MODEL-DRIVEN DESIGN AND VALIDATION OF SERVICE ORIENTED ARCHITECTURE BASED ON DEVS SIMULATION FRAMEWORK

Article · July 2014

DOI: 10.29268/stsc.2014.2.3.2

CITATIONS
2

READS
78

5 authors, including:

Jianpeng Hu
Shanghai University of Engineering Science
13 PUBLICATIONS **45** CITATIONS

SEE PROFILE

Linpeng Huang
Shanghai Jiao Tong University
170 PUBLICATIONS **738** CITATIONS

SEE PROFILE

MODEL-DRIVEN DESIGN AND VALIDATION OF SERVICE ORIENTED ARCHITECTURE BASED ON DEVS SIMULATION FRAMEWORK

Jianpeng Hu[1,2], Linpeng Huang[2], Renke Wu[2], Bei Cao[2], Xuling Chang[2]

[1]College of Electrical and Electronic Engineering, Shanghai University of Engineering Science, Shanghai, China; [2]Dept. of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China

mr@sues.edu.cn, huang-lp@sjtu.edu.cn, sjtuwrk@sjtu.edu.cn, caobei.sjtu@gmail.com, changxl@sjtu.edu.cn

Abstract

It is very important to validate functional requirements and evaluate non-functional requirements in earlier design phase of a Service Oriented Architecture (SOA) by executable modeling methodology. To make SOA executable, basically, most of the proposed approaches can be divided into two categories: formalism-based ones and model-driven ones, which both have the advantages and limitations. In this paper, we take advantage of both formalism-based and model-driven methodologies to specify a unified model-driven design and validation approach to SOA. This approach bridges generic service design and universal simulation paradigm with formal bases and practical implementation. To achieve this goal, we first extend the DEVS modeling language (DEVSML) to support nondeterministic state transition and enhance its capability to describe complex behavior of systems. Then we provide an automated transformation process using Extended DEVSML as a model transformation intermediary to bring together Model Driven Service Engineering (MDSE) with Service oriented architecture Modeling Language (SoaML) and Modeling & Simulation (M&S) methodology based on Discrete Event System Specification (DEVS). To demonstrate the applicability of this approach, we introduce an aircraft docking process in an airport scenario as the case study.

Keywords: Model Driven Service Engineering; Executable modeling; SOA; DEVS; Simulation; System of Systems

1. INTRODUCTION

Nowadays, enterprise information systems have become large-scale, composite systems, consisting of software and hardware components, which should be effectively combined to ensure system efficient operation. Service Oriented Architecture (SOA) is an attractive architecture paradigm for developing enterprise scale distributed software systems. It emphasizes loosely coupled, protocol independent distributed system development with the "software as service" concept. Many SOA-based systems show System of Systems (SoS) characteristics including large-scale, consisting of software and hardware components, and cooperative processes among independent systems. The term Service Engineering first appeared in the 1990s as a discipline in business studies describing a new approach for creating and managing business services. As the underlying technology gradually matured, service development or service engineering has received more attention. A general challenge for Service Engineering is to enable service modules to be rapidly developed, and to be deployed and composed without undesirable service interactions. This is a formidable problem and a very challenging and attractive application area for Model Driven Architecture (MDA). SOA has been promoted for many years without a specific language that supports modeling services.

In order to meet this requirement, the Service oriented architecture Modeling Language (SoaML) (Object Management Group, 2009) was specified. The goals of SoaML are to support the activities of service modeling and design and to fit into an overall model-driven development process. This is done in such a way as to support the automatic generation of derived artifacts following an MDA-based approach, and it is also more convenient for designers to transform a business model into a technical model and facilitate the alignment between high-level complex business requirements and IT systems. It is not the role of SoaML to define a methodology, but rather to provide a foundation for Model-Driven Service Engineering (MDSE) based on the MDA approach that can be adopted in different software development processes.

To design SOA-based software systems capable of satisfying multiple Quality of Service (QoS) attributes, executable modeling is desirable. For instance, simulation plays a central role in enabling tradeoff study among time-based quality of service attributes. For successful project managements, it is very important to validate Functional Requirements (FR) and evaluate Non-functional requirements (NFR) precisely in early design phase before

implementation of these systems. Most of current solutions, however, are based on interface testing, do not support early assessment of service performance, and mainly evaluate the overhead of service protocols and specifications. Also, such techniques do not allow customers, with no access to the service code, to simulate service performance at runtime (Ardagna, 2013). Other efforts are made to transform the SOA models into other executable models like discrete-event models (e.g. Symbolic Transition System (Ardagna, 2013), Petri Net (He, 2011) or its variants). Unfortunately, the transformed models including a lot of "places/states" and "transactions" which gives an appearance completely different from the original system architecture, hence this kind of simulation can't provide an intuitive observation of systems' behavior and interactions between services. On the other hand, Discrete Event System Specification (DEVS) which starts from general system theory may be the best option as the target formalism to make the SOA models executable, since it provides comprehensive and intuitive description of structure and behavior. In addition, it has been proven to be a universal formal mechanism to express a variety of discrete-event system subclasses, including Petri Net, Cellular Automata and Generalized Markov Chain (Vangheluwe, 2000).

This paper aims to provide an integrated model-driven design and validation approach to SOA by bringing together MDSE with SoaML and Modeling & Simulation (M&S) methodology based on DEVS. In the rest of this paper, section 2 discusses related work on similar problems in executable modeling approach to SOA. Section 3 introduces Parallel DEVS (P-DEVS) and DEVS Modeling Language (DEVSML). Section 4 proposes an Extended DEVS Modeling Language (E-DEVSML) based on extended elements in P-DEVS. Section 5 gives the outline of our approach and introduces the example used to demonstrate our approach. Section 6 presents our approach in detail and proves its applicability and practicability. Section 7 concludes the paper and proposes the direction for future research.

2. RELATED WORK

Many research works have been made in M&S related to service-oriented computing. To make SOA executable, basically, most of the proposed approaches can be divided into two categories: formalism-based methods and model-driven methods. Next, we present a comprehensive survey of them in detail.

