
6.                 cache i in the synchronize set
7.           end for
8.           wait until (D, t)'s are received from all imminent processors
9.           send (D, t) to the parent Coordinator
10.       else raise an error
11.       end if
12. end when
```

Fig 12. Coordinator Algorithm for (@, t)

Fig. 12 shows how a @ message is processed in coordinator. The time stamp is checked first. If the time stamp is not equal to tN, an error is raised. Only those models that are at their state transitioning points will receive the (@, t) message. The coordinator dispatches the @ message to all its imminent children and sends the receivers to the synchronization set. A D message is sent to its parent coordinator implying the completion of the collect phase in the coordinator after all D's have been received from the imminent children.

```
1.  When a (y, t) is received from child i
2.          for all influences, j of child i
3.                  q = z_{i,j} (y)
4.                  send (q, t) to child j
5.                  cache j in the synchronize set
6.          end for
7.          wait until all (D, t)'s are received from j's
8.          if self ∈I_i (y is to be transmitted upward) then
9.                  y = z_{i, self} (y)
10.                 send (y, t) to the parent Coordinator
11.         end if
12. end when
```

Fig 13. Coordinator Algorithm for (y, t)

Coordinator is responsible for dispatching Y messages to the all influences of the output messages. Upon receiving (y, t) (Fig. 13), the coordinator translates the output message into the external messages for all the child influences and sends them to the corresponding children. The child influences are cached into the synchronization set, in which the models are expected to experience state transitions at the next simulation cycle. If the coordinator

is one of the receivers of the output message, a proper output is generated for the coordinator and is forwarded upward to its parent coordinator.

The incoming external message (Fig. 14) is inserted into the equipped message bag for later calculation during an internal message processing.

```
1.  When a (q, t) is received from parent Coordinator
2.          lock the bag
3.          Add event q to the bag
4.          unlock the bag
5.  end when
```

Fig 14. Coordinator Algorithm for (q, t)

Coordinator is capable of processing * and * (sc) messages through a receive function for (*, t) (Fig. 15). * (sc) contains a non-zero value while a zero value indicates an internal message. * message is received in between the tL (the last transition time) and the tN (the next scheduled transition time) of the coordinator. Otherwise, an error is raised.

If the received message contains a non-zero value, the message is a structural change message. In FDSDE, structural changes are executed from bottom to up. That is to say, the structural change message is executed in the structure components standing at the lower model hierarchical level first, and then it is implemented in the structure components at the higher model hierarchical level. Coordinator handles the executing order with the depth first policy. The structural change message is passed to the child coordinators provided the coupled models associated with the child coordinators are structure components. The child coordinators are collected into a structure set first (line4 – line10), and then they get the copies of the structural change messages (line12 – line16). At the same time, the structural

value is stored (line11) and is used for the structural change in this coordinator. If no such a child coordinator exists, the coordinator passes the structural change message to the simulation processor of the structure agent (line17 – line21). Upon the D messages are received, the coordinator sends a D message to the parent coordinator to complete the structural changes process.

A zero-valued message indicates an internal message. Firstly, the external events in the message bag are routed to the corresponding components according to the coupling information preserved by the coupled model associated with the coordinator. The receiving components are cached into the synchronization set. Then a * message is sent to the components in the synchronization set. Until all D's are received from the models in the synchronization set, the updated tN with a D message is sent to the parent coordinator.

```
1.  When a (*, t) is received from the parent Coordinator
2.      if tL≤t≤tN then
3.        if the message value is a non-zero value // Structural change message
4.        for all the child i
5.            if i is a coupled model and i is a structural component
6.                if i is not in the structure set
7.                    cache i in the structure set
8.                end if
9.            end if
10.       end for
11.       store the message value // for self structural change
12.       if structure set is not empty
13.           for all j in the structure set
14.               send (*, t) (sc) to j
15.           end for
16.       end if
17.       else if structure set is empty
18.           if the associated coupled model is a structural component
19.               send (*, t) to the structure agent
20.           end if
21.       else end
22.       wait until all (D, t)'s are received
23.       send (D, t) to parent coordinator
24.     end if
```

```
24.           end if
```

```
25.       else if the message value is a zero value // Regular (*, t) message
26.             for all q ∈bag
27.                   for all receivers, j ∈I_self and all q ∈bag
28.                         q = z_self, j(q)
29.                          send (q, t) to j
30.                         cache j in the synchronize set
31.                   end for
32.                   empty bag
32.                   wait until all (D, t)'s are received
33.                   for all i in the synchronize set
34.                         send (*, t) to i
35.                   end for
36.                   wait until all (D, tN)'s are received
37.                   tL = t
38.                   tN = minimum of components' tN's
39.                   clear the synchronize set
40.                   send (D, t) to parent coordinator
41.             end else-if
42.       else raise an error
43.  end when
```

Fig 15. Figure 17 Coordinator Algorithm for (*, t)

Coordinator receives and processes D messages according to the different waiting modes (Fig. 18). The six waiting modes are set in coordinator. In the case of waiting for initialization message, coordinator simply picks the minimum tN and sends (D, tN) to the parent coordinator. Coordinator collects all D's from the children and sends D to the parent coordinator when it is in the waitingforCollect mode. When a D message is received in the mode of waitingforInternal, the procId is firstly removed from the synchronization set indicating the ending of the synchronization stage. If a D (sc) message is received, the procId and the message value of the sender are cached into **screq** (line18 – line20)**.** The coordinator determines whether a D (sc) or a D is sent to the parent coordinator depending on which sender(s) is (are) an imminent child. If the sender of the request pair (procId, value) is not an imminent child, a D message is sent to the parent coordinator. Otherwise, a D (sc) is sent to the parent coordinator when the sender in **screq** is also an imminent child (line25 – line28). When the coordinator receives a D in the mode of waitingforStructuralchange which implies a nested structural change process and the coordinator is waiting for structural change done messages from its children, the coordinator sends * (sc) to its structure agent to trigger the structural change if the associated coupled model is a structure component (line27 – line39). Otherwise, a D is sent to the parent coordinator (line40 – line43). The mode of waitingforSelfstructurechange indicates the coordinator is waiting for a D message from its structure agent. In this case, the new atomic models added to the structural change process are sent St messages (lien46 – line50), and the atomic models to be removed in the structural change process are deleted from the synchronization set of the coordinator (line51 – line55). After that the coordinator sends D with the minimum tN to the parent coordinator. The mode of waitingforStart is

waiting for the responses to the St messages sent to the new created models. At this stage, the minimum tN is updated and a D is sent to parent coordinator (line62 – line63).

```
1.  When a (D, t) is received from child simulators
2.          case waiting mode is waitingforInit
3.                  wait for all (D, t)'s are received from child simulators
4.                   tL = t
5.                   get the imminent children's tN
6.                   send (D, tN) to the parent coordinator
7.                end if
8.          end case
9.          case waiting mode is waitingforCollect
10.                 remove the procId from the collectWaitForDoneQ
11.                 if collectWaitForDoneQ is empty // all D's are received
12.                     send   (D, t) to the parent coordinator
13.                 end if
14.         end case
15.         case waiting mode is waitingforInternal
16.                 tL = t
17.                 remove the procId from the syncSet
18.                 if message value != 0
19.                      add the (procId, value) pair into screq
20.               end if
21.               wait for all (D, t)'s are received
22.               get the imminent children's tN
23.               if screq is not empty
24.                    get the procId from screq
25.                    if the procId is in the immiChildren
26.             sc = value
27.               send (D, tN) (sc) to parent Coordinator
28.                    end if
29.                    clear screq
30.               end if
31.               else
```

```
45.                 case waiting mode is waitingforSelfstructuralchange
46.                         if there are newly added models
47.                             for each new model i
48.                                     send (St, t) to i
49.                             end for
50.                         end if
51.                         if removedmodels is not empty // the models to be deleted
52.                             for each model i in removedmodels
53.                                 erase i from the synchronization set
54.                             end for
55.                         end if
56.                         else
57.                                 get tN's from the imminent children
58.                                 send (D, tN) to parent coordinator
59.                          end else
60.                 end case
61.                 case waiting mode is waitingforStart
62.                         get tN's from the imminent children
63.                         send (D, tN) to parent coordinator
64.                 end case
65.  end when
```

Fig 16. Coordinator Algorithm for (D, t)

Root Coordinator

Root Coordinator is a global scheduler standing at the top of the processor hierarchy. Root Coordinator executes a message loop and advances the simulation according to the timing pieces collected from the child processor. If there is no scheduled timing pieces (i.e. the tN is equal to infinity), Root Coordinator ends the simulation. The Root Coordinator algorithm is shown in Fig. 19. At first, the structural change request variable is initialized as 0. During the message loop, Root Coordinator collects the outputs and routes the outputs to the proper influences by issuing a @ message. If there is no structural change request is raised, Root Coordinator sends * message to trigger the state transitions of the models. The structural change process is invoked by a * (sc) if a structural change request is detected. The structural change request variable indicates the expected structure set and is assigned to the structural change variable in the * (sc).