In general, some directly use a modeling method based on formalism with executable semantic to analyze different non-functional properties of services. It shows that performance analysis can be integrated in the early development process. In fact, traditional formalism-based framework depends on certain simulation formalisms in a theoretical or mathematical way. Furthermore, many tools for service-oriented formalism-based simulation framework are implemented. A model based approach (Ardagna, 2013) that relies on Symbolic Transition Systems (STS) is proposed to describe web services as finite state automata and provide an early assessment of service performance. This approach uses simulation along the design and pre-deployment phases of the web service lifecycle to preliminarily assess web service performance. Another approach (Sarjoughian, 2008) is developed by unifying the DEVS and SOA frameworks. Based on the DEVS and SOA concepts and principles, a set of primitive and composite service model abstractions along with their interactions are defined. The resulting SOA-compliant DEVS (SOAD) framework supports simulations of service based systems. These approaches need modelers to be familiar with the formal basis and the corresponding modeling method. These researchers (Muqsith, 2010) also extend the SOAD framework by introducing dynamic structure DEVS to model and simulate the structure changes in service-based systems. The Cell-DEVS is another DEVS-based formalism that defines spatial models as cell spaces. Web enabling CD++ (Madhoun, 2006), which is an M&S toolkit to execute Cell DEVS models, can expose simulation functionalities as Web services to improve interoperability and reusability for the users' convenience. And D-CD++ (Wainer, 2008), an architecture of a web services based distributed simulation framework, is then put forward. Comparing with CD++, D-CD++ emphasizes the specific characteristic of distribution explicitly. It improves reusability and interoperability for users' convenience. Although the declarative formalism based methods (e.g. SOAD, D-CD++) has the advantages of rigorous theoretical basis, mathematical semantics, mature formalism and strong presentation capability to various systems. There still exist limitations. In one hand, formalism based methods are extremely too abstract and difficult to follow by users. In the other hand, the modeling for these methods is extraordinary complex. Hence, they have not been widely recognized by academic and industry.

Apart from formalism-base methods, there have been many approaches that aim to make static SOA models executable by MDA approach such as code generation. The Dynamic Distributed Service-Oriented Simulation Framework (DDSOS) (Tsai WT, 2006) focuses more on the domain of service-oriented software development. It is a distributed multi-agent service-oriented framework based on the Process Specification and Modeling Language for Services (PSML-S) (Tsai WT, 2007). A similar framework is also mentioned in (Jia L, 2009). The differences between their framework and DDSOS are the replacement of PSML with UML as the common model specification and the lack of some dynamic properties. Nevertheless, it lacks some high level formalism or theory basis for PSML and the DDSOS framework. The model driven methods perform excellently in modeling and generating code automatically. They have dynamic composability and support service-oriented systems engineering. Some limitations of the model

driven methods still exist and should be improved. For instance, these methods only focus on service oriented software development and have limited simulation capabilities. In addition, theory, efficiency, and applications still need to be improved.

Some researchers also propose methods which combines both formalism and MDA. Petri-net based approach is commonly used. In Narayanan's approach (Narayanan, 2003), the services are specified with using the DARPA Agent Markup Language for Services (DAML-S) and converted to Petri-net models and simulated using the "KarmaSim" environment. However, it provides a strong basis for verification and validation of the models, the combination of DAML-S and Petri-net lacks a sound basis for describing time-based dynamics of SOA, and this kind of simulation can't provide an intuitive observation of systems' behavior and interactions between services. By comparison, more powerful cross-platform framework DEVS/SOA (Mittal S, 2009) give ways to automatically generate DEVS models from various types of business process specifications and realize distributed simulation execution using web services, but the DEVSML used as key modeling language has difficulties to describe complex transitional behaviors due to limitations of Finite and Deterministic DEVS (FD-DEVS).

We also proposed a generic and comprehensive approach (Hu, 2014) to the design of SOA by combining a universal modeling language for SOA (SoaML) and a universal formal mechanism (DEVS) to service-oriented systems. This paper extends the former work presented in SCC 2014, which includes more details of the approach with some important improvements of DEVS modeling and expands on the worked example with more experimental results. We extended DEVSML to support nondeterministic state transition and enhance its capability to describe complex behavior of systems. Then we provided an automated transformation process using E-DEVSML as a model transformation intermediary to realize executable SOA in a DEVS-based simulation. We take advantage of both formalism-based and model-driven methodologies to specify a unified model-driven design and validation approach to SOA. This approach bridge generic service design and universal simulation paradigm with formal bases and practical implementation.

3. PARALLEL DEVS AND DEVS MODELING LANGUAGE

DEVS is a formal specification for general discrete event dynamic systems. Starting from the classic DEVS proposed by Zeigler, the simulation community have proposed different forms of DEVS for systems with different characteristics. As we shall introduce later, P-DEVS removes constraints in the classic DEVS that originated with the sequential operation of early computers and hindered the exploitation of parallelism (Zeigler, 2000).

3.1 Parallel DEVS

DEVS models can fall into two categories: atomic and coupled. The atomic model is the irreducible model definition that specifies the behavior for any modeled entity. The coupled model represents the composition of two or more atomic and coupled models connected by explicit couplings. An atomic model *M* and a coupled model *N* are defined by the following equations:

$$M = < IP, OP, X, S, Y, \delta_{int}, \delta_{ext}, \delta_{con}, \lambda, ta > \quad (1)$$

$$N = < IP, OP, X, Y, D, EIC, EOC, IC > \quad (2)$$

In an atomic model, *S* is the state space; *IP*, *OP* are the set of input and output ports; *X*, *Y* are the set of Inputs/Outputs, which are basically lists of port-value pairs, are the basic exchange medium.

$X = \{ (p,v) \mid p \in IP, v \in X_p \}$, $Y = \{ (p,v) \mid p \in OP, v \in Y_p \}$,where X_p and Y_p are input/output values on port *p*.

$\delta_{int} : S \rightarrow S$ is the internal transition function;

$\delta_{ext} : Q \times X^b \rightarrow S$ is the external transition function, where $Q = \{ (s,e) \; s \in S, 0 \leq e \leq ta(s) \}$ is the total state set, *e* is the time elapsed since last transition, and X^b is a set of bags composed of elements in *X*;

$\delta_{con} : S \times X^b \rightarrow S$ is the confluent transition function, which decides the order between δ_{int} and δ_{ext} in cases of collision between simultaneous external and internal events, subject to $\delta_{con}(s, \emptyset) = \delta_{int}(s)$, where $\emptyset$ means no input occurs.

$\lambda: S \rightarrow Y^b$ is the output function, where Y^b is a set of bags composed of elements in *Y*; $ta(s): S \rightarrow R_0^+ \cup \infty$is the time advance function. Two state variables are usually present in the state space of an atomic model: 'phase' and 'sigma'. Sigma keeps the time advance value. In the absence of external events the system stays in the current 'phase' for the time given by 'sigma'.

In a coupled model, *IP*, *OP*, *X and Y* have similar connotation as in atomic model, but mean external (not coupled) elements; *D* is a set of DEVS component models. *EIC* is the external input coupling relation; *EOC* is the external output coupling relation; *IC* is the internal coupling relation. The coupled model itself can be a part of a component in a larger coupled model system giving rise to a hierarchical DEVS model construction.

3.2 DEVS Modeling Language

A DEVS specification language or modeling language (e.g. XFD-DEVS (Mittal, S, 2013), DEVSpecL (Hong. KJ, 2006), DEVSML2.0 (Mittal, S, 2012)) is usually used as a model transformation intermediary to make static models executable. For example, UML models are first converted to DEVSML models, and then translated into executable codes. XFD-DEVS based on XML and FD-DEVS (Hwang Moon Ho, 2009) has many shortcomings, such as, no confluent function, no multiple inputs, no multiple outputs, no complex message types and no state variables. These

shortcomings are removed in the upgraded version DEVSML 2.0. Both based on Extended Backus-Naur Form (EBNF) notation, but by comparison with DEVSpecL, the DEVSML2.0 is more close to the true DEVS formalism with some necessary abstractions. In his recently work (Mittal, S, 2013), Mittal presented DEVSML 2.0 stack employing Model to Model, Model to DEVSML and Model to DEVS transformations, which aim to unify the Domain Specific Language (DSL) community with the DEVS community, and make those components fall under an DEVS unified process (DUNIP). This framework is attractive, however, how to realize M2DEVSML between different DSL and DEVSML 2.0 is still not given, and it has some limitations to express complex transition logic as it employs deterministic properties of the constituent elements, which are formally defined in FD-DEVS.

DEVSML contains three primary element types (i.e. the Atomic, the Coupled and the Entity). When DEVS is tied to a platform specific implementation, the message objects are exchanged according to the port-value pairs specified in the atomic model structure. In this manner, the entities are defined as a data class to depict message types. These entities are then declared in atomic or coupled components for their reuse. Both Atomic and Coupled model grammar stay as close as possible to the P-DEVS formalism. However, some abstractions still exist. For instance, any state transition based on the message content is not realizable because of FD-DEVS. The major specification of DEVSML is shown in *Figure 1*, and detail introduction can be found in (Mittal, S, 2012).

```
Entity:
 'entity' name=ID
 ('extends' superType = [Entity|QualifiedName])?
 ('{' (pairs += Variable)* '}')? ;

Atomic: 'atomic' name=ID
('extends' superType=[Atomic|QualifiedName])?'{'
 'vars''{'(variables += Variable)*'}'
 'interfaceIO' '{'(msgs += Msg)*'}'
 'state-time-advance' '{'(stas += STA)*'}'
 'state-machine''{'
   'start in ' init=InitState
   (atBeh += AtomicBeh)* '}'
'}';

AtomicBeh:
 stm1=Deltext | stm2=Outfn | stm3=Deltint |
 stm4=Confluent;
Coupled: 'coupled' name=ID
 ('extends' superType=[Coupled|QualifiedName])?'{
 'models' '{'(components += Component)*'}'
 'interfaceIO' '{'(msgs += Msg)*'}'
 'couplings' '{' (couplings += Coupling)*'}'
 '}';
Msg: type = ('input'|'output')
    ref=[Entity|QualifiedName] name=ID ;
```

Figure 1.DEVSML specified in EBNF grammar

4. EXTENDED DEVS MODELING LANGUAGE

In practice, when we model system architecture with previous DEVSML version, FD-DEVS usually can't resolve complex problem. For instance, when an input arrives on one port of an atomic model, it may invoke an external state transition or may not, that is depending on the practical requirement and specific scenario, although it is forbidden according to the deterministic property. Another example is an atomic model with two input ports where a transition to state *SA* will be triggered if a message arrives at port *A* but a transition to state *SB* should be invoked if the message arrives at port *B*. Certainly, there will be a collision if two messages simultaneously arrive on both ports. Although DEVSML provide user-defined code block that elaborating detailed behavior of models, it is still inconvenient to deal with troublesome nondeterministic state transitions. Therefore, we take appropriate measures to extend DEVSML to deal with these complex situations. This section presents the proposed extended part of DEVS modeling language and its formal bases.

4.1 Extended Elements in P-DEVS

There are three essential modeling elements: messages, states and transitions in DEVS. In practice, the number of messages and states is so huge and sometimes infinite that we cannot treat it easily by directly applying the DEVS formalism in the course of modeling. It's important to note that in the equations of P-DEVS, we could not directly understand what is the state space S and what is the set of inputs X^b. Thus we propose an extension called SP-DEVS on the base of these equations to exploit the notion of message as a structured form of related I/O, and of state variables to aggregate relevant sequential states. Note that basic definitions of atomic model and coupled model also follow *Equation 1* and *2*.

4.1.1 Formalized Message with Port, Value and I/O variable

As stated, X and Y are the set of I/O, which are basically lists of port-value pairs, and a message is a bag of elements in X or Y. If a message is arrived, several ports may catch some values at the same time. Therefore we have:

$X = \{ (p,v) \mid p \in IP, v \in dom(p) \}$, $Y = \{ (p,v) \mid p \in OP, v \in dom(p) \}$, where $dom(p)$ is type or domain of port p, if a coupling from one output port py to another input port px such that $dom(py) \subseteq dom(px)$.

The message set $M = X^b \cup Y^b$, and $X^b = \{ m_x \mid m_x \subseteq X \}$, $Y^b = \{ m_y \mid m_y \subseteq Y \}$. Sometimes, simultaneously arriving inputs on different ports make it complex to describe the behavior of the external transition function. If we choose an alternative transition to trigger, some inputs may be ignored or lost. Therefore we try to separate the processing of inputs from state transition, if we first store these inputs in some I/O variables, such that, any inputs will not be ignored or lost. To realize this idea, we define a set of Input-port-associate Variables (IV) V_x, and a set of Output-port-associate Variables (OV) V_y, these variables may be simple data type (e.g. integer, floating number, string) or a container of simple data types (e.g. queue, stack, list). We can also define a binding between an I/O variable and a port:

$Rev = \{ (p, v_x) \mid p \in IP, v_x \in V_x , dom(p) = dom(v_x) \}$, $Sed = \{ (p, v_y) \mid p \in OP, v_y \in V_y , dom(p) = dom(v_y) \}$ where Rev is used to receive messages and Sed is used to send messages.