```
1.  t = tN of the topmost coordinator
2.  sc = request // if a structural change request is raised, request = 1; otherwise request
    = 0
3.  while t≠∞ or more external events to come
4.          send (@, t) to the topmost coordinator
5.          wait until (D, t) is received from it
6.           if sc == 1
7.                  send (*, t) (sc) to the topmost coordinator
8.                  wait until (D, tN) is received from it
9.           end if
10.          else if sc == 0
11.                 send (*, t) to the topmost coordinator
12.                 wait until (D, tN) is received from it
13.          end else
14.          t = tN of the topmost coordinator
15.          if an external event arrive
16.                 send (q, t) to the topmost coordinator
17.          end if
18.  end while
19.  raise simulation completed
```

Fig 17. Root Coordinator Algorithm

Message-Passing Scenarios

The previous section illustrates the message-processing algorithms of each abstract simulator in FDSDE. In this section, typical message-passing scenarios are presented to present how the messages flow between the simulation processors. The first scenario presents the simulation process with a structural change process. The second scenario exhibits a nested structural change process. For each scenario, the model structure and the processor hierarchy are presented first, and then the message-passing scenarion of the given model is described using an *event precedent graph*. The vertexes (the black dots) are used to indicate the events, and the directed edges present the actions of sending messages. The message types are placed beside the directed edges. The subscript numbers of the message types give the sequence of the messages during the simulation. In the scenarios, RC denotes the Root Coordinator; Sx (x = 1, 2, 3,…) represent the simulators in charge of message processing of the atomic models. Cx (x = 1, 2,…) indicate coordinators; RSx (x = 1, 2,…) are the processors of the structure agents.

Scenario 1: A Simulation with a Structural Change Process

In this scenario, the structure component Coup1 contains two atomic models: A1 and A2 at the initial simulation stage. The structural change in Coup1 adds an atomic model A3 to the simulation system. C1 is the coordinator associated with Coup1. S1, S2 and S3 are the three simulators generating the model behaviours of A1, A2 and A3 respectively. The model structure and the processor hierarchy of the scenario 1 are shown in Fig. 18 and Fig. 19.

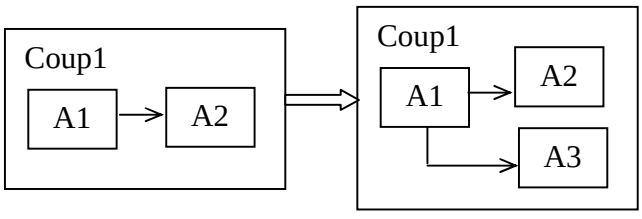


Fig 18. The model Structure Change in Coup1

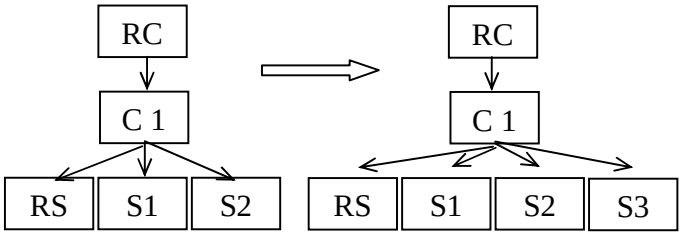


Fig 19. The Simulation Hierarchy Change in Scenario 1

Initially, I_1, I_2 and I_3 are used to initialize C, S1 and S2. As responses, D_4, D_5 and D_6 are replied by them. We skip the regular simulation cycles for they are same with scenario 1. At the simulation cycle T_i, RC sends $*_{i+1}$ to C and C sends $*_{i+2}$ to S1 for S1 is the only model that needs to be synchronized at this cycle. Assume S1 raises the structural change at this cycle. A D_{i+3} (sc) is sent back to C. C passes the structural change request to RC with D_{i+4} (sc). Instead of issuing @ message to enter collecting phase, RC issues structural change message $*_{i+5}$ (sc) to C and C delivers the structural change message to the processor of its structure agent RS with $*_{i+6}$ (sc). A message D_{i+7} is replied by RS after the structural changes finishes. Suppose a new simulator processor S3, corresponding to the atomic model A3, is added into the simulation. C initializes S3 by sending St_{i+8}. S3 replies tN to C with D_{i+9}. Till this point, C has finished the structural change process and collected the next event times from its children. C selects the minimum tN and sends to RC with D_{i+10} notifying the completion of the structural change process. In the structural change process, S3 joins the simulation. The simulation advances and RC sends $@_{i+11}$ to collect outputs at the

simulation cycle T_{i+1}. Suppose S1 is the receiver of the collect message $@_{i+12}$. S1 returns $Y_{i+13,}$ the output message of S1, and $D_{i+14,}$ the done message of S1, to C. Since S2 and S3 are the influences of S1, C converts the output message of S1 into the proper input messages and routed them to S2 (X_{i+15}) and S3 (X_{i+16}) respectively. As a result, S1, S2 and S3 are all cached into the synchronized set of C and C sends D_{i+17} to RC marking the ending of the collect phase of T_{i+1}. At the transition phase of T_{i+1}, synchronizing messages are spread to each model and trigger the transition functions ($*_{i+18}$, $*_{i+19}$, $*_{i+20}$, $*_{i+21}$). Accordingly, tNs are returned to RC with the done messages (D_{i+22}, D_{i+23}, D_{i+24}, D_{i+25}). Then RC sends a collect message ($@_{i+26}$) to start the new simulation cycle T_{i+2}.

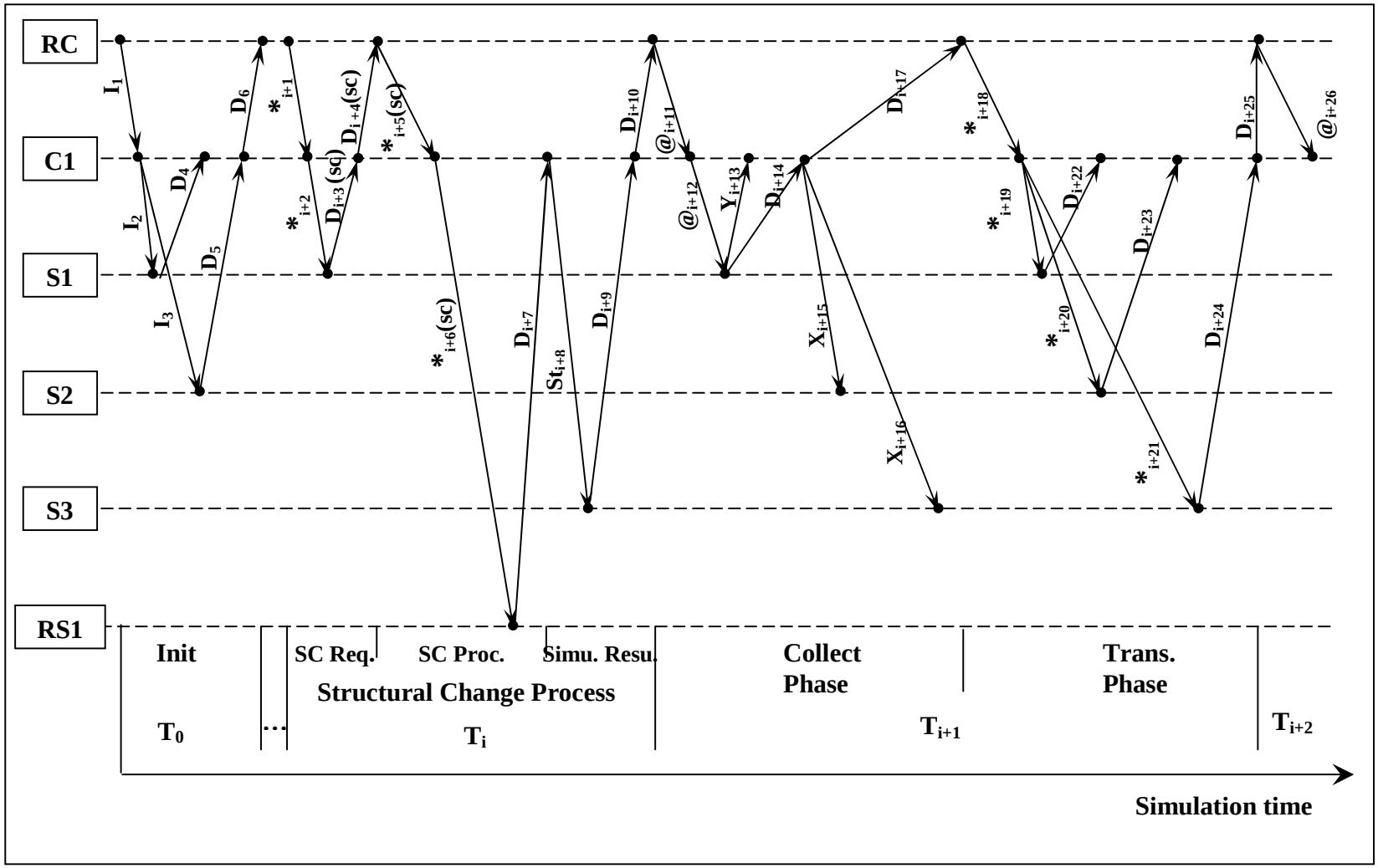


Fig 20. Message flow for a Structural Change Process

Scenario 2: A Nested Structural Change Process

In scenario 1, single structural change process responds to a structural change request. Sometimes, a nested structural change process involving more than one structure components in the model hierarchy is required to respond to a structural change request. In case of a nested structural change process, an executing priority of structure components should be determined. In FDSDE, nested structural change process is conducted from bottom to up. That is, the structure component at lower model hierarchical level executes the structural change first, and then one at the higher model hierarchical level executes the structural change. Scenario 2 presents a nested structural change process.

Scenario 2 involves two structure components Coup1 and Coup2. Coup1 governs Coup2 and an atomic model A1. Coup2 contains an atomic model A2. The structural change adds an atomic model A3 to Coup2 and adds a new coupling between A1 and Coup2. S1, S2 and S3 represent the simulators for A1, A2 and A3 respectively. C1 and C2 are the two coordinators of Coup1 and Coup2. Coup1 has a structure agent SA1 and Coup2 contains a structure agent SA2. RS1 and RS2 are the processors of the structure agents SA1 and SA2. The model structure change and the simulation hierarchy change in the scenario 2 are exhibited in Fig. 21 and Fig. 22.

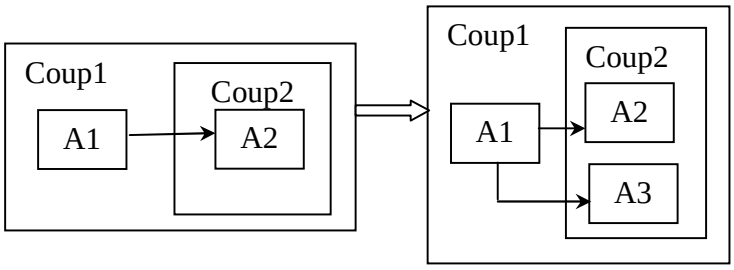


Fig 21. The Model Structure Change in Coup1 and Coup2

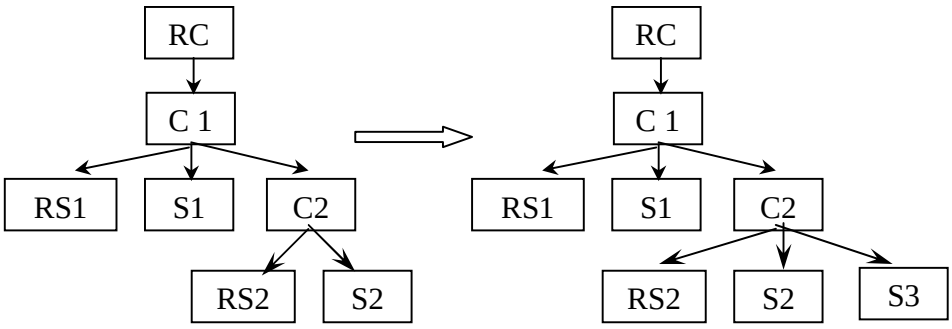


Fig 22. The Simulation Hierarchy Change in Scenario2

At the beginning of the simulation, all the processors participating the simulation are initialized (I_1, I_2, I_3, I_4) and the D messages are replied (D_5, D_6, D_7, D_8). At simulation cycle T_i, S2 raises a structural change request during the transition phase ($*_{i+1}$, $*_{i+2}$, $*_{i+3}$, D_{i+4} (sc), D_{i+5} (sc), D_{i+6} (sc)). Once RC receives a structural change request, RC starts a structural change process. Suppose it is a nested structural change process involving the structure component Coup1 and Coup2. When C1 receives $*_{i+7}$ (sc) sent from RC, C1 passes ($*_{i+8}$ (sc)) to its child C2. C2 is the structure component at the lowest hierarchical level; therefore the structural change of C2 is executed first. C2 sends $*_{i+9}$ (sc) to RS2 to execute the structural changes. RS2 returns D_{i+10} to C2 when the structural change has been done. Assume that a new model is added during the structural change. C2 sends St_{i+11} to S3, a new processor of the atomic model A3, to initialize it. When D_{i+12} is returned to C2, C2 then return D_{i+13} to C1 notifying the structural change has finished at the structure component Coup2. Once receiving the structural change done message from C2, C1 sends $*_{i+14}$ (sc) to RS1 to start a structural change. RS1 replies D_{i+15} to C1 and C1 passes D_{i+16} to RC marking the ends of the structural change at C1. According to the minimum tN, the simulation advances to simulation cycle T_{i+1}. Suppose S3 is not an imminent child of C2.

The collect phase of T_{i+1} is processed ($@_{i+17}$, $@_{i+18}$, $@_{i+19}$, D_{i+20}, D_{i+21}, D_{i+22}). The following transition phase ($*_{i+23}$, $*_{i+24}$, $*_{i+25}$, D_{i+26}, D_{i+27}, D_{i+28}) implements the transition functions of S2, which are bypassed during simulation cycle T_i. At simulation cycle T_{i+1}, RC obtains an updated tN and advances the simulation to T_{i+2} (T_{i+1} + tN). RC sends $@_{i+29}$ to collect outputs at simulation cycle T_{i+2}.

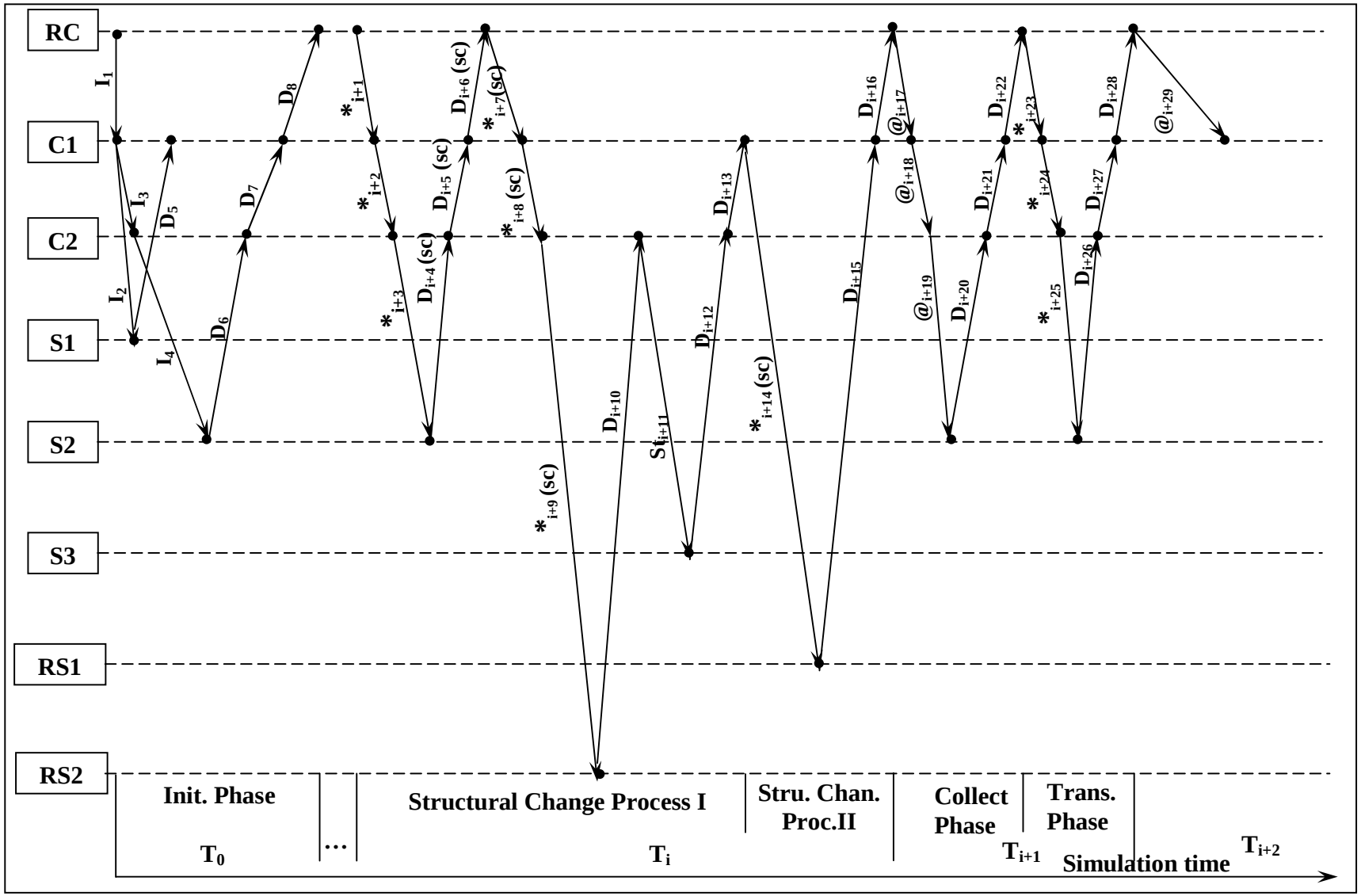

RC
C1
C2
S1
S2
S3
RS1
RS2
Init. Phase
Structural Change Process I
Stru. Chan. Proc.II
Collect Phase
Trans. Phase
T0
Ti
Ti+1
Ti+2
Simulation time

Fig 23. Message flow for Multiple Level Structural Change Process

Simulation Phases

In the message-passing scenarios, the simulation phases are clearly identified. In the regular simulation process, two phases are distinguished by @ message and * message respectively. @ messages dispatched to each processor and the corresponding D messages are compose of collect phase. Transition phase consists of * messages routed down and the returned D messages.

Structural change process is included in the regular simulation process. Three sub-phases are identified in a structural change process: structural change request, structural change and simulation resuming. Each sub-phase is marked by a message by which the sub-phase is invoked. If a (done, tN) (sc) is return to respond to a * message in a transition phase, the transition phase is identified as a structural change request sub-phase. The real transition phase is bypassed due to the raise of the structural change request. Root Coordinator initiates a structural change sub-phase by issuing (*, t) (sc) message upon receiving a structural change request. D messages are sent back indicating the ends of the structural change sub-phase. A coordinator associated with a structure component starts a simulation resuming sub-phase by sending (St, t) message to the newly added models. The newly added models return the scheduled tNs, which are used to schedule the next imminent simulation event and signals the finishing of the simulation resuming sub-phase.

Chapter 4 Algorithm Implementation and the Functionalities

An improved simulation engine is developed by combining FDSDE and P-DEVS real time simulation engine. Based on the improved simulation engine, the updated real-time DEVS-based experimental environment is constructed to support the dynamic structure real-time simulation and the real-time embedded system design. The new software is called DS-eCD++. The software architecture of DS-eCD++ adopts the four software components in eCD++. Various aspects of implementation differ from that in eCD++. Introduction of structure component and structure agent requires extra classes to represent the components. A structure component involves multiple structure options, but only one structure, which is an active structure of a structure component, participates in the simulation. The structure shifts call for cooperation among the components and the processors in the simulation. Also, the structure shifts brings a series of changes in the software architecture of eCD++. This chapter will discuss the implementing issues in DS-eCD++. The summary of the revisions are explained in section 4.1. The sections from 4.2 to 4.5 depict the implementing details of each software component in DS-eCD++. Finally, the functionalities of DS-eCD++ are discussed at the section 4.6.

Software Architecture Overview

In eCD++, the four software components contain well-defined behaviours and corporate with each other to execute real-time DEVS simulations. The following figure depicts the major parts of each software components and the relationships among them.

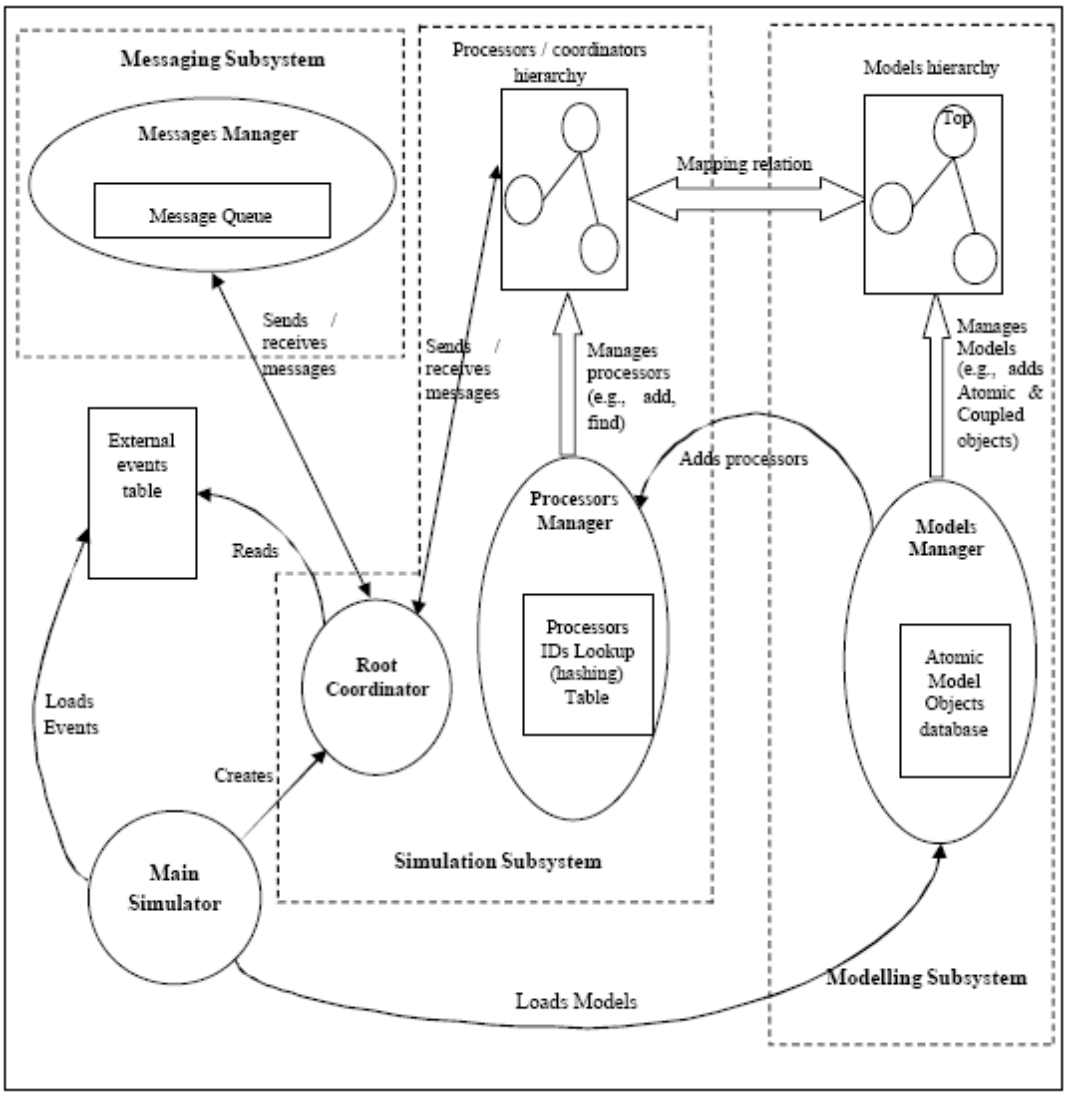


Fig 24. eCD++ Software Architecture

DS-eCD++ maintains the four software components; however, revisions have been made to fit the new features. The revisions of the four software components are characterized as:

1. The Main Simulator assumes the responsibility for loading coupled models, atomic models and structure agents. It takes charge of separating the model definition into two groups: the active components and the structure

components. It loads the initial model hierarchy at the initial stage of the simulation. The model hierarchy is updated through the structure agents of structure components as required during the simulation. In a simulation using the Flat Coordinator, Main Simulator takes charge of storing the initial model composition and the couplings used in the simulation.

2. The DEVS Modeling Subsystem maintains a model hierarchy tree composed of atomic models, coupled models and structure agents. The structure agent objects database is created along with the atomic model objects database. The Simulation Subsystem, including the *Root*, the *Coordinator*, the *Simulator* and the *RevSimulator*, maintains the processor hierarchy corresponding to the model hierarchy. In the Simulation Subsystem, the receive functions for different types of messages in the processor class are redefined to implement the message-passing algorithms described in the FDSDE algorithm. *RevSimulator* is a class of abstract simulator – RevSimulator which processes only the initial messages and the structural change messages. It is a special message processor for structure agents. The *FlatDEVSCoordinator* is redefined to implement the Flat Coordinator in the simulation using a flat coordinator. The five processor classes constitute the improved simulation engine in the Simulation Subsystem in DS-eCD++ supporting the dynamic structure real-time simulation.

3. The extra messages related to structural changes cause the expansion of the Messaging Subsystem. *InternalMessage* class and *DoneMessage* class are

reused to convey the structural change messages and the structural change requests by appending a non-zero value in a message. A new message class *StartMessage* is created for St message.

With the revised software architecture, the high level design walk-through draws an overall picture of how dynamic structure works in DS-eCD++.

1. Main Simulator separates the model definition into two groups: the initial model structure in the active component container and the structure components in the structure components container. During the initial model structure are loaded into simulation system, Main Simulator constructs the model hierarchy assisted by the Modeling Subsystem and builds the associated processor hierarchy with the help of the Simulation Subsystem. The simulation control is passed to Root once upon the simulation starts by Main Simulator. In the simulation using a flat coordinator, Main Simulator memorizes the initial flattened model structure in the simulation.
2. Once a structural change request is raised by an atomic model, the structural change request with the requested structural value is delivered upward to the Root provided the request is imminent among the simulation events. Root issues a structural change message and routes the message to all the structure components in simulation system. The structure components hand the structural change message to the

associated structure agents to process the structural changes.

3. The structure agents retrieve the expected model structure from the structure components container according to the structural values appended in the messages. The structure agents compare the two model structures. Aided by the structural change operations, the structure agents update the model structure in the simulation system. Done messages are sent to the structure components. When the done message returns to Root, the structural change process ends.
4. In the simulation with a flat coordinator, the structural change message is passed to the flat top and the flat top routes the message to its associated structure agent. The structure agent retrieves the expected model structure and compares it with the flattened model structure stored in Main Simulator. The flattened model structure in the simulation system is updated by the structure agent. The done message reaching the Root indicates the finishing of the structural change process.
5. The Root advances the simulation into the next simulation cycle and the simulation continues.

4.2. Main Simulator

Main Simulator as a subsystem includes three classes: *MainSimulator* is the very first object created during simulation and manages the overall aspects of the simulation. *Ini* is used by *MainSimulator* to parse the model definition. *MainSimulator* configures the

simulation environment through *SimLoader*. The modifications in Main Simulator are listed in the following class diagram. The tasks performed in Main Simulator are summarized as:

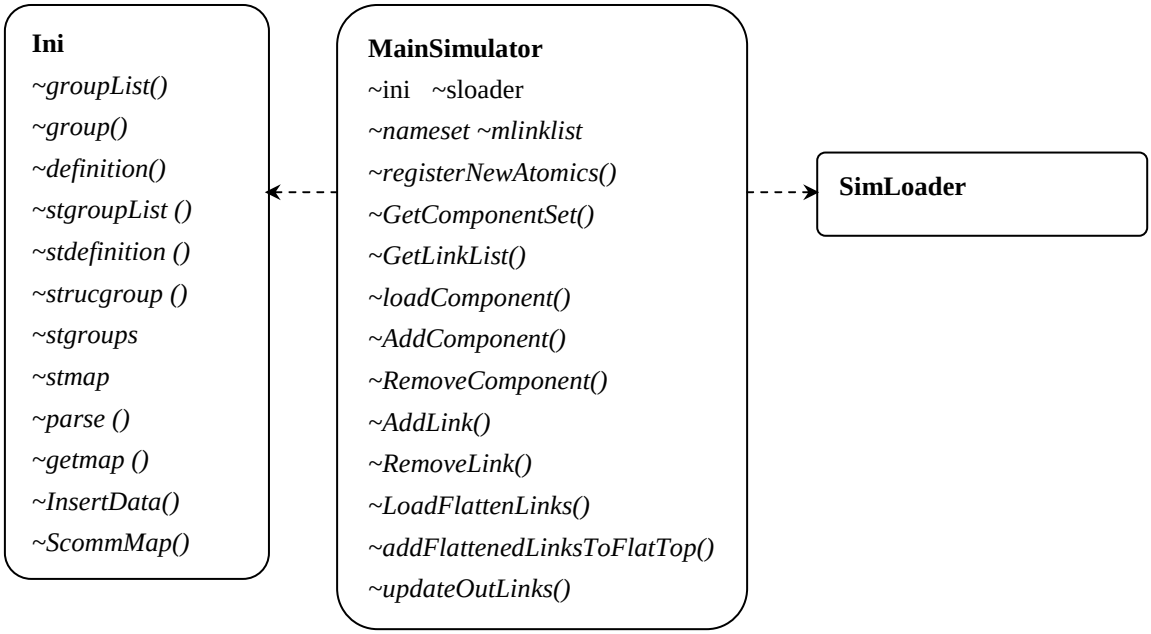


Fig 25. Main Simulator Class Diagram

1. Models registration. At the beginning of simulation, Main Simulator registers the pointers to the objects of the atomic models and the pointers to the objects of the structure agents using *registerNewAtomics()* method.
2. Models loading. The components, the couplings among the components and the input/output ports are loaded in the simulation system by parsing the model definition. Structure agents are absent from input/output ports; therefore, no ports and couplings are involved. Except for loading the coupled models and the atomic models, *loadComponents()* method loads the structure agents. The model types are identified (details in section 4.2.1 and section 4.2.2) through the different separators

in the model definition which are parsed and categorized by the operations in the class *Ini*.

3. Simulation environment configuration. The simulation parameters indicating the simulation environment are read in. Main Simulator is responsible for simulation environment configuration utilizing a simulation environment loader *SimLoader*.
4. Simulation start-up and ending. Once the preparations are ready. Main Simulator passes the control to Root Coordinator and simulation starts. Main Simulator recovers the control of the simulation and announces the termination at the end of the simulation.

4.2.1. Structure Component Description

A structure component is furnished with a structure agent to carry out the structural changes in the structure component. In DS-eCD++, new syntax is used to extend the build-in specification language provided in CD++ to describe the structure components and their alternative modeling structures.

The initial structure and the alternative structures of a structure component are explicitly described in the build-in specification language in the model definition. The initial structure of a structure component is defined as the definitions of coupled models. However, two new properties are introduced to describe a structure component. The following syntax is used:

- `modelName#className`. In the component list of a structure component, the syntax is used to appoint a structure agent. The separator ‘#’ distinguishes a

structure agent from other models.

- `Scomm`. This describes the structural command of a structure component. The structural command is associated with an option of a model structure. The atomic model raises a structural change by specifying a structural command. According to the designated structural change command, the corresponding model structure is called.

The alternative structures of a structure component are specified using separated groups in the model definition. The group name is defined as follows:

[CoupledmodelName + “update” + Index]

The CoupledmodelName presents the name of a structure component. “update” is a key word in the build-in specification language indicating an alternative structure of the structure component. Index defines the sequence of the model structure of the structure component. For example, [Topupdate01] denotes the first alternative structure of the structure component *TOP*. As in the definition of the initial structure, five properties are designated in each alternative structure of a structure component: `Component`, `Link`, `In`, `Out` and `Scomm`. The sample model definition of a structure component is presented using the build-in specification language. In Fig. 3, a block heading with a model name with a pair of square brackets is a `group` indicating the definition of a coupled model or an alternative model structure definition of a structure component. The properties of a `group` including `components`, `in`, `out`, `Scomm` and `Link`, are

the `definitions`. The text followed by the `definitions` after the clone is `identifications`.