4.1.2 Formalized State with Phase and State variable

A discrete-event system's status can be specified by a set of system variables or attributes. Each state variable represents an attribute characterizing the system. Thus a system state is a combination of values that the state variables have at a time. Now consider that we partition the composite state set into equivalent groups such that each group has a set of sequential states. Let each group have a single representative name, called a phase. Formally, a phase ψ is a representative value of a set of equivalent states which produce the same output event and/or have the same time advance at the states. Therefore the state space S can be defined as follow:

$$S = < \Psi, SV, \{dom(sv) \mid sv \in SV\}, \alpha_s > \quad (3)$$

Ψ is a set of phases, SV is a set of state variables, $dom(sv)$ is the range set of a state variable $sv \in SV$, and α_s is the one-to-one assignment function which is subject to the constraint:

$$\alpha_s^{-1} : \times_{sv \in SV} dom(sv) \rightarrow \Psi \quad (4)$$

which is a bijection such that $S = \{ s_i = \{\psi_i\} \cup \{sv_{1i}, sv_{2i}, \ldots\} \mid \psi_i \in \Psi, sv_{1i} \in dom(sv_1), sv_{2i} \in dom(sv_2), \ldots, sv_i \in SV, \exists s_i = s' = \{\psi'\} \cup \{sv_1', sv_2', \ldots\} \}$. An element $sv_i = \{sv_{1i}, sv_{2i}, \ldots\}$ mapped to a sequential state s_i is called a composite state in the state space. s' is the initial state of the state space and ψ' is the initial phase. By the way, a phase can be hierarchical, that is, a phase can be decomposed into sub-phases having disjoint composite state members. It is always true that a union of sub-phases within a phase gives the total states set of the phase and the intersection of all sub-phases results in an empty set. So this feature makes it easy to map P-DEVS onto StateCharts or UML StateMachine which also have composite state or sub-StateMachine.

Figure 2 gives an illustration of the notion of the structured states and phase transitions. There are three states with phases ψ_1, ψ_2, ψ_3, and ψ_1 has three sub-phases $\psi_{11}, \psi_{12}, \psi_{13}$; The values of two state variables sv_1 and sv_2 are grouped into these states with disjoint composite state members. When the system stay at ψ_{11} and a message m_1 arrives, an external transition is triggered, if the guard condition g_1 is satisfied, the system will transit into phase ψ_2 and the action a_1 is executed to change the value of sv_1 from sv_{11} or sv_{12} to sv_{14}. While the system stay at ψ_{12} and $e=ta(s_1)$,an internal transition is triggered, if the guard condition g_2 is satisfied, the system will transit into phase ψ_3 and the action a_2 is executed to change the value of sv_1 from sv_{13} to sv_{15} and a message m_2 will be sent out as well.

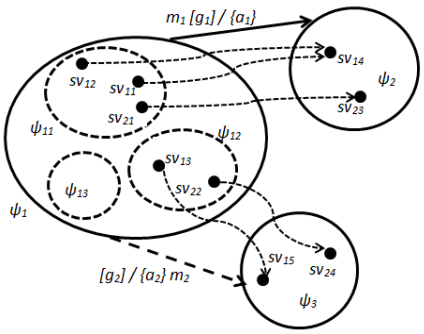


Figure 2. Illustration of the structured states and phase transitions

4.1.3 Formalized Transition with Event, Guard and Action

A state transition usually occurs by an **event/guard/action** pair, and is actually a phase transition here. The event refers to an external message arrival or an internal event ($e=ta(s)$) happening, but the guard and the action are still not formally defined in P-DEVS, and they are just implemented within transition functions in executable codes of DEVS. Some DEVS modeling languages are created based on a sub set of classic DEVS called FD-DEVS which only support deterministic state transitions. The guard condition is unnecessary in FD-DEVS, however, we wish P-DEVS to be capable of coping with complex situations including nondeterministic state transitions. So the guard condition should be added to transition functions. It may be a guard of messages or guard of variables.

The action is usually used to change value of variables including state variables and I/O variables. After that the output function λ can only be implemented by pushing out the data stored in OVs after an internal state transition. In this way, we revise the definition of the transition functions like these:

$$\delta_{ext} : Q \times X^b \times G \rightarrow S \times A \quad (5)$$

Where Q, S and X^b are the same as in P-DEVS; G is the guard condition set consisting of logical expressions on messages or variables. It is denoted by:

$$G: X^b \times_{v \in SV \cup Vx} dom(v) \rightarrow Boolean \quad (6)$$

And the action A is a data process function, which may construct a vector of individual actions for each variable:

$$A: \times_{v \in SV \cup Vx \cup Vy} dom(v) \rightarrow dom(v) \quad (7)$$

The internal transition function is also extended with guards and actions:

$$\delta_{int} : S \times G \rightarrow S \times A \quad (8)$$

The output function $\lambda: S \rightarrow Y^b$ and the time advance function $ta(s): S \rightarrow R_0^+ \cup \infty$ are the same as original P-DEVS. Finally the confluent transition function is:

$$\delta_{con} : Q \times X^b \times G \rightarrow O_t \ , \ Q = \{ (s,e), e=ta(s) \} \quad (9)$$

Which decides the order between δ_{int} and δ_{ext} in cases of collision between simultaneous external and internal events. O_t is a choice among the options including ignore-input, input-only, input-first and input-later, and then trigger the corresponding transition functions in this order.

4.2 Abstract Syntax of E-DEVSML

A textual language is usually specified using Extended Backus-Naur Form (EBNF) notation, and Xtext is a powerful tool to do this for the development of a domain specific language. After using Xtext's EBNF grammar language to define the abstract syntax of E-DEVSML, it will create the meta-model and the parser automatically by starting a generator. Considering of both convenience of use and conformity with the P-DEVS formalism, we specify

SPDML with modular and object-oriented features. Models in E-DEVSML are divided into three primary elements: the Atomic, the Coupled and the Entity.