```
[top]
components : cu@ECU motor topexec#Topexec
in : in
out : out
Scomm : topstruc1
Link : in in@cu
Link : eng_in@cu in@motor
Link : out@motor sen_out@cu
Link : out@cu out

[topupdate1]
components : cu@ECU motor
in : in
out : out
Scomm : topstruc2
Link : in in@cu
Link : eng_in@cu in@motor
Link : eng_test@cu test@motor
Link : out@motor sen_out@cu
Link : out@cu out
```

Fig 26. A Sample Definition of Structure Components

4.2.2. Structure Components Parsing and Storage

The model definition are divided into two groups and stored in two containers. One is the active component container storing the active model structure; the other is the structure components container including the model structures of the structure components. The two containers constitute a model database.

Method *parse()* takes two steps to build a model database. Firstly, the key word “update” is taken as a sign to separate alternative structures of a structure component

from the initial structure. The groups whose names contain no “update” are put to the active components container. The groups whose names contain “update” are sent to the structure components container. Secondly, *InsertData()* method copies the initial model structures of the structure components to the structure components container. The components in the active components container are loaded and participate in simulation. The structure components container provides an alternative model structures database for the structure components. The alternative model structures as the structure counterparts of the structure components are exchanged with the active model structure of the structure components. Fig. 4 shows the storage mechanism of the model definition in DS-eCD++. The method *parse()* also establishes a scomm map, in which a structure command is connected to a structural value. The scomm map is built through *Scommmap()* method and can be retrieved by the method *getmap()*. The scomm map is a structural command – structural value dictionary by which the expected model structure of the structure component is located by means of a structural value indicated by modellers.

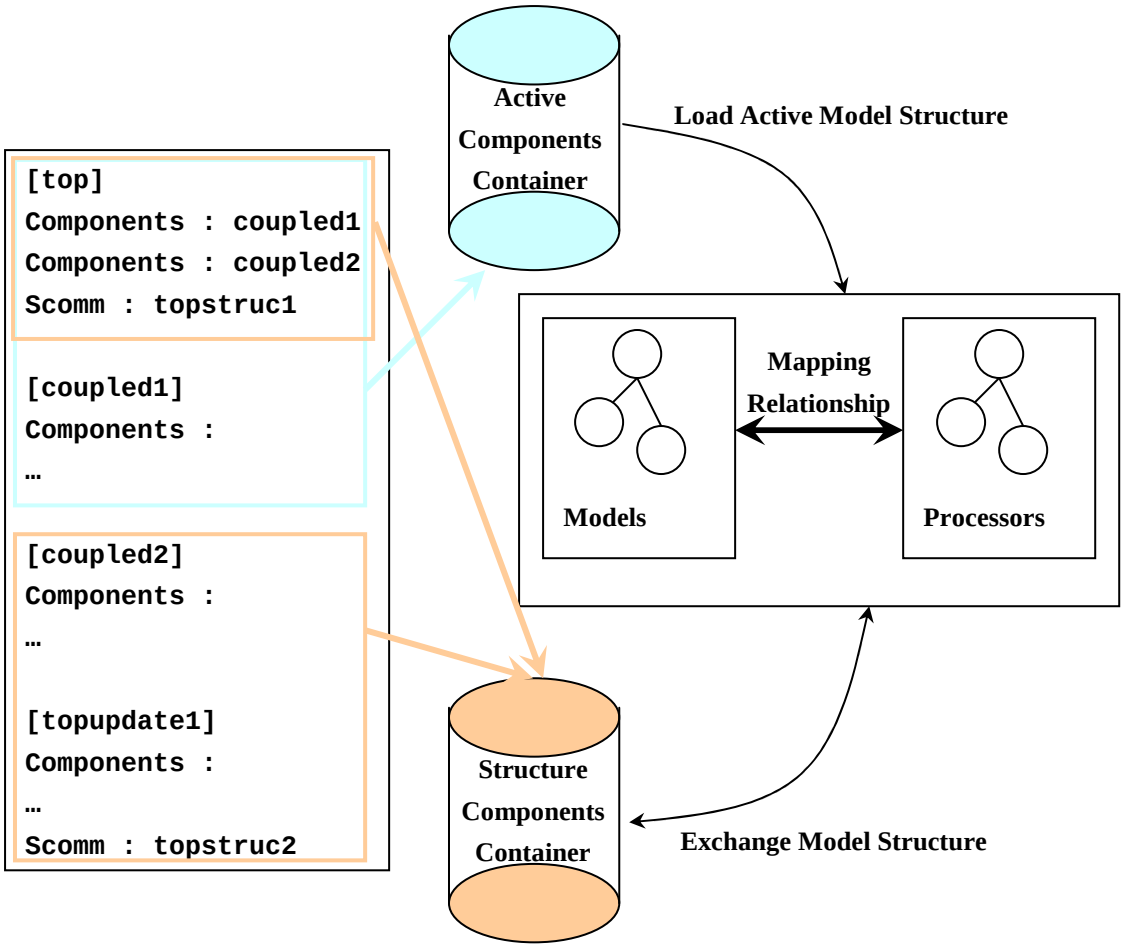


Fig 27. Model Storage and Loading

Having been stored in the two containers, the model definition is loaded into the simulation system step by step. In DS-eCD++, two groups of parsing operations are defined in the class Ini. One group of operations parse the model definition in the active component container including *groupList()*, *group()* and *definition()*. The operations are used in eCD++ to parse the model definition. The other group of parsing operations are specified for the structure components container, which are absent in eCD++. The *stgroupList ()* accesses the structure components container. *strucgroup()* method gets a whole group indicated by the group name which specifies a structural definition of a

structure component. *stdefinition ()* method retrieves the `identification` line indicated by the `definition` name.

4.2.2. The Flat Coordinator Technology

If the simulation runs with a flat coordinator, *MainSimulator* is responsible for flattening the simulation hierarchy by calling *loadFlattenLinks()* to rewire the couplings linking to coupled models directly to the far-end atomic models, calling *updateOutLinks()* to rewire any atomic models' couplings linking to coupled models directly to the far-end atomic models. Since the model structure flattening is performed after the model definition storing, the structure components container cannot obtained the flattened initial model structure. In order to backup the structure in case of recall, *MainSimulator* uses two data structures to store the flattened structure: `nameset` is a component set memorizing the components in the structure; `mlinklist` is a link list storing the couplings among the components. The two data structures are established along with the loading of the initial model structure. *AddComponent()* method adds a component to `nameset`; while *RemoveComponent()* removes a component from `nameset`. *AddLink()* method appends a new coupling to `mlinklist`. On the contrary, *RemoveLink()* method deletes a coupling from `mlinklist`. The four methods are executed with the flattening of the initial model structure. Finally, the initial flattened model structure is stored in `nameset` and `mlinklist`.

The Modeling Subsystem

The Modeling Subsystem organizes the models hierarchically. The class diagram of the Modeling Subsystem is shown in Fig. 5. The modifications of each class are presented in the class diagram but the inherited methods from eCD++ are not included.

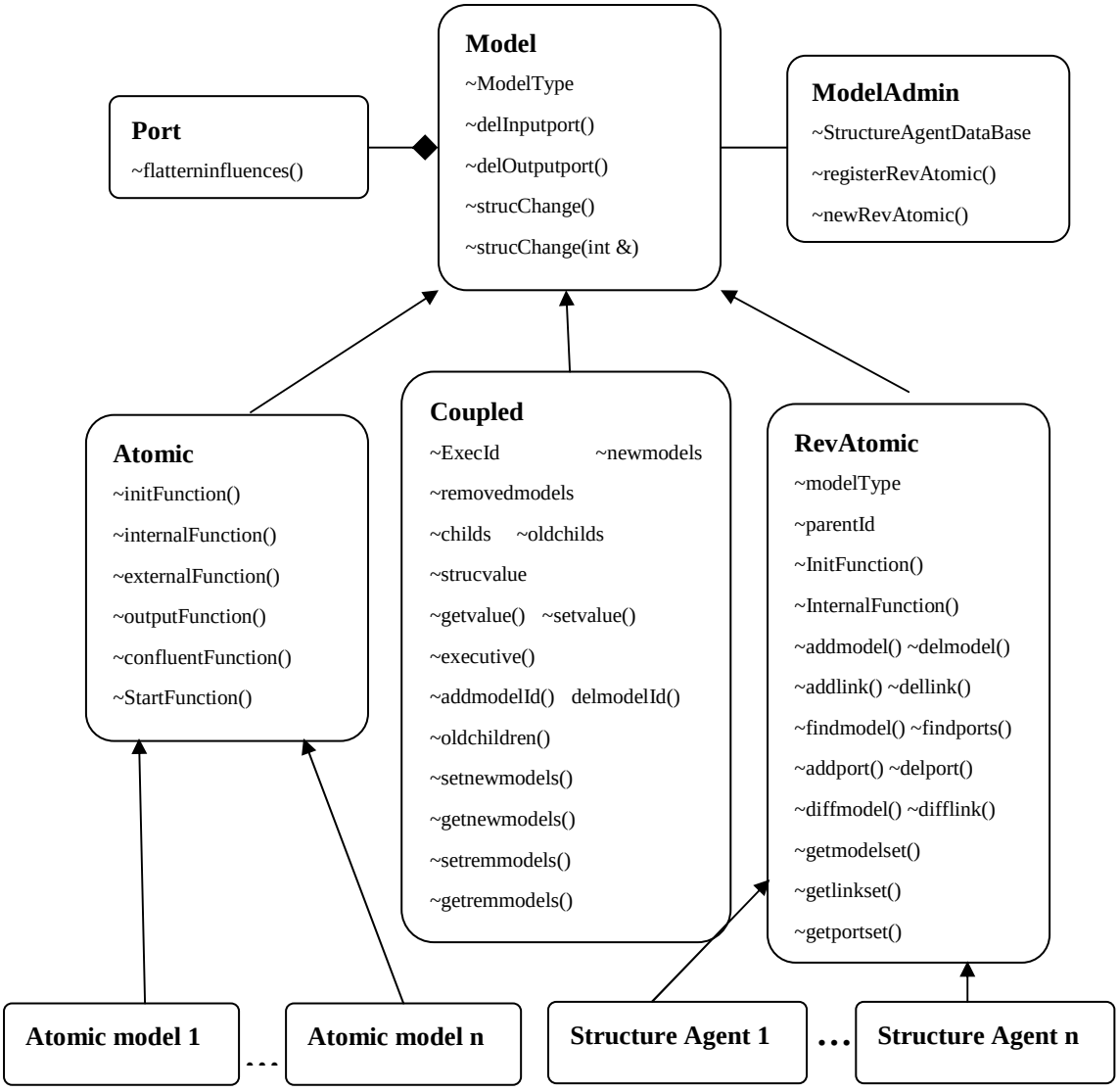


Fig 28. The Modeling Subsystem Class Diagram

The class *Model* provides the model operations theoretically. Three subclasses: *Atomic, Coupled* and *RevAtomic,* are derived from the class *Model* and encapsulate the implementations of atomic models, coupled models and structure agents. The substantial implementations of atomic models (Atomic model 1, …, Atomic model n) and structure agents (Structure agent 1, …, Structure agent n), which are derived from the virtual classes *Atomic* and *RevAtomic* respectively, are specified by modellers. Class *Coupled* possesses the model compositions and the couplings of coupled models. In DS-eCD++, the class *Coupled* also contains the implementations of structure components. The class *Port* encapsulates the implementations of input and output ports. In DS-eCD++, the model hierarchy tree has been changed in two ways: on one hand, the structure agents bring the changes to the model hierarchy; on the other hand, the structural changes of the structure components keep updating the model hierarchy during simulation. The class *ModelAdmin* manages the dynamic model hierarchy tree. The revised implementation details of the Modeling Subsystem are characterized as:

1. *ModelAdmin* specifying the implementations of Model Manager creates a structure agent object by means of *registerRevAtomic()* method and builds the structure agent objects database (a dictionary data structure building the relationships between a structure agent's string name and a pointer to the structure agent object). *newRevAtomic()* method creates a structure agent object utilizing the object pointer stored in the objects database. It also employs Processor Manager (see section 4.4) to create a processor for the structure agent. Once the structure agent objects database has been built,

Model Manager permits a dual traverse between the structure components and the associated structure agents. The `execId` allows a structure component to find its associated structure agent; on the contrary, the `parentId` encapsulated in a structure agent enables it to access its parent model.

2. The class *Model* encapsulates the logic implementations of model. There are three model types including `atomicType`, `coupledType` and `revatomicType` which are included in `modelType`. *delInputport()* and *delOutputport()* methods are defined to remove an input/output port from a coupled model. *strucChange(int &)* and *strucChange()* methods are inherited by an instantiated atomic model and call the homonymous methods in the simulator associated with the atomic model to assign and retrieve the structural value. The structural value is retrieved through *strucChange();* while a new structural value is assigned via *strucChange(int &).*
3. *Atomic* encapsulates the implementations of atomic models. In eCD++, five transition functions, including *InitFunction(), ExternalFunction(), InternalFunction(), ConfluentFunction()* and *OutputFunction(),* specify atomic model behaviours. In DS-eCD++, *StartFunction()* is introduced to re-initialize atomic models when the models join the simulation via structural changes.
4. *RevAtomic* specifies the implementations of structure agents. Similar with atomic models, *RevAtomic* employs transition functions to describe the behaviours of structure agents. *InitFunction()* is used to initialize a structure

agent. The structural transitions of a structure agent are specified in *IntenralFunction()*. In addition, *RevAtomic* encapsulates a group of structural change operations (explained in section 4.3.1), which are called by the concrete structure agent in their internal structural transition functions.

5. *Coupled* encapsulates the implementations of regular coupled models and structure components. The implementations of structure components are added to *Coupled* in DS-eCD++. `ExecId` is an attribute of structure components specifying the model ID of the structure agents. This attribute can be used to distinguish structure components from regular coupled models. If the value of `ExecId` is a valid integer, the model is a structure component. Otherwise, the model is a regular coupled model. e*xecutive()* method is used to retrieve the `ExecId`. *addmodelId()* and *delmodelId()* methods update the model composition of a structure component in a structural change. In the nested structural change process, the structural change is executed from bottom to up. The structure component at higher hierarchical level should store the structural change value in `strucvalue` via *setvalue()*. *getvalue()* retrieves the structural value when the structural change is recalled at this structure component. `childs` and `oldchilds` are two lists storing the new model composition and the model composition to be changed in a structure component. The difference between the two lists occurs when the structure component is experiencing a structural change. The model composition stored in `childs` is updated along with the structural change. The model

composition to be changed is backup in `oldchilds` using *oldchildren()* method. The differences between the two lists are calculated by *setnewmodels()* method and s*etremmodels()* method. *setnewmodels()* method stores the models to be added in a data member `newmodels`; while s*etremmodels()* method sends the models to be removed in another data member `removedmodels`. *getnewmodels()* method is invoked by the corresponding coordinator to retrieve the models from `newmodels`. *getremmodels()* method gets the models to be removed from `removedmodels`. These two methods are invoked by the corresponding coordinator to adjust its simulating behaviour. The St messages are sent to the models in `newmodels` to re-initialize the models for the next simulation cycle. The models in `removedmodels` are deleted from the synchronized set in the coordinator and are removed from simulation system finally.

6. *Port* defines a series of implementations of input/output ports. As we have explained in 4.2.2, the model structure is flattened if a flat coordinator is applied in simulation. *flatterninfluences()* extends the flattening in the class *Port* to update the influences of a port.

Structural Change Forms and the Operation Boundaries

Before introducing the structural change operations, the structural change forms and the operation boundaries are discussed. The discussion of structural change form gives a clue to investigate the structural change operations; while operation boundaries regulate structural changes in a safe and clear scope. The structural change forms also

provide useful hints in designing structural change scenarios and structural change cases.

In DEVS-based simulation systems, there are three kinds of component elements: component (an atomic model or a coupled model), coupling (links between components) and port (input port or output port). Structural changes aim to adjust the layout of the component elements. Therefore the basic structural change forms can be identified in the six types. 1) addition of a component; 2) removal of a component; 3) addition of a link between components; 4) removal of a link between components; 5) addition of an input / output port; 6) removal of an input / output port. The basic structural change forms constitute the structural changes in most cases. Update of a component refers to a component is updated by a new version which might have totally different behavior or interface from the old one. This can be considered as a composition of the basic structural change forms and can be accomplished by simply replacing the old component with a new one, which involves the addition and removal operations. According to the basic structural change forms, the structural change operations are defined. The structural change operations are combined together to accomplish most possible structural changes in non-distributed systems.

Structural changes cause modifications in the model hierarchy. Sometime conflicts between the structural change processes occur if the expected model structures call for opposite operations such as addition and removal of the same component etc. Operation

boundaries should be defined to avoid the conflicts and to regulate the structural changes operations in a conflict-free and determined range.

To specify the operation boundaries, the location information in relation to the model hierarchy of the all kinds of components should be analyzed. A component of a coupled model has knowledge of its parent, children list and the couplings among the children. A component of an atomic model is aware of its parent and its input/output ports. The components belonging to the same parent are brothers. The brother components are independent from each other. A component contains no information of its brother. As being defined in FDSDE, a structure agent is introduced to execute the structural change for a structure component. The structure agent is taken as a revised atomic model aware of only its parent and works on behalf of its parent. We can take a structure agent as a structural representative of a structure component. That is to say, a structure agent holds as the same structural view as the structure component. With the structural views of all the components, the operation boundaries can be defined:

1. The structural changes in a structure component are conducted by the associated structure agent. A structure component and an atomic model have no capability to dispose structure change operations.
2. Addition / removal of a component refer to add or remove an atomic model. A structure component is a model structure governor and can switch its model structure from one to the other with the help of the associated structure agent. The structure component itself would not be added or removed. A structure agent is

always associated with a structure component; therefore cannot be added or removed as well.

3. An atomic model is a structure unit and involves no structural change in it. The ports in an atomic model would not be changed during simulation. If different port sets in an atomic model are needed in different simulation stage, the union of the port sets are defined in the atomic model definition and the specific port sets are used at certain simulation stage.
4. Addition / removal of input / output ports are used to add or remove input / output ports in a structure component. By which the interface of the structure component is changed.
5. A structure component can only add / removed the couplings in which the sender and receiver pertain to the structure component. In case of a coupling spanning two different structure components, the situation becomes complex. Consider A is the sender of the coupling and B is the receiver of the coupling. The DEVS property, closure under coupling, ensures that there is a structure component existed to cover A and B. That is to say, the structure component is the parent of either A or B and the parent of the ancestor of either A or B. The nested structural change process is able to handle the situation link this.

4.3.2. Structural Change Operations

A group of structural change operations are defined in class *RevAtomic*, including structural change operations and supplementary operations:

- Structural change operations:
  - GetModels
  - GetLinks
  - GetInputPorts/GetOutputPorts
  - Add/Remove models
  - FullAdd/FullRemove models
  - Add/Remove links
  - Add/Remove input/output ports.
  - DiffLinks
  - DiffModels
  - DiffInputPorts/DiffOutputPort
- Supplementary operations:
  - FindModel
  - FindInputPort/FindOutputPort

The structural change operations provide necessary manipulations to the component elements (models, links and ports). *Get* actions retrieve the specified component elements. *Diff* actions aim to calculate the differences between the component elements subject to be changed and the component elements expected to join. *Add/Remove & FullAdd/FullRevmode* actions realize the actions of addition / removal of the component elements. The operations can be performed in two ways: full operations and light operations. In the full operations, the atomic model objects and the associated simulator objects are added / removed along with the addition / removal of the model references and the processor references in the structure components. Simple operations only add or remove the references of the atomic models in the structure component while keep the model objects and the associated simulators in the model object databases. The former operations are suitable for the new atomic models added in the simulation system or the atomic models removed permanently from the simulation system. If the models are removed temporarily at the previous simulation stage and will be reused at later simulation stage, the simple operations can be applied.

Two sets of operations offer flexibility to modellers who can keep balance between minimum memory usage and fast loading time. Supplementary operations are used to locate the component elements. Those operations are called by the concrete structure agents to define the real structural change operations in the simulation.

Simulation Subsystem

The Simulation Subsystem presents simulators and coordinators hierarchically. FDSDE redefines the message-passing algorithms for the abstract simulators. Accordingly, the abstract simulator classes in the Simulation Subsystems are revised to fit the changes. The class diagram of Simulation Subsystem is presented in Fig. 6. The modifications in each class are listed.