1) Entity: DEVS is a component-based framework where each of the components communicates using messages. These message objects are exchanged according to the port-value pairs specified in the atomic model, and the datatype of a input value can be defined as an entity and reused by some ports. According to the object-oriented principles, the entities are defined not as a part of the component but as a first-class citizen. *Figure 3* gives the definition of *Entity* in EBNF.

```
Entity : 'entity'  name=ID
 ('extends' superType = [Entity |QualifiedName])?
 ('{' (attributes += Variable)* '}' )? ;
Variable :
type = VarType name=ID ('default' deft=STRING)?;
VarType:
 simple = ('int'|'double'|'String'|'boolean')|
 container = ('queue'| 'stack' | 'list' )|
 complex = [Entity | QualifiedName];
```

Figure 3. Definition of Entity in EBNF

An Entity is specified by a name, *name=ID*. It may extend another entity. The expression *[Entity| QualifiedName]* means that the superType is to be specified as a *QualifiedName*, which is an Xtext construct and is of type Entity. For more details on *QualifiedName*, please refer Xtext manual. We define a variable type named *container* which is a common data structure (e.g. queue) to store a series of entities. The keyword *default* assigns values to variables when model is started or restarted.

2) Atomic: The Atomic is the most important and complicated part of DEVS. Every core concepts in SP-DEVS should be defined as corresponding elements in EBNF. The Atomic model is specified in EBNF grammar as *Figure 4* showing.

```
Atomic : 'atomic'  name=ID
('extends' superType = [Atomic |QualifiedName])?
'{'
 'vars''{'(variables += Variable)*'}'
 'interfaceIO'  '{' (ports += Port)*'}'
 'state-time-advance'   '{' (phase += Phase)*'}'
 'state-machine''{' 'start' initState = InitState
     (rev += ReceiveMessage)*
     (statefunc += AtomicBehavior)* '}'
  (functions += UserFunction)* '}';
Port : type = ('input'| 'output')
       reference=[Entity |QualifiedName] name=ID;
Phase : name=ID timeAdv=TimeAdv;
TimeAdv : ta=DOUBLE|inf='infinity'|taVar=[Variable];
InitState : state=[Phase] (code=Code)?;
UserFunction:
'function' name=ID 'return' (returnto =  [VarType])?
   '('(paras+=[Variable])*')'  usercode=Code ;
Code: '{' str=STRING  '}' ;
```

Figure 4. Definition of Atomic in EBNF

The keyword *vars* defines a set of variables including I/O variables and state variables. The *interfaceIO* specification gives the definition of ports with specific data type which is referenced as an *Entity* type. And *state-time-advance* defines set of states and the associated time-advances. Each state-time-advance pair is defined as a *Phase*. The time-advance *TimeAdv* can have values of either *DOUBLE*, *infinity* or a *Variable* declared above in the atomic model. The *state-machine* contains the initial state *InitState* and the atomic behavior *AtomicBehavior*. The expression *state=[Phase]* implies that the model references the state already defined in the construct *Phase* defined earlier in the model. The next expression *(code=Code)?* implies that there may be code snippet associated with setting up of the initial state. As we shall see in a later section on applicability of SPDML, the code expressed as a *STRING* is syntactically checked at run-time for any compilation errors.

E-DEVSML has four functions to specify the atomic behavior. Correspondingly, the *AtomicBehevior* is divided into four parts (shown in *Figure 5*): *Deltext, Deltint, Outfn* and *Confluent*. To enhance flexibility and convenience of modeling, some special features are provided:

a) Separation of message processing from state transition: from functional perspective, an abstract description of an atomic model is: accepting inputs or incentives, and generating an output with state changes. We could divide this input/output process into three parts: receiving a message, data processing during a series of transitions and sending a message. As any input ports may accept some inputs when a message arrives during any state, the *ReceiveMessage* is used to allocate the message (port-value pairs) to several IVs which is a static banding declared in an atomic model. In this way these variables play roles as buffers controlled by user. On the contrary, the *SendMessage* is a dynamic banding declared in the output function to pack values of OVs and output ports into a message. After outputs are pushed out these variables are cleared out automatically.

b) Guard transition defined to support for nondeterministic state transitions: a guard may be variable correlate or message correlate. We argue that a specific input can also be viewed as a part of a guard condition owing to it is stored in an IV. For example, a variable is used for storage of inputs on a port, after receiving a message, if this variable is not null (guard condition), transition is triggered. In both *Deltext* and *Deltint*, for more flexible description of a guard condition, literal strings are used as standard format which will be directly translated into executable codes.

c) Additional extended features: the *UserFunction* permit user to add code embedded in the generated atomic and coupled model class source file, user-defined method defined in an atomic model can be called by the atomic itself and the coupled model comprising it to implement interoperability. The *Hold* allows the model to still stay in the source state but advance the elapsed time or redefine the time-advance. The *SetSigma* rule also allows the resetting of time-advance of the target state.

3) Coupled: The specification of a coupled model is shown in *Figure 6*. To put it simply, the coupled model can extend from another coupled model and it has the same interface specification as the atomic model. It is composed of a set of models which can be either atomic component or coupled component. The definition of Coupling includes *EIC* (defined for connections originating from input interface of the coupled model to its subcomponents), *IC* (specified between the sub components) and *EOC* (specified from the contained component to the outside interface of the coupled model). The keyword *this* means the component itself.

```
AtomicBehavior:
 f1=Deltext | f2=Deltint | f3=Outfn |f4=Confluent;
Deltext:'deltext' '{'
'S:' state=[Phase] ( 'guard' '(' guard= Guard ')' )?
(dataprocess = Action)?  (hold = Hold |
transition = Transition )'}';
Deltint:'deltint' '{'
'S:' state=[Phase] ( 'guard' '(' guard= Guard ')' )?
 (dataprocess = Action)?  transition = Transition '}';
Outfn: 'Outfn'
   '{''S:' state=[Phase] (sed += SendMessage )*'}';
Confluent: 'confluent'  con=( 'ignore-input'| 'input-only'|
   'input-first' | 'input-later') ;
ReceiveMessage:'receive' '['into=[Port]
   ('.' subMsg = QualifiedName)?']'receiver=[Variable];
SendMessage: 'send' '['outfrom=[Port] ']'sender=[Variable];
Guard:guard=Code;  Action: code=Code ;
Hold: con='continue'| sig=SetSigma;
SetSigma:
'sigma' '('value = DOUBLE|inf='infinity'| sigVar=[Variable]')';
Transition: 'goto' target=[Phase] (sig=SetSigma)? ;
```