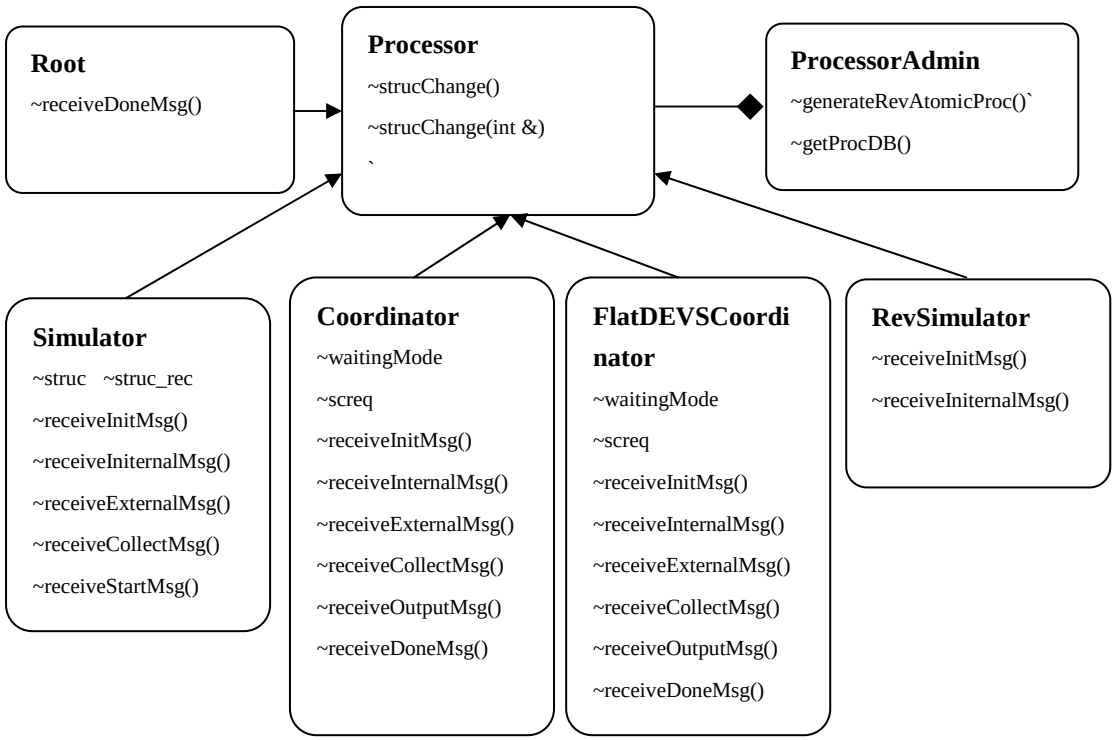


Fig 29. The Simulation Subsystem Class Diagram

The class *processor* defines virtual operations for the abstract simulators. The classes *Root*, *Coordinator*, *Simulator* and *RevSimulator* representing the abstract simulators are derived from the class *processor* to execute the corresponding message-passing mechanisms (refer to the section 3.3). *FlatDEVSCoordinator* is another subclass derived from the class *processor* to represent the abstract simulator of the flat coordinator. The concrete simulators and coordinators are instantiated from the abstract simulator classes. Class *ProcessorAdmin* plays as a processor manager responsible for generating the processors and maintaining a processor objects database (a dictionary database building a relationship between a processor id and a pointer to the processor object). The modifications in the Simulation Subsystem are concluded as:

1. The receive functions for the different types of messages received in each abstract simulator are replaced with the message-passing algorithms described in the section 3.3.
2. *generateRevAtomicProc()* in *ProcessorAdmin* is called by *NewRevAtomic()* in *ModelAdmin* to generate a concrete processor for a structure agent. With the help of *getProcDB()*, the processor objects database can be accessed by a structure agent to add / delete a processor when the associated model has been added / deleted.
3. *strucChange(int &)* and *strucChange()* methods are inherited by the concrete simulators to assign and retrieve the structural value of the processors. A new structural value is assigned through *strucChange(int &)* method in the internal transition function, the external transition function or the confluent transition function of an atomic model if the atomic model tries to raise a structural change

request. Two data members in the corresponding simulator hold the structural value and *strucChange(int &)* method updates the data member `struc` in the simulator. If a new structural value is assigned, `struc` contains a different value with `struc_rec`. The simulator detects the difference between the two data members via *strucChange()* and determines whether if a structural change request is raised. The *receiveStartMsg()* in Simulator takes charge of message processing of start messages which are received in an atomic model.

4. *RevSimulator* encapsulates the message-passing algorithm of RevSimulator. The processor instantiated from *RevSimulator* generates the behaviours of structure agents. *RevSimulator* handles initial message, which initializes the model, and the structural change messages, which invoke the structural change processes in structure agents with the expected structural values.

Messaging Subsystem

Message Subsystem is responsible for management of message classes and maintenance of a message queue. Virtual attributes and operations of a message are defined in *Message*. Seven message classes are inherited from the virtual class encapsulating the corresponding messaging implementations. Classes *InitMessage*, *InternalMessage*, *ExternalMessage*, *DoneMessage*, *OutputMessage*, *CollectMessage* and *StartMessage* represent initial message, internal message, external message, done message, output message, collect message and start message respectively. The message class diagram is shown in Fig. 7.

1. *MessageAdmin* as a message manager maintains an unprocessed message queue and dispatches messages.
2. *DoneMessage* and *InternalMessage* classes are extended to represent a structural change request and a structural change message respectively. In eCD++, done message and internal message are used for simulation control purpose and no value involved. In DS-eCD++, done message and internal message require message values to carry structural values. In *DoneMessage* class and *InternalMessage* class, *setvalue(value)* method sets a structural value in the data member `value`. The structural value can be accessed via *getvalue()* method. The concrete messages are the instantiations of the message classes.

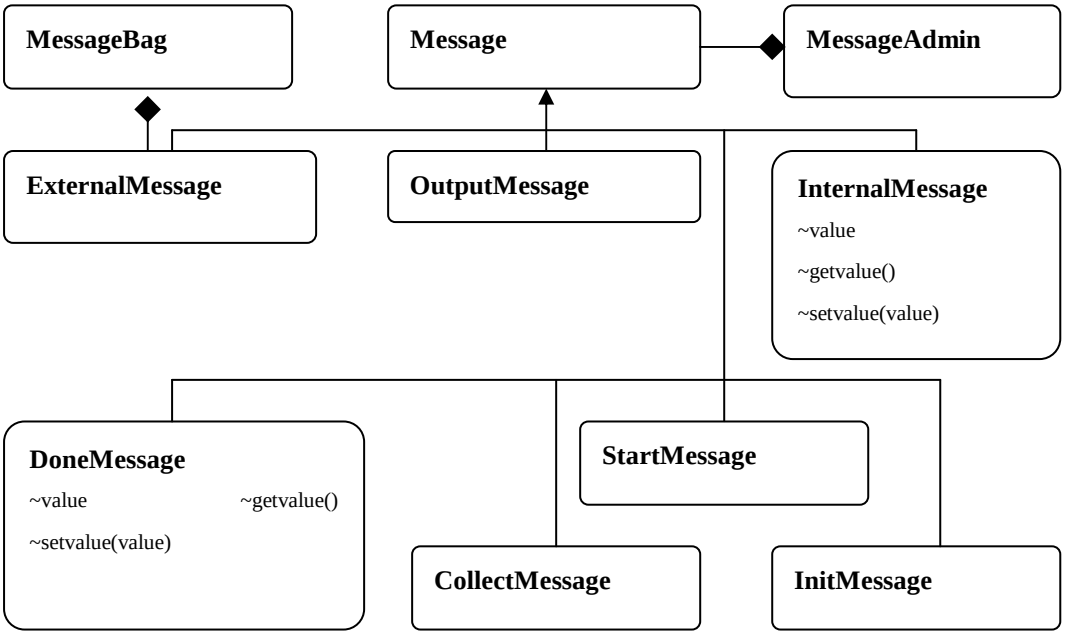


Fig 30. Messaging Subsystem

Functionalities of DS_ECD++

The functionalities of DS_ECD++ are investigated in this section. The most important functionality is dynamic structural change. In Chapter 3, we discussed the basic structural change forms. The compositions of those basic structural change forms constitute a variety of structural change scenarios. DS_ECD++ is able to perform the various structural change scenarios by combining the basic structural change forms. The structural changes may be raised at any time. FDSDE specifies that the structural change has higher priority over other simulation events. When a structural change request becomes imminent, other imminent events have to wait until the next simulation cycle. A structural change request can be raised in the internal transition function, the external transition function of an atomic model. The message-passing paradigms defined FDSDE are in line with the P-DEVS formalism; therefore parallel simulation is possible in DS-eCD++. That is to say, a structural change request can be handled in the confluent function of an atomic model. Moreover, DS-eCD++ also supports the nested dynamic structural change, in which a structural change request may cause a series of structural change processes in different structure components in the model hierarchy.

The revised flat coordinator supports dynamic structure simulation with a flat coordinator. In the simulation with a flat coordinator, the solo structure agent executes the structural change for the structure component – flattop.

DS-eCD++ supports dynamic structure in real time. Employing the interval time function, DS-eCD++ enables to run the simulation with real time advance. Sometime, the dynamic structural change in real time simulation takes longer time than that in virtual time simulation for the structural change needs more time to process.

DS-eCD++ is able to cooperate with the GGAD interpreter to implement simulation using GGAD-defined DEVS models. Whatever one or more atomic models are replaced with the GGAD equivalents, the dynamic structure simulation can run as exactly the same as the simulation with C++ language defined models. Also, the dynamic structure simulation with the GGAD models fits both virtual time advance and real time advance.

The functionalities of DS-eCD++ can be embodied by the structural change scenarios. In the next chapter, the structural change scenarios are devised and the corresponding case studies are conducted to test the functionalities.

Chapter 5 Structural Change Scenarios and Case Studies

In order to evaluate the FDSDE algorithm and the software logic, the case studies are investigated in this chapter. The structural change scenarios presented in the first section combines the basic structural change forms described in Chapter 3 and the major functionalities in eCD++. In the following sections, two cases: DSAMS (Dynamic Structure Automatic Manufacturing System) and MTRS (Motor Tracing and Replacement System) are studied. In each case, a series of experiments are provided to verify the structural change scenarios. For each case, the model description is explained. The structure components in the cases are identified and the formal specifications of the structure components based on the DSDEVS formalism are exhibited. A series of experiments covering a coupled of the structural change scenarios are carried out and the simulation results are analyzed.

Structural Change Scenarios

Ten structural change scenarios are presented to evaluate the dynamic structural change functionality and the compatibility with eCD++.

- **Scenario 1:** Structural change request is raised in the external function of an atomic model
- **Scenario 2:** Structural change request is raised in the internal function of an atomic model
- **Scenario 3:** The structural changes involving transition conflicts can be

properly handled by means of confluent transition function of an atomic model

- **Scenario 4:** Addition or/and removal of internal links (The sender and the receiver of the links are within a coupled model).

- **Scenario 5:** Addition or/and removal of an atomic model

- **Scenario 6:** Replacement of a coupled model

- **Scenario 7:** A nested structural change process caused by a structural change request.

- **Scenario 8:** Structural changes in a flat coordinator.

- **Scenario 9:** Structural changes of the interface of a coupled model. (changes of the input ports or/and output ports of a coupled model)
- **Scenario 10:** Running dynamic structure real time simulation using the GGAD models

Case 1 DSAMS

Description

DSAMS (Dynamic Structure Automated Manufacturing System) is composed by the dedicated stations that perform assembling and painting tasks on different products in a manufacturing plant, including a conveyor belt that transports the products to/from those workstations. The *Controller Unit* is an atomic model used to control the actions of the *Conveyor* according to external inputs (which schedule the manufacturing of a

given product). The *Conveyor* transports the products being manufactured to the other units, as indicated by *Controller Unit*. The *Conveyor* itself is a coupled model consisting of an *Engine* (to move the belt) and a *Sensor* (to detect the current position in order to decide when we need to stop the belt). The *Engine Assembly* workstation *(ES)* is an atomic model, modeling a dedicated workstation standing beside the *Conveyor* to take assembling tasks. The second dedicated workstation - *Painting workstation (PS)* - is a coupled model containing a *Painter* (which paints the products) and two models of painting arms: *Chrome* and *Color*. The timing parameters used in DSAMS are shown in the table.

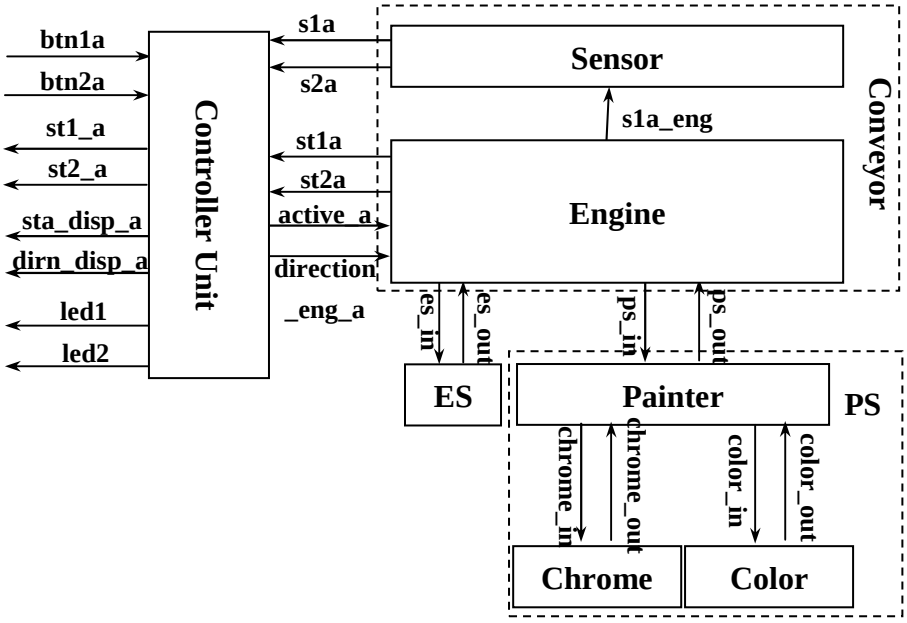


Fig 31. The Scheme of DSAMS

Table 1 The Timing Parameters in DSAMS

Model Name	Time Variables	Duration	Description
Engine	preparationTime2Start	Time(0, 0, 0, 5)	start a product
	preparationTime2Stop	Time(0, 0, 0, 5)	stop a product
	movingTime	Time(0, 0, 0, 5)	move from one station to the other
ES	workingTime1	Time(0, 0, 1, 0)	Working time during daytime
ES'	workingTime2	Time(0, 0, 1, 050)	Working time during night
Painter	workingTime1	Time(0, 0,2, 20)	paint color and chrome
	workingTime2	Time(0, 0, 1, 0)	paint color
	workingTime3	Time(0, 0, 2, 0)	Paint chrome
Chrome	preparationTime	Time(0, 0, 0, 10)	prepare chrome painting arm
Color	preparationTime	Time(0, 0, 0, 20)	prepare color painting arm

Initially, a product is placed on the *Conveyor* belt besides the *ES*, waiting for the indications from the *Controller Unit*, which receives the external events from *btn1A* indicating that the product is processed in *ES*, and from *btn2A*, which tells that the product is processed in *PS* (as we can see in the table, multiple events are received throughout the simulation on each of the buttons). The *Controller Unit* also receives the status of the products on *Conveyor* from *Sensor* (from inputs *s1a* and s2a), and outputs them through the output ports *sta_disp_a* or *dirn_disp_a*. *sta_disp_a* displays the number of the station that the product has reached (ES = 11 and PS = 21); while *dirn_disp_a* indicates the moving direction of the conveyor (0: stopped, 1: moving forward, and 2: moving backward). Two LED output ports, *led1* and *led2*, are associated with the two stations (*ES* and *PS*). The corresponding LED turns on (value = 1) when the destined station is assigned, and turns off (value = 0) when the product

reaches the station. The completion of the tasks in the stations is indicated by the two output ports *st1_a* and *st2_a* respectively. The *Engine* receives indications from the *Controller Unit* via *active_a* and *direction_eng_a*. The expected station is input through *active_a* indicating the destination the *Engine* moves to. The moving direction of the Engine is designated via *direction_eng_a*. *s1a_eng* tells the *Sensor* the current station the *Engine* reached. The *Engine* starts *ES* and receives the ending signal from *ES* via *es_in* and *es_out*. *ps_in* and *ps_out* are used to signals *PS* and reports the task completion in *PS*. The *Painter* initiates chrome arm and color arm via *chrome_in* and *color_in*. The preparation done messages are returned from the *Chrome* and the *Color* through *chrome_out* and *color_out*.

Two kinds of changes are considered in the DSAMS:

1) Variation of the duty shift, which will produce a change between *ES* and *ES1*. For the simulation, it is supposed that the duty time is 10 minutes for both *ES* and *ES1*.
2) Switch of painting modes. Some products need painting both color and chrome (painting mode = 1) while other products require painting either color (painting mode = 2) or chrome (painting mode = 3). The painting mode is indicated by the CU, which will generate an external event representing the corresponding painting mode.

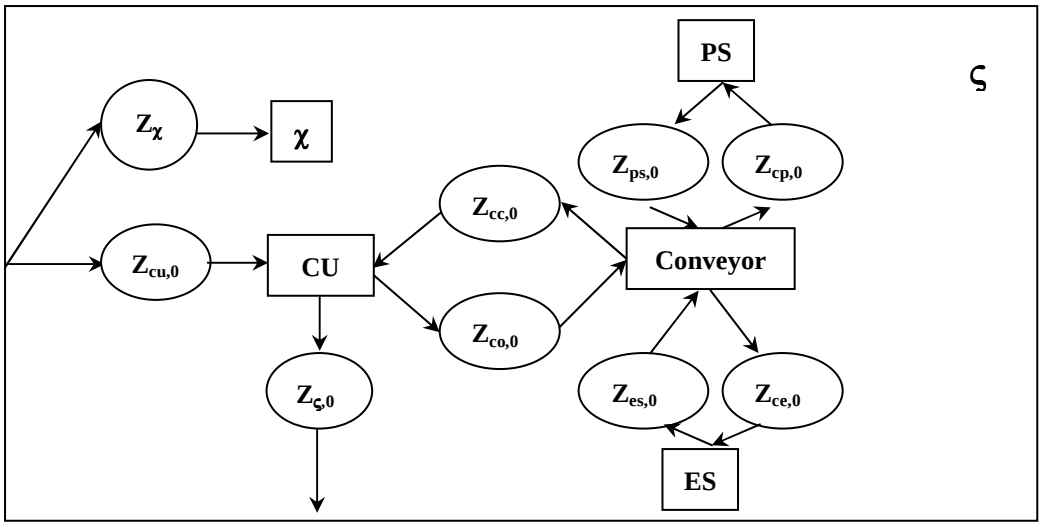
Formal Specifications

The structure components in DSAMS are shown in a formal diagram, in which a rectangle with a name represents a model, while an ellipse with $Z_{i, j}$ denotes a transition function (i is the model name, j refers to its structural state). χ indicates the structure agent associated with the structure component. Z_χ is the internal transition function of the structure agent.

The formal specifications give formal definitions of the structure components. ζ represents the structure component TOP; while π denotes the structure component PS.

$\varsigma = (X_\varsigma, Y_\varsigma, \chi, M_\chi)$ $X_\varsigma = \{activate\}$ $Y_\varsigma = \{out\}$ $M_\chi = \{X_\chi, s_{0,\chi}, S_\chi, \delta_\chi, \tau_\chi\}$
$X_\chi = \{struc1, struc2\}$ $S_\chi = \{s_{0,\chi}, s_{1,\chi}\}$ $\tau s_{0,\chi} = \tau s_{1,\chi} = 10$ minutes
$\delta_\chi (s_{0,\chi}, e, change) = s_{1,\chi}$ $\delta_\chi (s_{1,\chi}, e, change) = s_{0,\chi}$
$\gamma (s_{0,\chi}) = \{D_0, \{M_{i,0}\}, \{I_{i,0}\}, \{Z_{i,0}\}\}$ $\gamma (s_{1,\chi}) = \{D_1, \{M_{i,1}\}, \{I_{i,1}\}, \{Z_{i,1}\}\}$
$D_0 = \{CU, Conveyor, PS, ES\}$ $D_1 = \{CU, Conveyor, PS, ES'\}$
$M_{cu,0} = M_{cu,1} = \{ X_{cu}, s_{0,cu}, S_{cu}, Y_{cu}, \delta_{cu}, \lambda_{cu}, \tau_{cu}\}$
$M_{co,0} = M_{co,1} = \{ X_{co}, s_{0,co}, S_{cc}, Y_{cc}, \delta_{cc}, \lambda_{cc}, \tau_{cc}\}$
$M_{ps,0} = M_{ps,1} = \{ X_{ps}, s_{0,ps}, S_{ps}, Y_{ps}, \delta_{ps}, \lambda_{ps}, \tau_{ps}\}$
$M_{es,0} = \{ X_{es}, s_{0,es}, S_{es}, Y_{es}, \delta_{es}, \lambda_{es}, \tau_{es}\}$
$M_{es',0} = \{ X_{es'}, s_{0,es'}, S_{es'}, Y_{es'}, \delta_{es'}, \lambda_{es'}, \tau_{es'}\}$
$I_{cu} = \{ \varsigma \}$ $I_{co} = \{CU\}$ $I_{cp} = \{Conveyor\}$ $I_{cc} = \{Conveyor\}$ $I_{ce} = \{Conveyor\}$ $I_{ps} = \{PS\}$ $I_{es} = \{ES\}$ $I_{es'} = \{ES'\}$ $I_{\chi,0} = I_{\chi,1} = \{\varsigma\}$
$Z_{\chi,0} = Z_{\chi,1} = Z_\chi$ $Z_\chi: X_\varsigma \rightarrow X_\chi$ $Z_{cu,0} = Z_{cu,1} = Z_{cu}$ $Z_{cu} : X_\varsigma \dashrightarrow X_{cu}$
$Z_{cc,0} = Z_{cc,1} = Z_{cc}$ $Z_{cc} : X_{cu} \rightarrow X_{co}$ $Z_{co,0} = Z_{co,1} = Z_{co}$ $Z_{co} : X_{co} \rightarrow X_{cu}$
$Z_{ps,0} = Z_{ps,1} = Z_{ps}$ $Z_{ps} : X_{ps} \rightarrow X_{co}$ $Z_{cp,0} = Z_{cp,1} = Z_{cp}$ $Z_{cp} : X_{co} \rightarrow X_{ps}$
$Z_{es,0} : X_{es} \rightarrow X_{co}$ $Z_{ce,0}$ $Z_{ce} : X_{ce} \rightarrow X_{co}$ $Z_{es',1} : X_{es'} \rightarrow X_{co}$ $Z_{ce,1} : X_{co} \rightarrow X_{es'}$

a)



b)

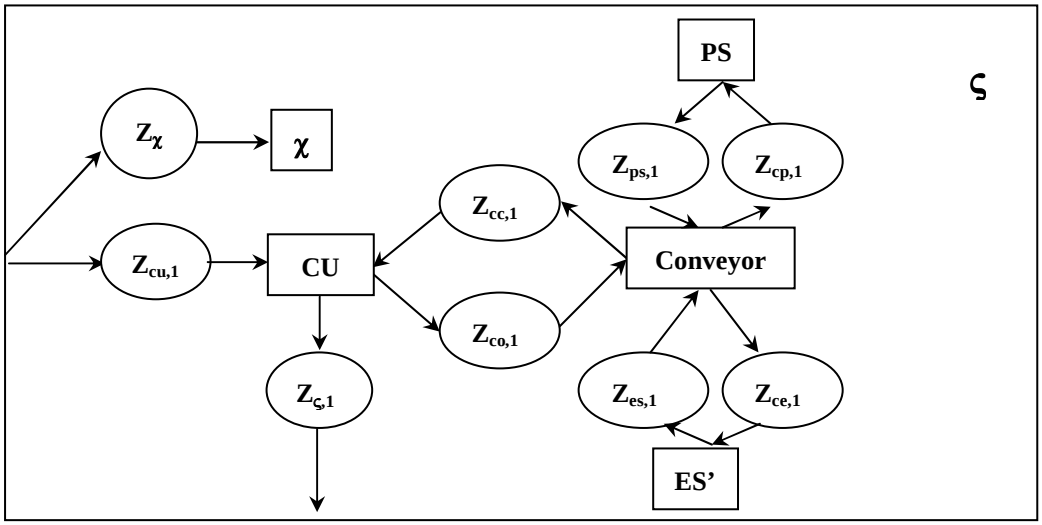
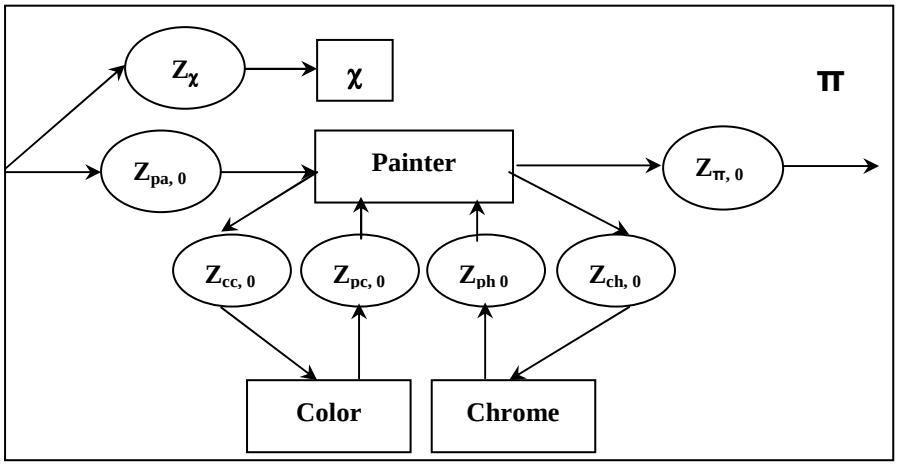
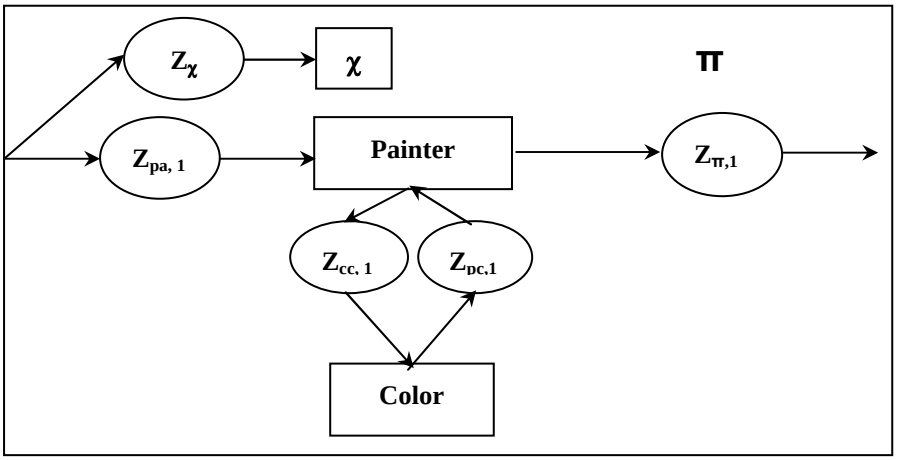


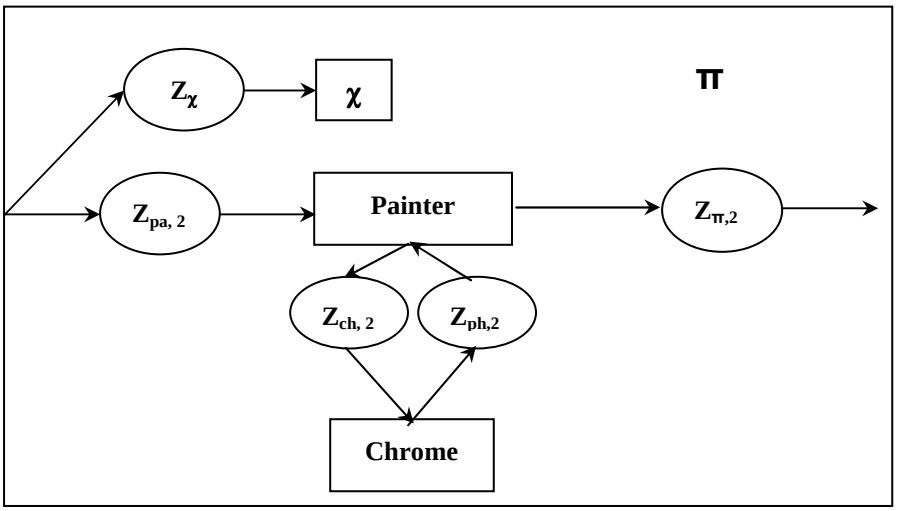
Fig 32. a) The Model Structure of the *TOP* including *ES* b) The Model Structure of the *TOP* including *ES'* c) Formal specification of the structure component *TOP*



a)



b)



c)

$\pi = (X_\pi, Y_\pi, \chi, M_\chi)$ $X_\pi = \{activate\}$ $Y_\pi = \{out\}$ $M_\chi = \{X_\chi, s_{0,\chi}, S_\chi, \delta_\chi, \tau_\chi\}$

$X_\chi = \{changemode1, changemode2, changemode3\}$

$S_\chi = \{s_{0,\chi}, s_{1,\chi}, s_{2,\chi}\}$

$\tau\, s_{0,\chi}$ = workingTime1 $\tau\, s_{1,\chi}$ = workingTime2 $\tau\, s_{2,\chi}$ = workingTime3

$\delta_\chi (s_{0,\chi}, e, changemode2) = s_{1,\chi}$ $\delta_\chi (s_{1,\chi}, e, changemode3) = s_{2,\chi}$

$\delta_\chi (s_{0,\chi}, e, changemode3) = s_{2,\chi}$ $\delta_\chi (s_{1,\chi}, e, changemode1) = s_{0,\chi}$

$\delta_\chi (s_{2,\chi}, e, changemode1) = s_{0,\chi}$ $\delta_\chi (s_{2,\chi}, e, changemode2) = s_{1,\chi}$

$\gamma (s_{0,\chi}) = \{D_0, \{M_{i,0}\}, \{I_{i,0}\}, \{Z_{i,0}\}\}$ $\gamma (s_{1,\chi}) = \{D_1, \{M_{i,1}\}, \{I_{i,1}\}, \{Z_{i,1}\}\}$ $\gamma (s_{2,\chi}) = \{D_2, \{M_{i,2}\}, \{I_{i,2}\}, \{Z_{i,2}\}\}$

$D_0 = \{Color, Painter\}$ $D_1 = \{Color, Chrome, Painter\}$ $D_2 = \{Chrome, Painter\}$

$M_{cc,0} = M_{cc,1} = \{ X_{cc}, s_{0,c}, S_c, Y_c, \delta_{cc}, \lambda_{ccl}, \tau_{cc}\}$

$M_{pan,0} = M_{pan,1} = \{ X_{pa}, s_{0,pa}, S_{pa}, Y_{pa}, \delta_{pa}, \lambda_{pa}, \tau_{pa}\}$

$M_{ch,1} = M_{ch,2} = \{ X_{ch}\, s_{0,ch}, S_{ch}, Y_{ch}, \delta_{ch}, \lambda_{ch}, \tau_{ch}\}$

$I_{cc,0} = I_{cc,1} = \{Painter\}$ $I_{pa,0} = \{\pi, Color\}$ $I_{pa,1} = \{\pi, Color, Chrome\}$ $I_{pa,2} = \{\pi, Chrome\}$

$I_{\chi,0} = I_{\chi,1} = I_{\chi,2} = \{\pi\}$
$Z_{\chi,0} = Z_{\chi,1} = = Z_{\chi,1} = Z_{\chi}$ and Z_{χ}: $X_{\pi} \rightarrow X_{\chi}$ $Z_{cc,0} = Z_{cc,1} = Z_{cc}$ Z_{cc}: $X_{pa} \rightarrow X_{cc}$
$Z_{ch,1} = Z_{ch,2} = Z_{ch}$ Z_{ch}: $X_{pa} \rightarrow X_{ch}$ $Z_{pa,0} = X_{cc,0}$ x X_{π} $Z_{pa,1} = X_{cc} \times X_{ch}$ x X_{π} $Z_{pa,2} = X_{ch}$ x X_{π}
$Z_{\pi,0} = Y_{pa,0}$ $Z_{\pi,1} = Y_{pa,1}$ $Z_{\pi,2} = Y_{pa,2}$

Fig 33. a) PS workstation with color painting arm b) PS workstation with color and chrome painting arms c) PS workstation with chrome painting arm d)Formal specification of the structure component PS

Model Definitions

The model definition of DSAMS using CD++ build-in specification language is listed in the following figure. In the model definition, the model structures of the two structure components are specified. TOP has two structural states and PS has three structural states.

```
[top]
components : conveyorA topexec#TOPEXEC
components : dsecu@DSECU es@ES ps
in : btn1A btn2A st1A_in st2A_in
out : led1 led2 stn_disp_A dirn_disp_A st1_A st2_A
Scomm : struc1
Link : btn1A b1A@dsecu       Link : btn2A b2A@dsecu
Link : activate_A@dsecu     activate_A@conveyorA
Link : direction_eng_A@dsecu    direction_eng_A@conveyorA
Link : pmodeA@dsecu pmode_in@conveyorA
Link : es_in@conveyorA inA@es    Link : ps_in@conveyorA inA@ps
Link : outA@es es_out@conveyorA  Link : outA@ps ps_out@conveyorA
Link : s1A@conveyorA s1A@dsecu   Link : s2A@conveyorA s2A@dsecu
Link : st1@conveyorA st1A_in@dsecu Link : st2@conveyorA st2A_in@dsecu
Link : l1@dsecu led1               Link : l2@dsecu led2
Link : station_display_A@dsecu stn_disp_A
Link : direction_display_A@dsecu dirn_disp_A
Link : st1_A@dsecu st1_A        Link : st2_A@dsecu st2_A
Link : st1_out@engA st1         Link : st2_out@engA st2
```

[conveyorA]
components : engA@engineA dsscA@dssensorboxA
in : activate_A direction_eng_A pmode_in es_out ps_out
out : s1A s2A st1 st2 es_in ps_in
Link : activate_A startstop@engA
Link : direction_eng_A engdirection@engA
Link : es_out es_out@engA **Link :** ps_out ps_out@engA
Link : pmode_in pmode_in@engA **Link :** floor@engA s1A_eng@dsscA
Link : sen1A@dsscA s1A **Link :** sen2A@dsscA s2A
Link : es_in@engA es_in **Link :** ps_in@engA ps_in
[ps]

components : painter@Painter color@Color chrome@Chrome psexec#PSEXEC

in : inA **out :** outA **Scomm** : struc1

Link : inA inA@painter **Link :** outcolor@painter in@color

Link : outchrome@painter in@chrome **Link :** out@chrome inchrome@painter

Link : out@color incolor@painter **Link :** outA@painter outA

[topUpdate1]

components : conveyorA dsecu@DSECU es1@ES1 ps
in : btn1A btn2A st1A_in st2A_in
out : led1 led2 stn_disp_A dirn_disp_A st1_A st2_A
Scomm : struc2
…
Link : es_in@conveyorA inA@es1 **Link :** ps_in@conveyorA inA@ps
Link : outA@es1 es_out@conveyorA **Link :** outA@ps ps_out@conveyorA
…

[psUpdate1]

components : color@Color painter@Painter

in : inA **out :** outA **Scomm :** struc2

Link : inA inA@painter **Link :** outcolor@painter in@color

Link : out@color incolor@painter **Link :** outA@painter outA

[psUpdate2]

Fig 34. Model Definitions of DSAMS

The DSAMS Experiments Using DS-eCD++

Experiment 1

This experiment aims to verify the dynamic structure of the simulation environment. The atomic models of the DSAMS were defined in C++, and the compositions and the couplings are specified in the coupled models. *PSEXEC* is a structure agent executing the structural changes on behalf of *PS* according to the indicated painting modes. *TOPEXEC* is another structure agent taking charge of the duty shifts between *ES1* and *ES* on behalf of *TOP*.

The simulation ran in real time mode and the following table of the external events was scheduled and sent to the Controller Unit. The first job in the table arrived at time 00:00:01:500 from the input port btn1A, which means to put the product at ES (ES'). The value received in btn1A in the last column indicated the working mode in ES (ES'). ES (ES') has the only one possible working model (value = 1). Also, the associated output port of the job was st1_A. The output time should be no later than 00:00:03:500. The remaining jobs scheduler for ES (ES') are the fourth job at 00:00:12: 500, the sixth job at 00:00:19:985 and the seventh job at 00:00:25:000. The jobs scheduled for PS were the second job at 00:00:10:500, the third job at 00:00:10:500 and the fifth job at 00:00:15:000. Among them, the third one was in the painting mode 2 and the fifth one was in the painting mode 3.

Table 2 The Table of the External Events

Event time	Deadline	Input port	Output port	Value
00:00:01:500	00:00:03:500	btn1A	st1_A	1
00:00:04:500	00:00:08:500	btn2A	st2_A	1
00:00:10:500	00:00:13:500	btn2A	st2_A	2
00:00:12:500	00:00:14:500	btn1A	st1_A	1
00:00:15:000	00:00:17:500	btn2A	st2_A	3
00:00:19:985	00:00:23:000	btn1A	st1_A	1
00:00:25:000	00:00:27:500	btn1A	st1_A	1

Four dynamic structure changes were identified during the simulation:

1). At 00:00:10:000, the scheduled duty time of *ES* was expired and a duty shift between *ES* and *ES1* occurred.

2). At 00:00:10:505 (5ms was used to start the *Engine*), *PS* switched its painting mode from 1 to 2. The *Chrome* model was removed, while the models of *Painter* and *Color* were maintained.

3). At 00:00:15:015 (*Engine* took 15ms to be activated and moved to *PS*), *PS* switched its painting mode from 2 to 3. The *Color* model was removed while the *Chrome* model was added to *PS.*

4). At 00:00:20:000, *ES1* shifted the duty to *ES*. It was noticed that the input event arrived at *ES1* at the time 00:00:20:000. Simultaneously, the scheduled internal state of *ES1* was expired. As a result, the confluent function of *ES1* was invoked at 00:00:20:000. In the confluent function of *ES1*, the external transition function was

given a higher priority over the internal transition function. At 00:00:20:000, the *ES1* executes assembling task first, and then the duty shift happens.

Fig. 5 exhibits the structural changes in *PS*. Initially, *PS* was in painting mode 1 (structural state is *PS1*), which included both the color arm and the chrome arm. At 00:00:10:505, the painting mode switched to 2 (structural state is *PS2*), which included the color arm. *PSEXEC* executed the structural changes transferring the structural state of PS from PS1 to PS2.

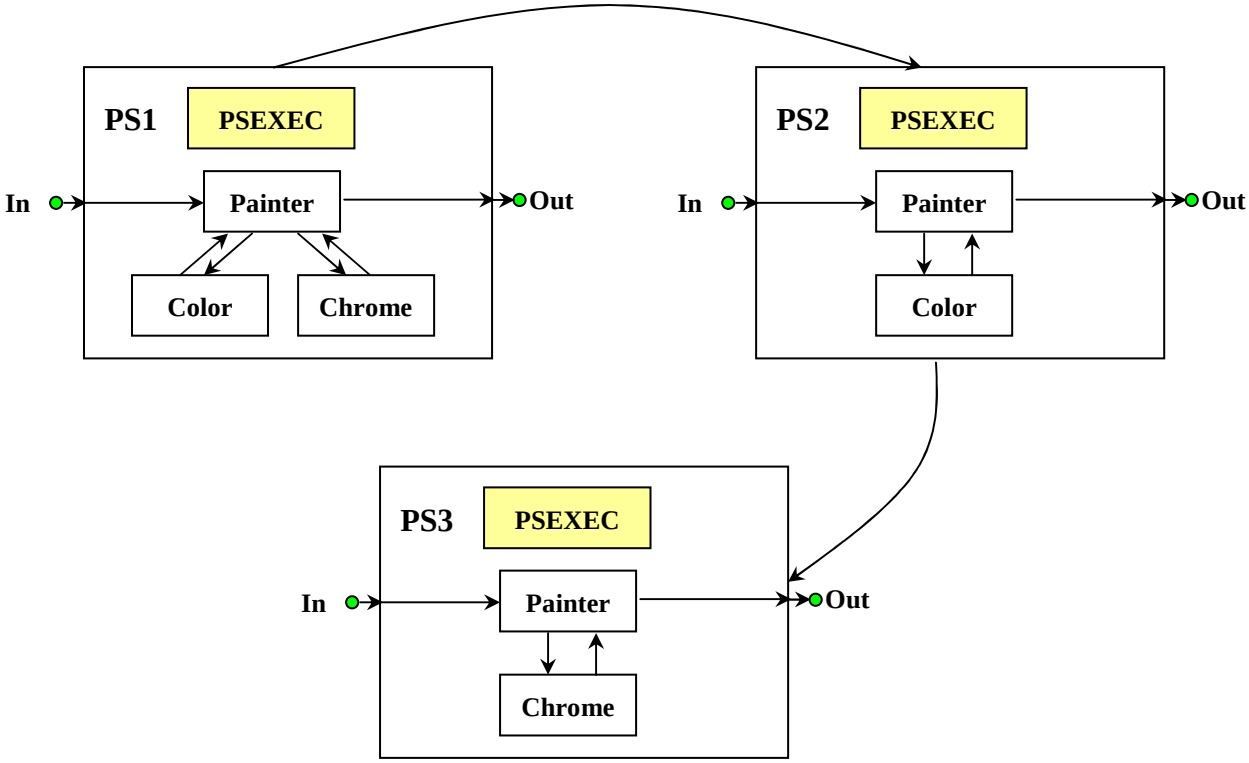


Fig 35. The Structural Changes in PS

Structural Operations in PSEXEC (PS1 → PS2):

1. DelModel (“ps”, “Chrome”)
2. DelLink (“ps”, outlink) (outlink : out@Chrome inchrome@Painter)

3. DelLink (“ps”, inlink) (inlink : outchrome@Painter in@Chrome)

The painting mode shifted to 3 (structural state is *PS3*) at 00:00:15:015. The structural state of PS is changed from PS2 to PS3.

Structural Operations in PSEXEC:

1. DelLink (“ps”, outlink) (outlink : out@Color incolor@Painter)
2. DelLink (“ps”, inlink) (inlink : outcolor@Color, in@Color)
3. DelModel (“ps”, “Color”)
4. AddModel (“ps”, “Chrome”)
5. AddLink (“ps”, outlink) (outlink : out@Chrome inchrome@Painter)
6. AddLink (“ps”, inlink) (inlink : outchrome@Painter in@Chrome)

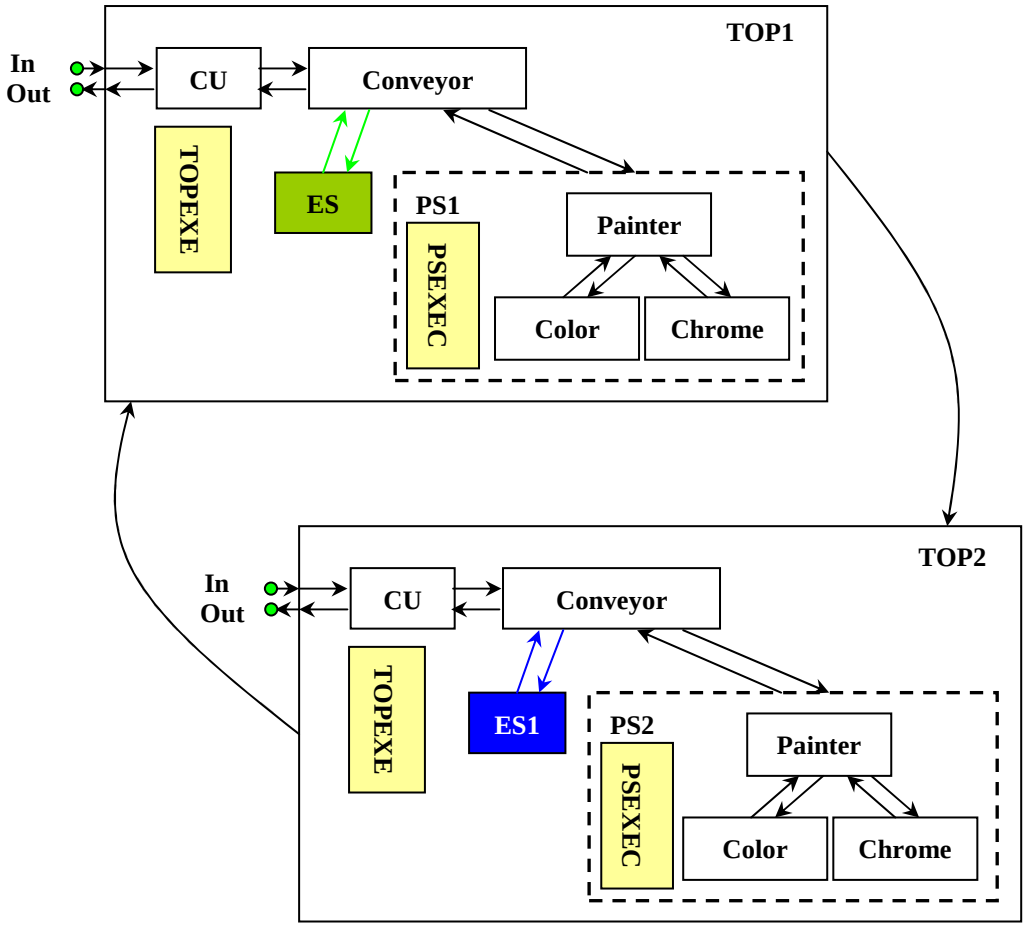


Fig 36. The Structural Changes in TOP

Fig. 6 presents the structural changes in *TOP*. *TOPEXEC* executed the structural changes on behalf of *TOP*. *ES* and *ES1* switched every 10 minutes. At 00:00:10:000, *ES* (structural state is *TOP1*) was replaced by *ES1* (structural state is *TOP2*). The structural operations in TOPEXEC are:

1. DelLink (“top”, outlink) (outlink : out@ES es_in@Conveyor)
2. DelLink (“top”, inlink) (inlink : es_out@Conveyor, in@ES)
3. FullDelModel (“top”, “ES”)
4. FullAddModel (“top”, “ES1”)
5. AddLink (“top”, outlink) (outlink : out@ES1 es_in@Conveyor)
6. AddLink (“top”, inlink) (inlink : es_out@Conveyor, in@ES1)

As scheduled, the duty shift from *ES1* to *ES* would occur at 00:00:20:000. The real duty shift occurred at 00:00:21:500 for the confluent function gave higher priority to the external function of *ES1*. Consequently, the structural change, which happened in the internal function of *ES1*, has been delayed. The structural operations from the structural state TOP2 to the structural state TOP1 are:

1. DelLink (“top”, outlink) (outlink : out@ES1 es_in@Conveyor)
2. DelLink (“top”, inlink) (inlink : es_out@Conveyor, in@ES1)
3. FullDelModel (“top”, “ES1”)
4. FullAddModel (“top”, “ES”)
5. AddLink (“top”, outlink) (outlink : out@ES es_in@Conveyor)
6. AddLink (“top”, inlink) (inlink : es_out@Conveyor, in@ES)

The Experiment 1 involved the five structural change scenarios described in the first section of this Chapter. The switch of painting mode in PS depended on the external value received in the input port ps_in of the Painter. Therefore the structural changes in PS were caused from the external transition function of the Painter (scenario 1). The duty shifts between ES and ES’ were raised by the internal transition function of ES

(ES') (scenario 2), in which the duty time was counted and the structural change is raised when the duty time was expired. As we have described, the fifth job brought transition conflict between the internal transition and the external transition of ES'; therefore, the confluent transition function of ES' was invoked to handle the conflict. In the confluent transition function, the internal transition function gives higher priority to the external transition function. Consequently, the duty shift from ES' to ES was delayed until the end of the external transition function. That is to say, the conflicts can be properly handled in DS-eCD++ employing the confluent transition function (scenario 3). The structural changes in PS and TOP involved the addition&removal of the internal links and the atomic models (scenario 4 and scenario 5).

The simulation results are listed in Table 3. The first column shows the wall-clock value (the time elapsed since the beginning of the simulation execution) at which the outputs have been sent out. The second column is the expected deadlines. The results and the output ports are displayed in the third and the fourth column. The fifth column presents the values output from the output ports. According to the external event time and the timing parameters shown in the table 1 and the table 2, we have verified that the results reflect the external events correctly and meet the expected deadlines.

Table 3. Simulation Results in Experiment 1

Output time	Deadline	Result	Output Port	Value
00:00:02:510	00:00:03:500	Succeed	st1_A	1
00:00:04:500	No deadline		Led2	1
00:00:04:500	No deadline		dirn_disp_a	1
00:00:04:510	No deadline		sta_disp_a	21

00:00:04:510	No deadline		dirn_disp_a	0
00:00:04:510	No deadline		led2	0
00:00:06:560	00:00:08:500	Succeed	st2_A	1
00:00:11:520	00:00:13:500	Succeed	st2_A	1
00:00:12:500	No deadline		led1	1
00:00:12:500	No deadline		dirn_disp_a	2
00:00:12:510	No deadline		sta_disp_a	11
00:00:12:510	No deadline		dirn_disp_a	0
00:00:12:510	No deadline		led1	0
00:00:14:030	00:00:14:500	Succeed	st1_a	1
00:00:15:000	No deadline		led2	1
00:00:15:000	No deadline		dirn_disp_a	1
00:00:15:010	No deadline		sta_disp_a	21
00:00:15:010	No deadline		dirn_disp_a	0
00:00:15:010	No deadline		led2	0
00:00:17:040	00:00:17:500	Succeed	st2_A	1
00:00:19:985	No deadline		led1	1
00:00:19:985	No deadline		dirn_disp_a	2
00:00:19:995	No deadline		sta_disp_a	11
00:00:19:995	No deadline		dirn_disp_a	0
00:00:19:995	No deadline		led1	0
00:00:21:510	00:00:23:000	Succeed	st1_a	1
00:00:26:020	00:00:27:000	Succeed	st1_a	1

The messages log the simulation details. It is noticed that the light operations were used in the structural changes in PS; while the full operations were applied in the structural changes in TOP. When the simulation starts, the model ids are designated (shown in the figure). In the third job, the Chrome model is removed using the light operation DelModel(). When the Chrome model was reused in the fifth job, the model reference (id = 10) was simply added into the children list of PS (figure). The model id of ES model was 06 at the beginning of the simulation. The ES model was removed with FullDelModel() during the duty shift at 00:00::10:000. The new model id (id = 16) was assigned to ES model when ES rejoins the simulation at 00:00:21:500.