Figure 5. Definition of AtomicBehavior in EBNF

```
Coupled:
 'coupled' name=ID
 ('extends' superType=[Coupled|QualifiedName])?
 '{'   'vars''{'(variables += Variable)*'}'
  'models'   '{' (components += Component)* '}'
 'interfaceIO'  '{' (ports += Port)*'}'
 'couplings'  '{'  (couplings += Coupling)*'}'
  ('user-code'  userCode=Code)? '}';
 Component:  AtomicComp | CoupledComp;
AtomicComp:
'atomic'  at=[Atomic | QualifiedName] name=ID ;
CoupledComp :
'coupled' cp=[Coupled | QualifiedName] name=ID;
Coupling : ic=IC | eoc=EOC | eic=EIC;
EIC : 'eic'    'this'':' srcport = [Port ] '->'
   dest = [Component] ':' destport = [Port];
IC: 'ic'   src= [Component | QualifiedName]  ':'
 srcport =  [Port] '->'
 dest= [Component | QualifiedName]  ':'
 destport = [Port];
EOC : 'eoc'
src = [ Component] ':'  srcport = [Port ] '->'
'this'':' destport = [Port];
```

Figure 6. Definition of Copuled in EBNF

Compared to the original version, in this paper we make a lot of changes on the atomic model definition to enhance its capability of complex behavior description but fewer changes on coupled model. Without operational characters and complex control statements, unlike a programming language, E-DEVSML still needs some embedded code to describe complex logics. Fortunately, these codes are only involved in data processing from IVs to OVs, which is independent of specific DEVS simulator.

5. Overview of the Approach

5.1 Outline of the Approach

The MDSE methodology based on MDA guides solution architects in how to specify services that are aligned with the business process models (Elvesæter, B., 2011). In our model-driven approach shown in *Figure 7*, the Business Architecture is first built as a Computation Independent Model (CIM), the computational and implementation details of the system are hidden at this level of description. The CIM is transformed into the System Architecture Model, a Platform Independent Model (PIM) which contains the necessary computational information for the application, but no information specific to the underlying platform technology which will be used to eventually implement the PIM. After finishing the design of the SOA, the SoaML models can be transformed into platform independent DEVS models in E-DEVSML and finally are transformed into a Platform Specific Models (PSM), which are actually described using executable codes. At the same time, we also need to design an Experimental Frame (EF) to start the simulation. The construction of the EF is as important as SOA models for this simulation environment because it will introduce QoS goals associated with system quality attributes and will calculate quality indicators for each attribute to be analyzed. In this paper we'll construct generators and transducers to serve as components in the experimental frame module for measuring performance of this service-oriented system.

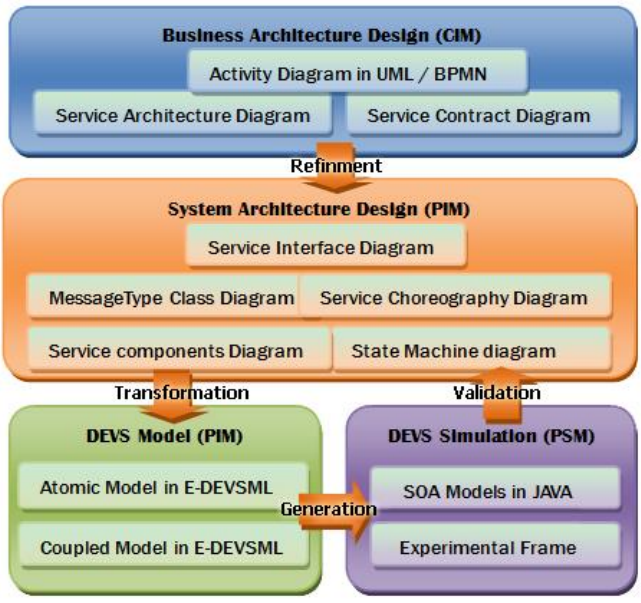


Figure 7. Framework of the model-driven approach to SOA

Generally, there are two different ways to implement a DEVS simulation from other PIMs. One is directly mapping formalism of different diagrams to DEVS formalism and generating executable codes for DEVS simulators on different platforms. Another way is to apply a DEVS specification language or modeling language as a model

transformation intermediary. According to the first method simulation can be implemented on a single platform, and the conversion process is relatively complex; The second method is more reasonable because it has two major advantages: (1) the DEVS modeling languages are generally platform-independent and can be transformed into executable codes on different platforms, that exactly satisfies the requirements of SOA; (2) Before creating a DEVS simulation we need to remove redundancy and take the intersection of the information provided by models of different perspectives. Therefore, we certainly take advantage of the two-steps method to make SOA models executable. At last, we use one of the open source DEVS simulator named DEVS-Suite (The Arizona Center for Integrative Modeling and Simulation, 2014) to validate the SOA models.

5.2 Illustrative Scenario

An airport is a typical SoS and it is composed of systems from different suppliers that use different design methods and implementation technologies, thus SOA is usually applied to deal with that kind of heterogeneity. We motivate our approach using the airport scenario and

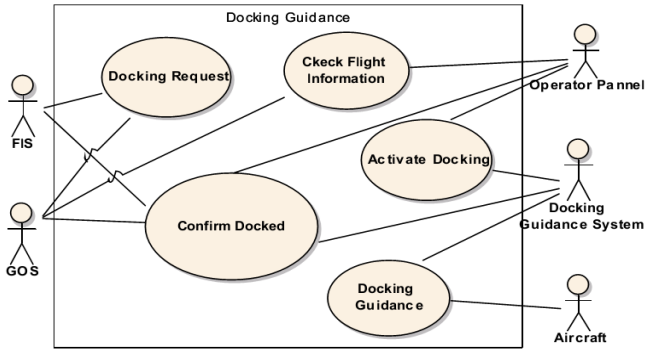


Figure 8. Illustrative scenario described by a use case

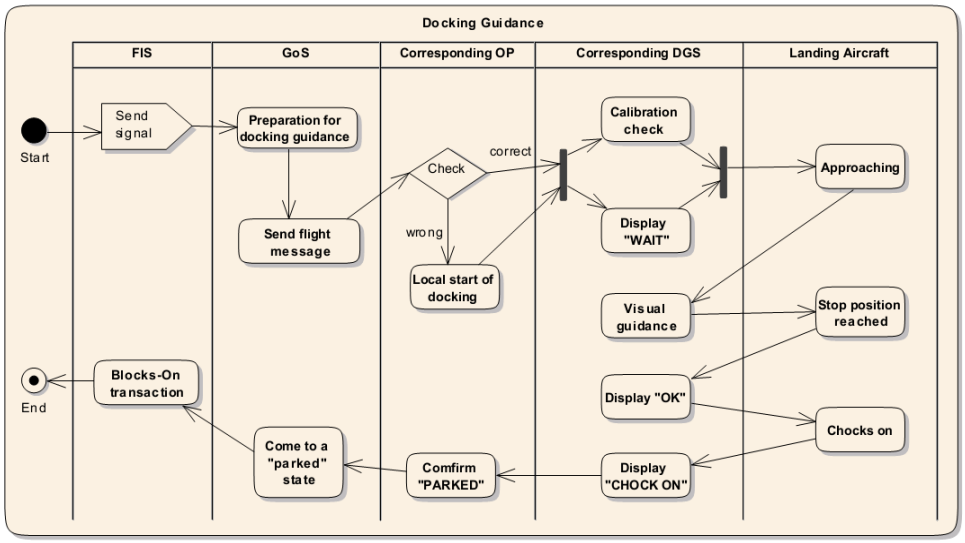


Figure 9. Illustrative scenario described by an use case

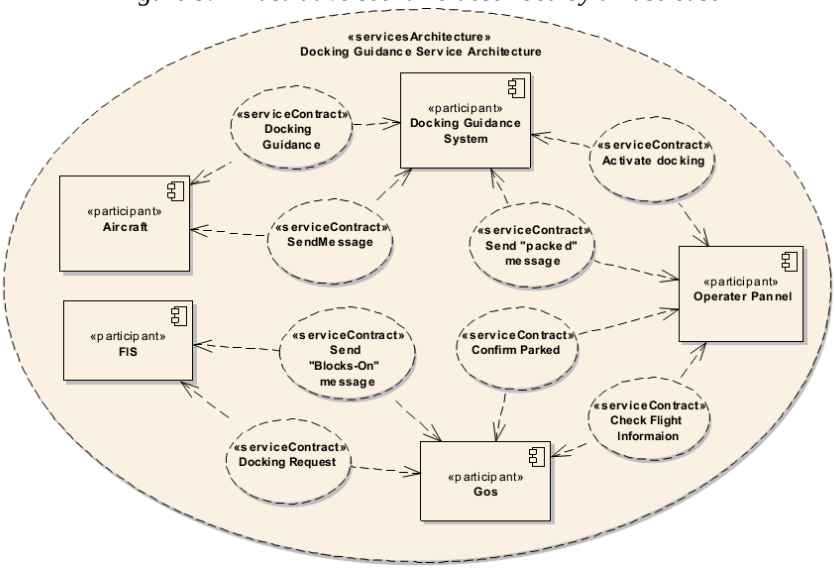


Figure 10. Services architecture of docking guidance in SoaML

business process of an aircraft docking is illustrated. Figure 8 show this scenario by an Use Case diagram, the collaboration of the involved systems includes FIS (Flight Information System), GOS (Gate Operation System), DGS (Docking Guidance System) and Operator Pannel (OP). They are all necessary for the aircraft to be smoothly manoeuvred to the correct centerline and stop-position.

First, the FIS sends docking request to the GOS before aircrafts' landing, and the GOS assigns the suitable gates (each gate has a set of DGS including an OP) for each aircraft according to the schedule. After that, the flight information is sent to the corresponding DGS. And the operator checks it for correct aircraft type and flight number, then activate a docking process. Note that a GOS usually takes charge of many gates with DGS, for concise illustration of this example we only show few sets of DGS in the business process.

6. Model-driven Design and Validation of SOA

6.1 Business Architecture design

Our model-driven approach starts from design of Business Architecture Model (BAM). It illustrates the business processes with the associated elements of information by using a UML Activity Diagram (AD) shown in *Figure 9*. It can provide a great assistance for capturing business activities and identifying services in ADs. The BAM further describes the services architecture of the business community and the service contracts between the business entities participating in the community as illustrated in the SoaML diagrams. The Business Process Modeling Notation (BPMN) is often used to describe business processes while mapping rules between BPMN and SoaML frequently appear in articles on MDSE (Elvesæter, B., 2011). Note that we decided to use the activity diagram at this level in contrast with BPMN. The main reason is that AD is a standard notation, well-known from software developers, while BPMN is a notation created for business people, although both notations and views on the meta-model are very similar. In this activity model, actions are displayed while business collaborators are displayed as a partition in the AD. Then the SoaML diagrams can be derived from this AD and some refinements may be required.

A service architecture is a high level description of how participants work together for a common purpose by providing and using services expressed as service contracts. Participants are identified in an AD partition, and when two single actions follow one another across several partitions and are connected with a control flow, this collaboration can be identified as a service contract. Once the service contracts are identified, the consumer and provider roles should be specified. The services architecture and service contracts of the docking guidance process are shown in *Figure* 10 and *Figure* 11.