```
MSG: I / 00:00:00:000 / Root(00) TO top(01)
MSG: I / 00:00:00:000 / top(01) TO conveyora(02)
MSG: I / 00:00:00:000 / top(01) TO dsecu(05)
MSG: I / 00:00:00:000 / top(01) TO es(06)
MSG: I / 00:00:00:000 / top(01) TO ps(07)
MSG: I / 00:00:00:000 / top(01) TO topexec(13)
MSG: I / 00:00:00:000 / conveyora(02) TO enga(03)
MSG: I / 00:00:00:000 / conveyora(02) TO dssca(04)
MSG: D / 00:00:00:000 / dsecu(05) / ... /          0.00000 TO top(01)
MSG: D / 00:00:00:000 / es(06) / 00:00:10:000 /          0.00000 TO top(01)
MSG: I / 00:00:00:000 / ps(07) TO painter(08)
MSG: I / 00:00:00:000 / ps(07) TO color(09)
MSG: I / 00:00:00:000 / ps(07) TO chrome(10)
MSG: I / 00:00:00:000 / ps(07) TO psexec(11)
MSG: D / 00:00:00:000 / topexec(13) / ... /          0.00000 TO top(01)
MSG: D / 00:00:00:000 / enga(03) / ... /          0.00000 TO conveyora(02)
MSG: D / 00:00:00:000 / dssca(04) / ... /          0.00000 TO conveyora(02)

MSG: D / 00:00:00:000 / painter(08) / ... /          0.00000 TO ps(07)

MSG: D / 00:00:00:000 / color(09) / ... /          0.00000 TO ps(07)

MSG: D / 00:00:00:000 / chrome(10) / ... /          0.00000 TO ps(07)

MSG: D / 00:00:00:000 / psexec(11) / ... /          0.00000 TO ps(07)
```

Fig 37. The initialization of the Simulation

```
MSG: X / 00:00:15:015 / ps(07) / ina /          3.00000 TO painter(08)
MSG: * / 00:00:15:015 / ps(07) /          0.00000 TO painter(08)
MSG: D / 00:00:15:015 / enga(03) / ... /          0.00000 TO conveyora(02)
MSG: D / 00:00:15:015 / painter(08) / 00:00:00:000 /          3.00000 TO ps(07)
MSG: D / 00:00:15:015 / conveyora(02) / ... /          0.00000 TO top(01)
MSG: D / 00:00:15:015 / ps(07) / 00:00:00:000 /          3.00000 TO top(01)
MSG: D / 00:00:15:015 / top(01) / 00:00:00:000 /          3.00000 TO Root(00)
MSG: * / 00:00:15:015 / Root(00) /          3.00000 TO top(01)
MSG: * / 00:00:15:015 / top(01) /          3.00000 TO ps(07)
MSG: * / 00:00:15:015 / ps(07) /          3.00000 TO psexec(11)
MSG: D / 00:00:15:015 / psexec(11) / ... /          0.00000 TO ps(07)
MSG: St / 00:00:15:015 / ps(07) TO chrome(10)
MSG: D / 00:00:15:015 / chrome(10) / ... /          0.00000 TO ps(07)
MSG: D / 00:00:15:015 / ps(07) / 00:00:00:000 /          0.00000 TO top(01)
MSG: D / 00:00:15:015 / top(01) / 00:00:00:000 /          0.00000 TO Root(00)
```

Fig 38. The Structural State Transition from PS2 to PS3

```
MSG: D / 00:00:21:500 / es1(15) / 00:00:00:000 /         4.00000 TO top(01)
MSG: D / 00:00:21:500 / enga(03) / 00:00:00:000 /         0.00000 TO conveyora(02)
MSG: D / 00:00:21:500 / conveyora(02) / 00:00:00:000 /         0.00000 TO top(01)
MSG: D / 00:00:21:500 / top(01) / 00:00:00:000 /         4.00000 TO Root(00)
MSG: * / 00:00:21:500 / Root(00) /         4.00000 TO top(01)
MSG: * / 00:00:21:500 / top(01) /         4.00000 TO ps(07)
MSG: * / 00:00:21:500 / ps(07) /         4.00000 TO psexec(11)
MSG: D / 00:00:21:500 / psexec(11) / ... /         0.00000 TO ps(07)
MSG: D / 00:00:21:500 / ps(07) / ... /         0.00000 TO top(01)
MSG: * / 00:00:21:500 / top(01) /         4.00000 TO topexec(13)
MSG: D / 00:00:21:500 / topexec(13) / ... /         0.00000 TO top(01)
MSG: St / 00:00:21:500 / top(01) TO es(16)
MSG: D / 00:00:21:500 / es(16) / 00:00:10:000 /         0.00000 TO top(01)
MSG: D / 00:00:21:500 / top(01) / 00:00:00:000 /         0.00000 TO Root(00)
```

Fig 39. The Structural State Transition from TOP2 to TOP1

Experiment 2

This experiment replaced the Sensor model with GGAD notation equivalent and executed the simulation using the table of the external events presented in the table (scenario 10). Fig. 6 and Fig. 7 display GGAD definitions *of Sensor*, which is used to replace the *Sensor* defined with C++. It was found that the *Sensor* defined in the two ways behaved exactly the same and had the same simulation results. Since the GGAD notation can build equivalent atomic models with less effort than the C++ definitions, it is useful for non-expert modellers. This experiment verified that DS-eCD++ can connect to the GGAD interpreter to execute GGAD models correctly (scenario 10).

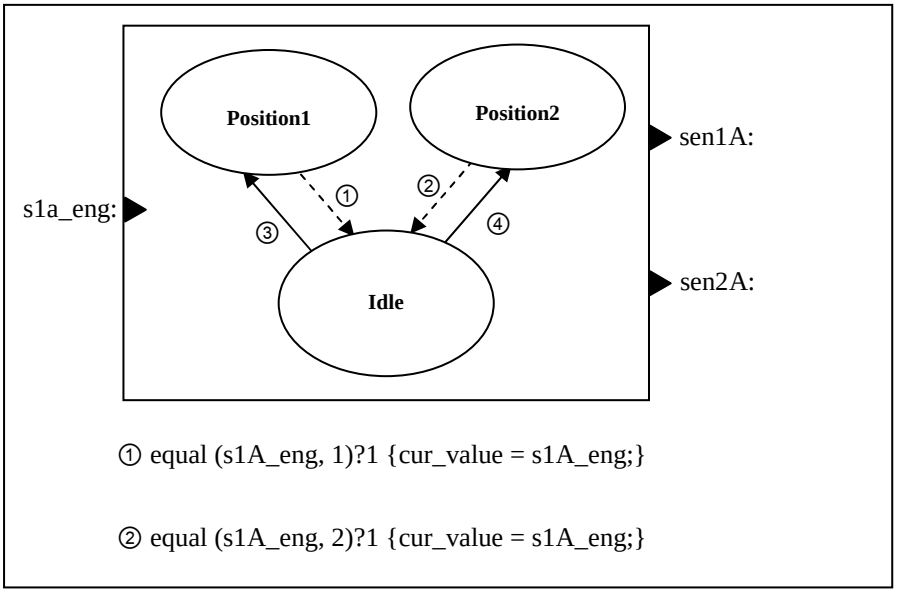


Fig 40. GGAD Graphical Definition of the *Sensor*

```
[Sensor]
in: s1A_eng
out: sen1A sen2A
var : cur_value last_value
state: idle position1 position2
initial: idle
ext: idle position1 equal(s1A_eng, 1)?1 {cur_value = s1A_eng;}
ext: idle position2 equal(s1A_eng, 2)?1 {cur_value = s1A_eng;}
int: position1 idle sen1A!1 {last_value = cur_value;}
int: position2 idle sen2A!1 {last_value = cur_value;}
idle: infinite
position1: 0:0:0:0
position2: 0:0:0:0
```

GGAD notation Definition of *Sensor*

Experiment 3

A different test applied the flat coordinator technique to the dynamic structure simulation of DSAMS (scenario 9). The *Flat Coordinator* helps to improve the simulation performance by flattening the model hierarchy and reducing the number of messages delivered among the models dramatically. The *Flat Coordinator* in DSAMS

enables the atomic models to exchange messages with the *FLATTOP* directly. *FTOPEXEC* is the solo structure agent taking charge of the structural changes on behalf of *FLATTOP*.

Fig. 12 exhibits the processor hierarchy using the flat coordinator. The coordinators of the structure components *PS* and *TOP* are replaced with the flat coordinator. The *FLATTOP* exchanges the messages directly with the atomic models, while *FTOPEXEC* executes the structural changes on behalf of *FLATTOP*. The simulation with the flat coordinator produces the same simulation results as those of Experiment 1, but played a higher simulating performance. The total messages exchanged among the processors in Experiment 1 are 1,104; while the total messages delivered in Experiment 2 are 703. The improvement ratio is (1104 - 703) / 1104 = 36.32%. The comparison of the numbers of messages between the two experiments is shown in Fig.13. The sample messages generated in the simulation of DSAMS using the flat coordinator are presented in the figure.

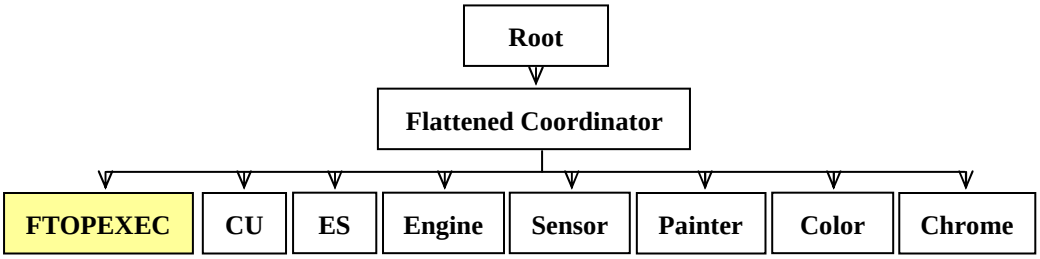


Fig 41. Simulation Hierarchy with a Flat Coordinator

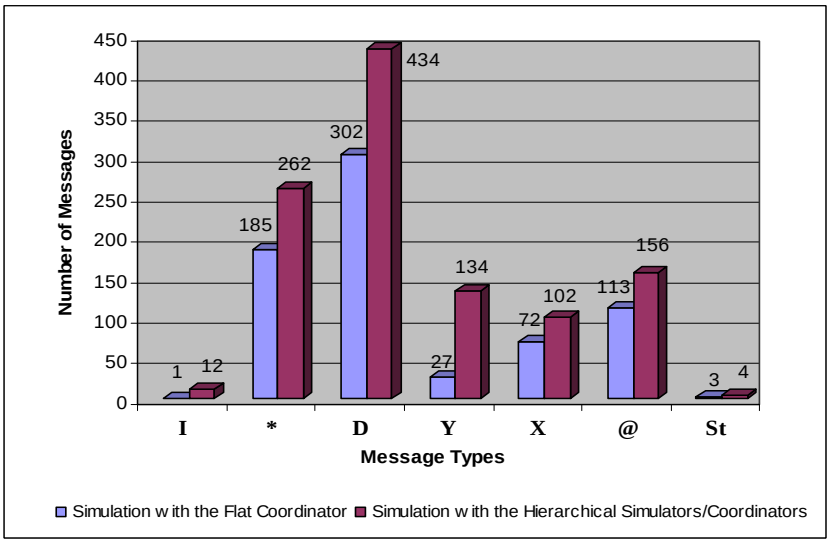


Fig 42. Comparison of the Number of Messages between the two simulation fashions

```
MSG: I / 00:00:00:000 / Root(00) TO flattop(01)

MSG: D / 00:00:00:000 / flattop(01) / 00:00:10:000 /          0.00000 TO Root(00)

…

MSG: D / 00:00:10:000 / es(06) / 00:00:00:000 /          5.00000 TO flattop(01)
```

```
MSG: D / 00:00:10:000 / ftopexec(11) / ... /          0.00000 TO flattop(01)

MSG: St / 00:00:10:000 / flattop(01) TO es1(13)

MSG: D / 00:00:10:000 / es1(13) / 00:00:10:000 /          0.00000 TO flattop(01)

MSG: D / 00:00:10:000 / flattop(01) / 00:00:10:000 /          0.00000 TO Root(00)

…

MSG: D / 00:00:10:505 / enga(03) / ... /          0.00000 TO flattop(01)

MSG: D / 00:00:10:505 / painter(08) / 00:00:00:000 /          2.00000 TO flattop(01)

MSG: D / 00:00:10:505 / flattop(01) / 00:00:00:000 /          2.00000 TO Root(00)

MSG: * / 00:00:10:505 / Root(00) /          2.00000 TO flattop(01)
```

Fig 43. The Message Flows in the Simulation using the flat coordinator

Case 2: Motor Tracing and Replacement System (MTRS)

Description

Motor tracing and replacement system aims to trace and control the moving motor in real time. When the motor fails report its states within the given period, the motor malfunction is considered and a new motor is started to replace the old one. *Controller Unit* is an atomic model to control and trace the target motor. A *motor* is a coupled model containing two atomic models: *Engine* and *Sensor*. The *Engine* drives the motor moving according to the directions indicated by the *Controller Unit*. The *Sensor* senses the position of the motor and reports it to the *Controller Unit*.

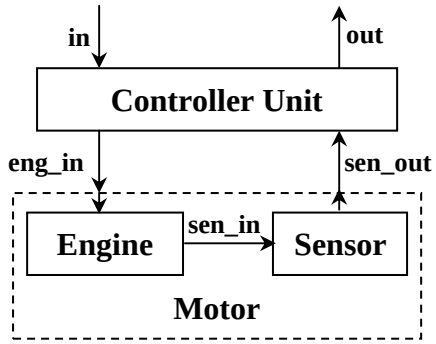


Fig 44. The Scheme of MTRS

Table 4 The Timing Parameters in MTRS

Model Name	Time Variables	Duration	Description
Engine	preparationTime2Start	Time(0, 0, 0, 5)	Time used to start moving a product
	preparationTime2Stop	Time(0, 0, 0, 5)	Time used to stop moving a product
	movingTime	Time(0, 0, 0, 10)	Time used to move from one station to the other

	turningTime	Time(0, 0, 0, 5)	Time used from direction turning
CU	period	Time(0, 0, 0, 40)	Motor should report its states within the period, otherwise a failure is raised

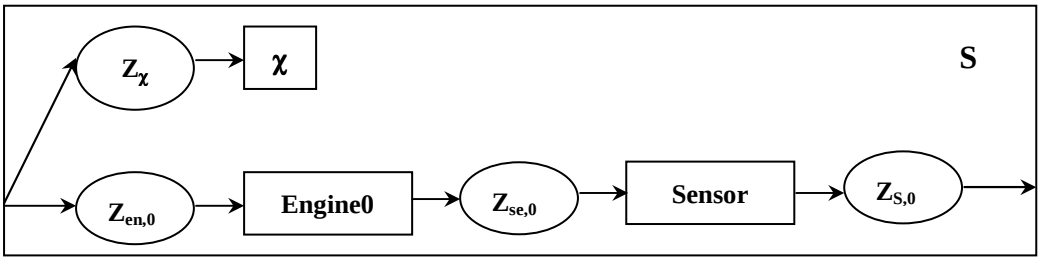
Initially, the motor stops to the north. When the *Controller Unit* receives an instruction (the moving directions with north: 1, east: 2, south: 3, west: 4) from *in*, the Controller Unit passes the instruction to the *Engine* through *eng_in*. The *Engine* drives the motor to move at the indicated direction for up to 10 seconds. If a new instruction comes during the moving of the motor, the moving will be interrupted and act according to the new instruction. Otherwise, the motor stops. The *Sense* senses the position of the motor via its input port *sen_in* and report is to the *Controller Unit* through *sen_out*. The directions that have been acted successfully are output through the *out* port of the *Controller Unit*.

In MTRS, the motor is replaced by another motor when it fails to report the status to the *Controller Unit* within the given period. The *motor* is a structure component, in which the *Engine* and the *Sensor* may be changed by the counterparts. The *TOP* is another structure component, in which a new coupling from the *test* in the *Controller Unit* to the *test* in *motor* is added.

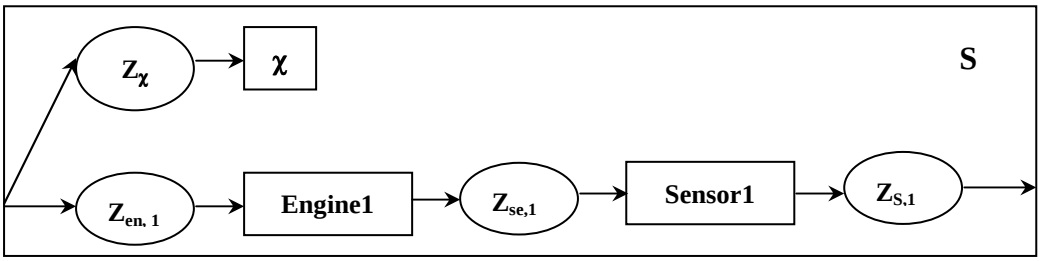
Formal Specifications

The formal specification of the structure component in MTRS is presented based on the DSDEVS formalism. In the figure, S represents the structure component *motor*. The Engine and the Sensor in the fig. a) are replaced by the Engine1 and the Sensor1 in the

fig. b). The fig. c) is the formal specification of *motor*. The figure xx shows the formal specification of the *TOP*. The structural change of the interface of the coupled model lies in the transition function between the *Controller Unit* and the structure component *motor* ($Z_{mo,0} \rightarrow Z_{mo,1}$).



a)

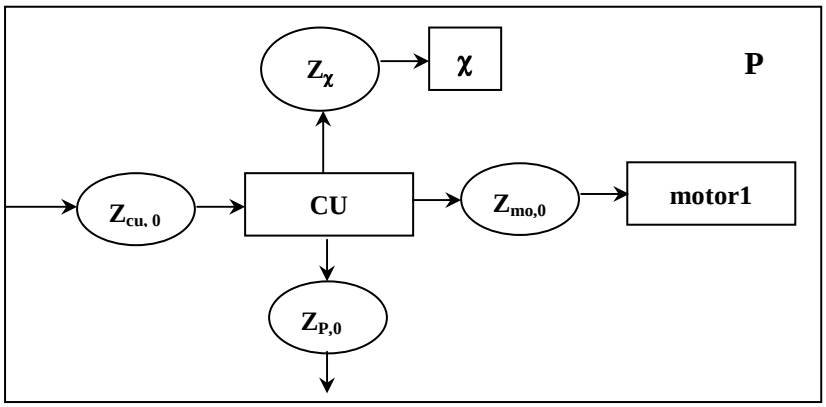


b)

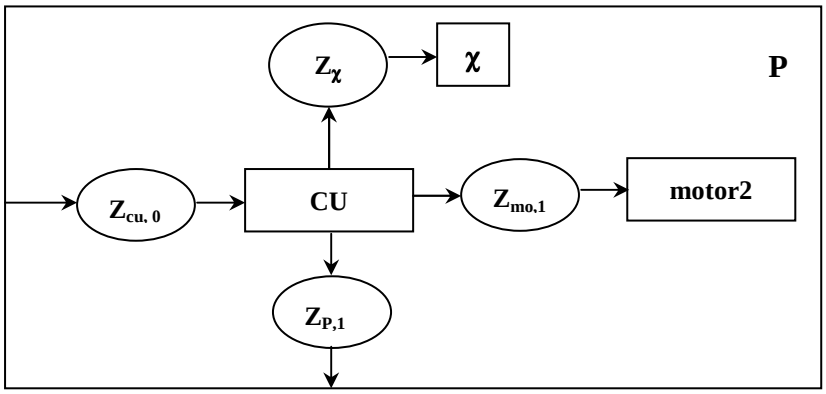
$S = (X_S, Y_S, \chi, M_\chi)$ $X_S = \{1, 2, 3, 4\}$ $Y_S = \{1, 2, 3, 4\}$ $M_\chi = \{X_\chi, s_{0,\chi}, S_\chi, \delta_\chi, \tau_\chi\}$
$X_\chi = \{\text{Timeout}\}$ $S_\chi = \{s_{0,\chi}, s_{1,\chi,}\}$ $\tau\, s_{0,\chi} = \tau\, s_{1,\chi} = \text{Inf}$ $\delta_\chi\,(s_{0,\chi}, e, \text{Timeout}) = s_{1,\chi}$
$\gamma\,(s_{0,\chi}) = \{D_0, \{M_{i,0}\}, \{I_{i,0}\}, \{Z_{i,0}\}\}$ $\gamma\,(s_{1,\chi}) = \{D_1, \{M_{i,1}\}, \{I_{i,1}\}, \{Z_{i,1}\}\}$
$D_0 = \{\text{Engine, Sensor}\}$ $D_1 = \{\text{Engine1, Sensor1}\}$
$M_{en,0} = \{ X_{en}, s_{0,en}, S_{en}, Y_{en}, \delta_{en}, \lambda_{en}, \tau_{en}\}$
$M_{se,0} = \{ X_{se}\, s_{0,se}, S_{se}, Y_{se}, \delta_{se}, \lambda_{se}, \tau_{se}\}$
$M_{en1,1} = \{ X_{en1}, s_{0,en1}, S_{en1}, Y_{en1}, \delta_{en1}, \lambda_{en1}, \tau_{en1}\}$
$M_{se1,1} = \{ X_{se1}\, s_{0,se1}, S_{se1}, Y_{se1}, \delta_{se1}, \lambda_{se1}, \tau_{se1}\}$
$I_{en,0} = \{S\}$ $I_{en1,1} = \{S\}$
$I_{se,0} = \{\text{Engine}\}$ $I_{se1,1} = \{\text{Engine1}\}$
$I_{\chi,0} = I_{\chi,1} = \{S\}$
$Z_{\chi,0} = Z_{\chi,1} = Z_\chi$ and Z_χ: $X_S \rightarrow X_\chi$ $Z_{en,0}$: $X_{S,0} \rightarrow X_{en,0}$ $Z_{en1,1}$: $X_{S,1} \rightarrow X_{en1,1}$ $Z_{se,0}$: $X_{en,0} \rightarrow X_{se,0}$ $Z_{se1,1}$: $X_{en,1} \rightarrow X_{se1,1}$
$Z_{S,0}\, Y_{se,0} \rightarrow Y_{S,0}$ $Z_{S,1}\, Y_{se,1} \rightarrow Y_{S,1}$

c)

Fig 45. a) The Motor1 is controlled by the Controller Unit b) The Motor2 is controlled by the Controller Unit c) Formal specification of the structure component *motor*



a)



b)

$P = (X_p, Y_p, \chi, M_\chi)$ $X_p = \{1, 2, 3, 4\}$ $Y_p = \{1, 2, 3, 4\}$ $M_\chi = \{X_\chi, s_{0,\chi}, P_\chi, \delta_\chi, \tau_\chi\}$
$X_\chi = \{\text{Timeout}\}$ $S_\chi = \{s_{0,\chi}, s_{1,\chi,}\}$ $\tau\, s_{0,\chi} = \tau\, s_{1,\chi} = \text{Inf}$ $\delta_\chi\,(s_{0,\chi}, e, \text{Timeout}) = s_{1,\chi}$
$\gamma\,(s_{0,\chi}) = \{D_0, \{M_{i,0}\}, \{I_{i,0}\}, \{Z_{i,0}\}\}$ $\gamma\,(s_{1,\chi}) = \{D_1, \{M_{i,1}\}, \{I_{i,1}\}, \{Z_{i,1}\}\}$
$D_0 = \{\text{Controller Unit, motor}\}$ $D_1 = \{\text{Controller Unit, motor}\}$
$M_{cu,0} = M_{cu,1} = \{ X_{cu}, s_{0,cu}, S_{cu}, Y_{cu}, \delta_{cu}, \lambda_{cu}, \tau_{cu}\}$
$M_{mo,0} = M_{mo,1} = \{ X_{mo}\, s_{0,mo}, S_{mo}, Y_{mo}, \delta_{mo}, \lambda_{mo}, \tau_{mo}\}$
$I_{cu,0} = \{P\}$ $I_{cu1,1} = \{P\}$
$I_{mo,0} = \{\text{Controller Unit}\}$ $I_{mo,1} = \{\text{Controller Unit}\}$
$I_{\chi,0} = I_{\chi,1} = \{\text{Controller Unit}\}$
$Z_{\chi,0} = Z_{\chi,1} = Z_\chi$ and Z_χ: $X_P \rightarrow X_\chi$ $Z_{cu,0}$: $X_{P,0} \rightarrow X_{cu,0}$ $Z_{cu,1}$: $X_{P,1} \rightarrow X_{cu,1}$ $Z_{mo,0}$: $X_{cu,0} \rightarrow X_{mo,0}$ $Z_{mo,1}$: $X_{cu,1} \rightarrow X_{mo,1}$
$Z_{P,0}\, Y_{cu,0} \rightarrow Y_{P,0}$ $Z_{P,1}\, Y_{cu,1} \rightarrow Y_{P,1}$

c)

Fig 46. a) The Controller Unit Connected with Motor1. b) The Controller Unit Connected with Motor2. c) The Formal Specification of the Structure Component *TOP.*

Model Definitions

The model definition of MTRS is shown in the Fig. 18. The model structure described in the group **[Topupdate1]** is the alternative model structure of the structure component *TOP*. The alternative model structure of *motor* is presented in the group [motorupdate1]. For each group, the structural command is designated to take as a code name of the corresponding model structure.

```
[top]
components : cu@ECU motor topexec#ExecTop
in : in   out : out   Scomm : topstruc1
Link : in in@cu   Link : eng_in@cu in@motor
Link : out@motor sen_out@cu   Link : out@cu out
[topupdate1]
components : cu@ECU motor
in : in   out : out Scomm : topstruc2
Link : in in@cu Link : eng_in@cu in@motor Link : eng_test@cu test@motor
Link : out@motor sen_out@cu Link : out@cu out
```

Fig 47. Model Definition of MTRS

The MTRS Experiments Using DS-eCD++

Experiment 1

The MTRS is simulated in real time with the external events in the following. Initially, motor1 is controlled by the Controller Unit including the *Engine* and the *Sensor*. The *Engine* contains one input port *in*. A soft fault was set in the *Engine*, which makes the *Engine* cannot report its status when the direction is west (value = 4). Motor2 including the *Engine1* and the *Sensor1* starts to replace motor1. Suppose motor2 containes two input ports *in* and *test*. The instructions from the *Controller Unit* are received via *in*, while *test* allows receiving more information from the *Controller Unit*.

A nested structural change process was involved in the experiment. MotorEXEC replaceed the components and the couplings of motor1 with the counterparts of motor2 on behalf of the structure component *motor*. TOPEXEC took charge of building a new coupling between Controller Unit and motor2 on behalf of the structure component *TOP*.

Table 5 The Table of the External Events

Event time	Deadline	Input port	Output port	Value
00:00:01:500	00:00:01:535	in	out	2
00:00:04:500	00:00:04:535	in	out	3
00:00:10:500	00:00:10:535	in	out	1
00:00:15:000	00:00:15:035	in	out	4
00:00:20:000	00:00:20:035	in	out	3

At 00:00:15:000, an instruction was sent to the *Controller Unit*. The instruction comes to the *Engine* of at 00:00:15:010. Due to the soft fault in the *Engine*, the instruction was

not executed properly. Therefore, the *Controller Unit* did not receive the report from the *Sensor*. The Controller Unit considered the malfunction of the motor1. At 00:00:15:040, the Controller Unit sent an output with value 9 indicating the failure of the motor1. A structural change request is raised by the Controller Unit once upon the failure report. The structural change request caused the structural changes both in the *motor* and the *TOP* as in the figure 19. The structural change in the motor was executed first. *MotorEXEC* replaces the couplings and the atomic models in the *motor*. The structural change operations in *MotorEXEC* were:

1. DelLink (“motor”, link1) (link1: in in@engine)
2. DelLink (“motor”, link2) (link2 : out@engine in@sensor)
3. DelLink (“motor”, link3) (link3 : out@sensor out)
4. DelModel (“motor”, “Engine”)
5. DelModel (“motor”, “Sensor”)
6. AddModel (“motor”, “Engine1”)
7. AddModel (“motor”, “Sensor1”)
8. AddInputPort (“motor”, “test”)
9. AddLink (“motor”, link1) (link1: in in@engine1)
10. AddLink (“motor”, link2) (link2 : test test@engine)
11. AddLink (“motor”, link3) (link3 : out@engine1 in@sensor1)
12. AddLink (“motor”, link4) (link4 : out@sensor1 out)

After MotorEXEC finished the structural change execution, the structural change message was delivered to the *TOP*. TOPEXEC adds a coupling in the *TOP*. TOPEXEC implements the following structural change operations:

1. AddLink (“TOP”, link) (link : eng_test@CU test@Motor)

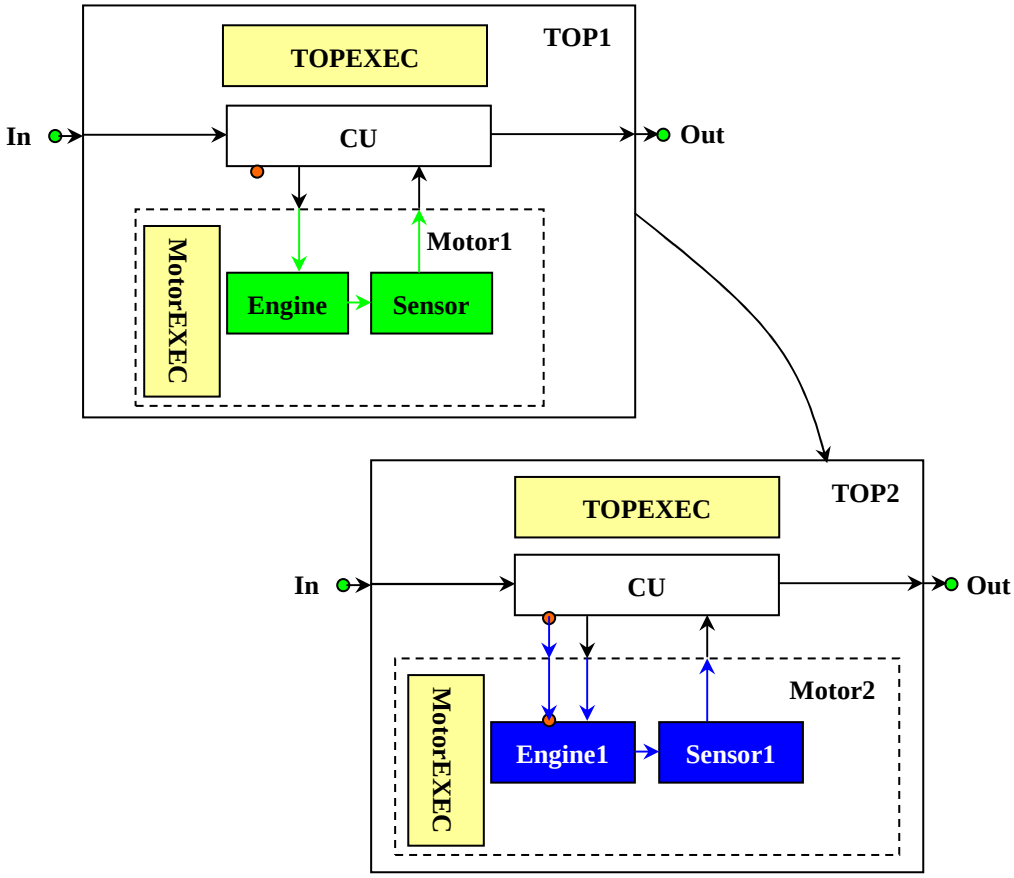


Fig 48. The Structural Changes in MTRS

The Table 6 reflects the simulation results. In this experiment, the first three external events have been executed and reported correctly. The fourth event is input at 00:00:15:000 with the direction 4. The *Engine1* failed to report the action to this instruction and caused the structural changes in MTRS.

Three structural change scenarios were involved in the experiment. The motor1 was replaced by the motor2. That is to say, the coupled model *motor* has been replaced by the new one (scenario 6) although the same name is used in the simulation. A nested structural change process (scenario 7) involved the two structure components – the *motor* and the *TOP*. The new coupling from the output port *eng_test* of the Controller Unit to the input port *test* of the *motor* changes the interface of the *motor* to the

Controller Unit (scenario 9). At 00:00:15:040, the value 9 is output indicating the malfunction of the controlled motor. And then a structural change request was raised in the Controller Unit. MotorEXEC and TOPEXEC complete the structural changes for their structure component respectively. As a result, the motor2 replaced the motor1 to receive the instructions from the Controller Unit. At real time 00:00:20:020, the motor2 reacted the instruction arriving at 00:00:20:000 and sent its status correctly.

Table 6 The Simulation Results in the MTRS Simulation

Output time	Deadline	Result	Output Port	Value
00:00:01:510	00:00:01:535	Succeed	out	2
00:00:04:510	00:00:04:535	Succeed	out	3
00:00:10:520	00:00:10:535	Succeed	out	1
00:00:15:040	00:00:15:035	Not Succeed	out	9
00:00:20:020	00:00:20:035	Succeed	out	3

The following message flows presents the detailed logging information. The Fig. 20 shows the failure report of the *Controller Unit*. At 00:00:15:040, the *Controller Unit* sent 9 to the output port *out*. The nested structural change processor followed the failure report. The lines with shadow in the Fig. 21 indicate that the structural change messages were sent to the structure agents of the structure component and the structure agents returned done messages to the structure components. From the figure, we can see that the structural change in the *motor* was executed first. The *Engine1* and the *Sensor1* join the simulation. Afterward the structural change in the *TOP* was implemented. At 00:00:20:000, the new coupling was used to deliver messages.

```
MSG: @ / 00:00:15:040 / Root(00) TO top(01)

MSG: @ / 00:00:15:040 / top(01) TO cu(02)

MSG: X / 00:00:15:040 / cu(02) / out /          0.00000 TO top(01)
```

Fig 49. Failure Report from the Controller Unit

```
MSG: D / 00:00:15:040 / top(01) / 00:00:00:000 /          0.00000 TO Root(00)

MSG: * / 00:00:15:040 / Root(00) /          0.00000 TO top(01)

MSG: * / 00:00:15:040 / top(01) /          0.00000 TO cu(02)

MSG: D / 00:00:15:040 / cu(02) / 00:00:00:000 /          2.00000 TO top(01)

MSG: D / 00:00:15:040 / top(01) / 00:00:00:000 /          2.00000 TO Root(00)

MSG: * / 00:00:15:040 / Root(00) /          2.00000 TO top(01)

MSG: * / 00:00:15:040 / top(01) /          2.00000 TO motor(03)

MSG: * / 00:00:15:040 / motor(03) /          2.00000 TO motorexec(06)

MSG: D / 00:00:15:040 / motorexec(06) / ... /          0.00000 TO motor(03)
```

Fig 50. The Message Flows in the Nested Structural Change Process

```
MSG: X / 00:00:20:000 / Root(00) / in /          3.00000 TO top(01)

MSG: * / 00:00:20:000 / Root(00) /          0.00000 TO top(01)

MSG: X / 00:00:20:000 / top(01) / in /          3.00000 TO cu(02)

MSG: * / 00:00:20:000 / top(01) /          0.00000 TO cu(02)

MSG: D / 00:00:20:000 / cu(02) / 00:00:00:000 /          0.00000 TO top(01)

MSG: D / 00:00:20:000 / top(01) / 00:00:00:000 /          0.00000 TO Root(00)

MSG: @ / 00:00:20:000 / Root(00) TO top(01)

MSG: @ / 00:00:20:000 / top(01) TO cu(02)
```

```
MSG: * / 00:00:20:000 / top(01) /          0.00000 TO cu(02)

MSG: * / 00:00:20:000 / top(01) /          0.00000 TO motor(03)

MSG: D / 00:00:20:000 / cu(02) / 00:00:00:040 /          0.00000 TO top(01)

MSG: X / 00:00:20:000 / motor(03) / in /          3.00000 TO engine1(10)

MSG: X / 00:00:20:000 / motor(03) / test /          3.00000 TO engine1(10)
```

Fig 51. The Message Flows Using the New Coupling At 00:00:20:000

Chapter 6 Conclusions and Future Work

Based on the proposed FDSDE algorithm and the P-DEVS real time simulation engine, DS-eCD++ is developed to be an advanced DEVS-based real time experimental environment supporting both the dynamic structure function and the real-time simulation. This work advanced the functionality of eCD++ to meet the rigorous requirements in modeling and design of real time embedded systems.

An advanced simulation engine combining FDSDE and P-DEVS real time simulation engine is defined. The Root Coordinator, the Coordinator and the Simulator, which constitute the real time simulation engine in eCD++, are redefined to fit dynamic structure and real time simulations. The concept of structure component is introduced in DS-eCD++ to represent the coupled models which are subject to structural changes. Each structure component is furnished with a structure agent to specify the structural changes for the structure component. A new abstract simulator RevSimulaor is introduced to generate the model behaviours of structure agents. Moreover, two typical message passing scenarios, one structural change process and nested structural change process, are presented to exhibit how the message flow among the processor at a global view. In the message passing scenarios, the simulation phases are clearly identified.

DS-eCD++ takes advantage of the major four software components in eCD++. However, the revisions have been made to accommodate to the dynamic structural changes in the real time simulation. The modifications of the Main Simulator, The Modeling Subsystem, The Simulation Subsystem and The Messaging Subsystem are

explained. Moreover, the structure component identification and the structural change operation in structure agents are highlighted to present how the dynamic structure is implemented. The functionalities of DS-eCD++ are discussed showing the expected performance.

In order to verify the logics and implementations of the algorithm, a series of experiments are conducted. The devised structural change scenarios are firstly enumerated presenting a functional profile of DS-eCD++. The cases corresponding to the different structural change scenarios are implemented and analyzed in the following section. It has been verified that DS-eCD++ not only performs real time simulation in the different structural change scenarios but also is able to implement GGAD notation models and the simulation with the flat coordinator. We even expect that DS-eCD++ can further serves as a DEVS-based Real-Time experimental environment for real time embedded systems modeling and design. Besides enabling the implementation of the hybrid software and hardware systems and the seamless transformation from the simulation stage to the design stage of real-time systems, DS-eCD++ allows defining both the structural changes and the behavioural changes of systems therefore achieve high flexibility and reliability of the real time embedded systems.

6.1. Future Works

We have proved that FDSDE algorithm performs well in the DS-eCD++ environment. However, the further studies should be investigated to improve the functionality and performance of DS-ECD++.

1. Performance evaluation. With the devised structural change scenarios, the functionality is the major concern of this work. Whereas, in order to achieve the critical requirements of the real-time embedded systems, the performance evaluation is another key issue to be conducted. The performance evaluating metrics are necessary to provide an evaluating environment. The performance experiments should be conducted to ensure the sensitive detection of the structural change conditions and the fast response to the conditions.
2. Real environment examination. The case studies are conducted in the virtual environment. The real environment calls for more rigorous timing and memory requirements to maintain the reliability of the systems. Further experiments in real environment should be done to test the capabilities of handling the real situations in DS-eCD++.
3. Algorithm optimization. More experiments, especially the structural changes in the complex real-time embedded systems, should be implemented to refine the implementation of the algorithm. More structural changes should be tried to test the accuracy of the implementation.
4. Distributed and parallel implementations [Liu07]. DCD++ and PCD++ realize the DEVS simulation in distributed and parallel environment. The structural changes in the distributed and parallel environments may span different simulation nodes. Under the conditions, the structural changes may require several structure agents to cooperate together to implement the structural change tasks. The coordinating messages should be handled in the structure agent processors. The structural

changes in the advanced simulation environment should be further explored in the future research.

References

1. [Bar94] Barros, F.J.; M. T. Mendes and B.P.Zeigler. “Variable DEVS – Variable Structure Modeling Formalism: An Adaptive Computer Architecture Application”. AI, Simulation, and Planning in High Automomy Systems, “Distributed Interactive Simulation Environments”. Proceedings of the Fifth Annual Conference. 1994.
2. [Bar95]Barros, F.J. 1995. “Dynamic Structure Discrete Event System Specifications: A New Formalism for Dynamic Structure Modeling and Simulation”. In the Proceedings of the 1995 Winter Simulation Conference, pp.781-785. Arlington, USA.
3. [Bar97]Barros, F.J. 1997. “Modelling Formalisms for Dynamic Structure Systems”. ACM Transactions on Modeling and Computer Simulation, Vol. 7, No. 4, pp. 501-515.
4. [Bar98a]Barros, F. J.; Zeigler, B. P.; Fishwick, P. A. “Multimodels and Dynamic Structure Models: An Integration of DSDE/DEVS and OOPM”, Proceedings of the 1998 Winter Simulation Conference.
5. [Bar98b]Barros, F.J. 1998. “Abstract Simulators for the DSDE Formalism”. In the Proceedings of the 1998 Winter Simulation Conference, pp.407-412. Washington DC, USA.
6. [Bar01]Barros, F.J. “Representation of Dynamic Structure Discrete Event Models: A Systems Theory Approach”, Discrete event modeling and simulation

technologies: a tapestry of systems and AI-based theories and methodologies pages 167-185, Springer-Verlag, New York, 2001.

7. [Bar03a]Barros, F.J and Zeigler. B.P. “Model Interoperability in the Discrete Event Paradigm: Representation of Continuous Models”. In Modeling and Simulation; Theory and Practice, G.A. Bekey e B.Y.Ko, January, pp.103-126, 2003
8. [Bar03b] Barros, F.J. “Dynamic Structure Multiparadigm Modeling and Simulation” ACM Transactions on Modeling and Computer Simulation, Vol. 13, No. 3, July 2003, pp. 259-275
9. [Bar05] Barros, F.J. 2005. “Requirements for Modeling and Simulation of Self-Adaptive Systems: A Hierarchical and Modular Approach”. Proceedings of the 16th International Workshop on Database and Expert Systems Applications (DEXA’05).
10. [Cho94] Chow, A. C.; Zeigler, B. “Parallel DEVS: A parallel, hierarchical, modular modeling formalism”. Proceedings of the Winter Computer Simulation Conference. Orlando, FL. USA. 1994.
11. [Cho00] Cho, Y. K., B.P. Zeigler, H. J. Cho, H. S. Sarjoughian, and S. Sen. 2000. “Design Considerations for Distributed Real-Time DEVS”. AIS 2000. Tucson, USA.
12. [Cho01a] Cho. Y.K., Zeigler, B.P. and Sarjoughian, H.S. “Design and Implementation of distributed real-time DEVS/CORBA”. IEEE International Conference on System, Man, and Cybernetics. Tucson, AZ. October 2001.

13. [Cho01b] Cho, S., and T.G. Kim. 2001. “Real Time Simulation Framework for RT-DEVS Models”. *Transactions of the Society for Computer Simulation International.* Vol. 18, No. 4, pp. 203 – 215.
14. [Gli04a]Glinsky, E., and G. Wainer. “Modeling and Simulation of Systems with Hardware-in-the-loop”. In the Proceedings of the 2004 Winter Simulation Conference. Washington DC, USA. 2004.
15. [Gli04b]Glinsky, E., and G. Wainer. “Model-Based Development of Embedded Systems with RT-CD++”. In the Proceedings of the WIP session, IEEE Real-Time and Embedded Technology and Applications Symposium. Toronto, Canada. 2004.
16. [Her00]Herrman, J.W.; E. Lin; B. Ram and S. Sarin. “Adaptable simulation models for manufacturing”. Proceedings of the 10th International Conference on Flexible Automation and Intelligent Manufacturing, College Park, Maryland, USA, Volume 2, pp. 989-995. 2000.
17. [Hon97] Hong, J., H. Song, T.G. Kim, and K.H. Park. A Real-time Discrete Event System Specification Formalism for Seamless Real-time Software Development. *Discrete Event Dynamic systems: Theory and Applications,* Vol. 7, No. 4, pp. 355-375. 1997.
18. [HuX03] Hu, X.; Zeigler, B.P. and Mittal, S. “Dynamic Reconfiguration in DEVS Component-Based Modeling and Simulaiton”, Simulation: Transactions of the Society of Modeling and Simulation International, November 2003.

19. [HuX04]Hu, X.; Zeigler, B. P. “Model Continuity to Support Software Development for Distributed Robotic Systems: A Team Formation Example”. Journal of Intelligent and robotic Systems 39: pp. 71- 87. 2004.
20. [HuX05a] Hu, X.; Ganapathy N.; Zeigler, B. P. “Robots in the loop: Supporting an Incremental Simulation-based Design Process”. IEEE International Conference on Systems, Man, and Cybernetics, October, 2005.
21. [HuX05b] Hu. X.; Zeigler, B. P. “Model Continuity in the Design of Dynamic Distributed Real-Time Systems”. IEEE Transactions on Systems, Man, and Cybernetics – Part A: Systems and Humans, Vol., 35, NO. 6, November, pp. 867-878. 2005.
22. [HuX05c] Hu, X.; Zeigler, B. P. “A Simulation-based Virtual Environment to Study Cooperative Robotic Systems”. Integrated Computer-Aided Engineering 12 (2005) IOS Press. pp. 353 – 367.2005.
23. [HuX05d] Hu. X.; B.P. Zeigler, and S. Mittal. “Variable Structure in DEVS Component-Based Modeling and Simulation”. Simulation: Transactions of the Society for Modeling and Simulation International, Vol. 81, No. 2, pp. 91-102, 2005.
24. [Kim01] Kim, T.G., S.M. Cho, and W.B. Lee. DEVS Framework for Systems Development. *Discrete Event Modeling & Simulation: Enabling Future Technologies.* Springer-Verlag. 2001.

25. [Kop00] Kopetz, H. “Software Engineering for Real-Time: A Roadmap”. In the Proceedings of the Conference on the Future of Software Engineering, pp.201-211, Limerick, Ireland. 2000.

26. [LiL03] Li, L.; Pearce, T.W. and Wainer, G. “Interfacing Real-time DEVS models with a DSP platform”. In proceedings of the Industrial Simulation Symposium. Valencia, Spain. 2003

27. [Liu03] Liu, S.; J. Wei; and W. Xu. “Towards Dynamic Process with Variable Structure by Reflection”. Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC’03). 2003.

28. [Liu06] Liu, Q. “Distributed Optimistic Simulation of DEVS and Cell-DEVS Models with PCD++”. M. A. Sc. Thesis. Carleton University. Canada. 2006.

29. [Liu07] Liu, Q.; and Wainer, G. “Improving CD++ Parallel simulation engine”. In Proceedings of the 39th IEEE/SCS Annual Simulation Symposium. Norfo7lk, VA. USA. 2007

30. [Mac04] MacSween, P. and Wainer, G. “On the construction of Complex models using reusable Components”, In Proceedings of SISO Spring Interoperability Workshop. Arlington, VA. U.S.A. 2004

31. [Mit06] Mittal, S.; E. Mak and J. J. Nutaro. “DEVS-Based Dynamic Model Reconfiguration and Simulation Control in the Enhanced DoDAF Design Process”. The Society for Modeling and Simulation International. JDMS, Vol. 3, Issue 4, pp. 95-123. 2006.

32. [Paw96] Pawletta, T.; Lampe, B.; Pawletta, S. And Drewelow, W. “A New Approach for Simulation of Variable Structure Systems”. In Proceedings of the 41th Conference, KoREMA, Aagreb, Croatia. 1996.

33. [Pea03] Pearce, T. W.; “Simulation-Driven Architecture in the Engineering of Real-Time Embedded Systems”. Real-Time System Symposium, Work-in-Progress Session. Cancun, Mexico. 2003.

34. [Mad07] Madhoun, R. and Wainer, G. “Performance analysis of Web-Based CD++”. In proceedings of DEVS Symposium 2007. Norfolk, VA. 2007.

35. [Sha06] Shang, H..; Wainer, G. “A Simulation Algorithm for Dynamic Structure DEVS Modeling”. Proceedings of the 2006 Winter Simulation Conference. Monterey, CA. USA.

36. [Sha07] Shang. H.; Wainer, G. “A Flexible Dynamic Structure DEVS Algorithm towards Real-Time Systems”. Proceedings of the 2007 Summer Computer Simulation Conference. Sandiego, CA. USA

37. [Uhr93] Uhrmacher, A.M. “Variable Structure Models: Autonomy and Control – Answers from Two Different Modeling Approaches”. Proc. AI, Simulation, and Planning in High Autonomy Systems. IEEE Computer Society Press, 1993, pp. 133-139

38. [Uhr01] Uhrmacher, A. M. 2001. “Dynamic Structure in Modeling and Simulation: A Reflective Approach”. ACM Transactions on Modeling and Computer Simulation. Vol. 11, No. 2, pp. 206-232.

39. [Uhr04] Uhrmacher, A.M., and J. Himmeelspach. 2004. "Processing dynamic PDEVS models". In the Proceedings of the IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS'04). Volenlam, Netherlands.

40. [Uhr06] Uhrmacher, A. M.; Himmelspach, J.; Rohl, M.; and Ewald, R. "Introducing Variable Ports and Multi-couplings for Cell Biological Modeling in DEVS". Proceedings of the 2006 Winter Simulation Conference, Monterey, CA., USA.

41. [Wai02] Wainer, G. 2002. "CD++: a toolkit to define discrete-event models". In Software, Practice and Experience. Wiley. Vol. 32, No.3, pp. 1261-130.

42. [Wai05] Wainer, G., E. Glinsky, and P. Macsween. 2005. Model-Driven Architecture of Real-Time Systems. Model-driven Software Development - Volume II of Research and Practice in Software Engineering. S. Beydeda and V. Gruhn eds., Springer-Verlag.

43. [Wai06] Wainer, G. "Applying Cell-DEVS Methodology for Modeling the Environment". In *Simulation, Transactions of the SCS*. Vol. 82, No. 10, 635-660. October 2006

44. [YuJ07] Yu, J. and Wainer, G. "E-CD++: a tool for modeling embedded real-time applications". In proceedings of the 2007 SCS Summer Computer Simulation Conference. San Diego, CA. 2007

45. [Zei76] Zeigler B.P. Theory of modeling and simulation. John Wiley Editor, New York, 1976.

46. [Zei84] Zeigler B.P. Multifaceted Modeling and Discrete Event Simulation. Academic Press, London UK,1984.

47. [Zei86] Zeigler, B. P. 1986. Toward a simulation methodology for variable structure modeling. In Modelling and Simulation Methodology in the Artificial Intelligence Era, M. Elzas, B. Zeigler, and T. Oeren, Eds. Elsevier Sci. Pub. B. V., Amsterdam, the Netherlands, 195–210.

48. [Zei89] Zeigler, B. P. 1989. Concepts for distributed knowledge maintenance in variable structure models. In Modelling and Simulation Methodology - Knowledge Systems Paradigm, B. Zeigler, M. Elzas, and T. Oeren, Eds. Elsevier North-Holland, Inc., Amsterdam, the Netherlands, 45–54.

49. [Zei91] Zeigler B. P., Kim, T. G.,and Lee, C. 1991. Variable structure modeling methodology: An adaptive computer architecture example. Trans. Soc. Comput. Simul. 7, 4 (Dec. 1990), 291–318.

50. [Zei93] Zeigler, B.; Jimwoo, K. “Extending the DEVS-Scheme Knowledge-Based Simulation Environment for Real-Time Event-Based Control”. IEEE Transactions on Robotics and Automation, Vol., 9, NO., 3, JUNE, pp.351-356, 1993.

51. [Zei00] Zeigler, B.P., Kim, T.G. and Praehofer, H. “Theory of Modeling and Simulation”, 2nd Edition, Academic Press. New York, NY, 2000.

52. [Zei02] Zeigler, B.; Sarjoughian, H. S. “DEVS Component-Based M&S Framework: An Introduction”. Proceedings of AI, Simulation and Planning in High Autonomy. 2002

53. [Zei03] Zeigler, B. “DEVS Today: Recent Advances in Discrete Event-Based Information Technology”. Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS’03). 2003.