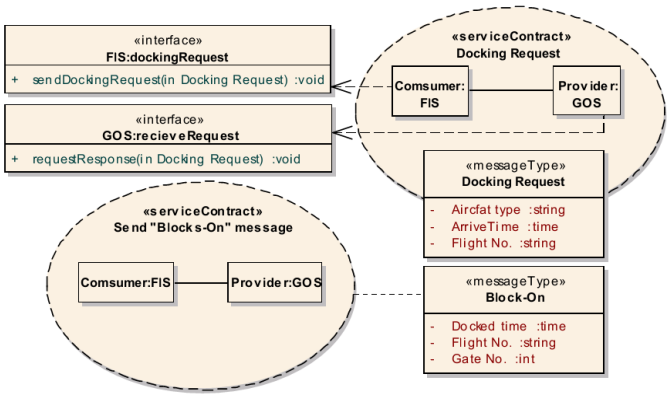


Figure 11. Service Contract, Services Interface and Message Typese

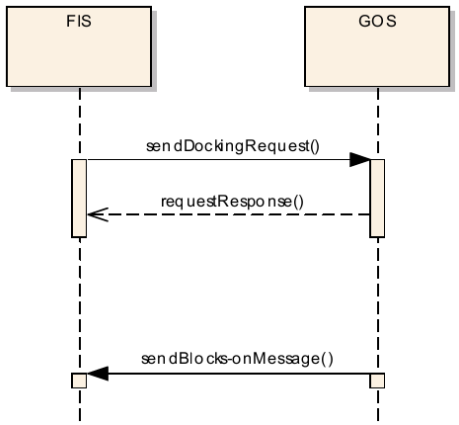


Figure 12. Service Choreography between FIS and GOS

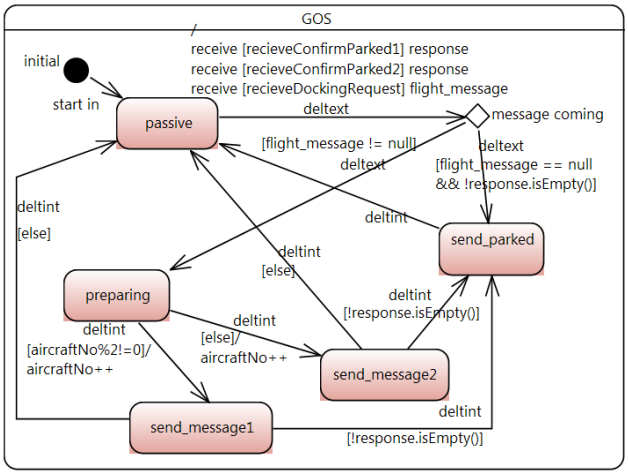


Figure 13. State Machine Diagram of GOS

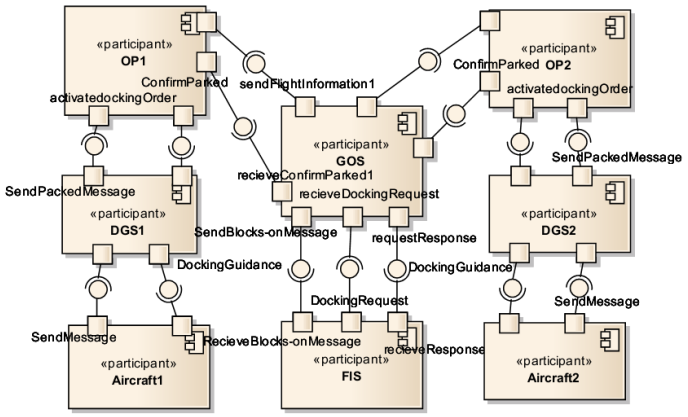


Figure 14. State Machine Diagram of GOS

6.2 System Architecture design

The System Architecture Model (SAM) describes the IT perspective of a service-oriented architecture. The SAM is a refinement of the BAM, and is used to express the overall architecture of the system at the PIM level. It partitions the system into components which are specified in a service components diagram. It also defines the components in terms of "what interfaces are used" and "how the interfaces should be used". Furthermore, *Figure 11* shows the Service Interfaces and the data objects in the control flows which are represented by SoaML message types. Provided and required interfaces are denoted by provider and consumer. The provided interface contains the operations of the service, while the required interface can have callbacks, which are specified as signals. For instance *Figure 11* shows the consumer interface dockingRequest with the operation sendDockingRequest, and the provider interface with the callback requestResponse. Service choreographies in *Figure 12* are usually described by a Sequence Diagram (SD). It also specifies how operations and callbacks are put together into a conversation between the two participants. The message specification types are closely related to the specification of the operations and callbacks in the interfaces. The message type Docking Request in *Figure 8* has some properties which contain additional information to support the FIS communications with the GOS. On *Figure 13*, the State Machine Diagram (SMD) of the GOS can be acquired by refining the AD in *Figure 9*. Finally, the component model focuses on specifying the involved software components that realize the services architecture shown in *Figure 14*.

6.3 Mapping DEVS onto SoaML

As mentioned above, there are two ways to realize transformation from SoaML to DEVS and both of them require some mapping rules between source models and DEVS formalism. First, all the data types (e.g. Class, MessageType) inherit from UML::Class can be translated into data types of Inputs (X) / Outputs (Y) in DEVS as entities in E-DEVSML. Second, from structural perspective, Class Diagram give definition of attributes in an atomic model, while Composited Participants provide structural information about a coupled model including components in coupled model and couplings between them. In addition, Service choreography is able to represent both coupled model structure and behavior of its model, and Messages between lifelines are used to define ports in each components and couplings between them. However it's a little bit complex to transform it into atomic models without explicit states and transitions definition. Roberto (Pasqua Roberto, 2012) has made it possible by generating states and transitions between two consecutive specification occurrences. At last, from behavioral perspective, a SMD is more suitable to provide behavioral information of an atomic model since it is more convenient and provide more complete information. The detailed mapping rules between these diagrams and DEVS models are listed in table 1. And there is no concept can be mapped onto δcon. The default definition of confluent transition function simply applies δint before applying δext to the resulting state.

TABLE I. Mapping rules between DEVS and SoaML.

P-DEVS	IP	OP	D	EIC	EOC	IC
E-DEVSML	interfaceIO input	interfaceIO outout	models	couplings eic	couplings eoc	couplings ic
Composited Participants	Ports with provided interface	Ports with required interface	Components	Input delegation connectors	Output delegation connectors	Interfaces connected
P-DEVS	**S**	**ta**	**δ_{int}**	**δ_{ext}**	**λ**	**X/Y**
E-DEVSML	state-time-advance		deltint	deltext	outfn	receive/send
StateMachine Diagram	State	Constraint	Transition effect	Transition effect	State exit	Event/Action
Service choreography	Execution or Message Occurrence Specification	Constraint	Consecutive Occurrence Specification	Consecutive Occurrence Specification	Message Occurrence Specification	Message interchange

6.4 Transform SoaML into E-DEVSML and Java Codes

Xtext is a powerful framework which we applied in the development of E-DEVSML. As long as we define a grammar and execute the MWE2 workflow in Xtext, it automatically generates a compiler, an editor and a rich validation framework. In addition, the Xtext framework is seamlessly integrated with the Eclipse Java framework by code generator with Xtend. The automated transformation process from SoaML to DEVS is a two-step approach which employs and tools which contains model transformation technologies such as Xpand and Xtend. As SoaML models are stored in XMI files, due to the textual style of E-DEVSML, we use Model to Text (M2T) technology such as Xpand, a language specialized on code generation based on Eclipse Modeling Framework (EMF) models, to transform SoaML into E-DEVSML. Then the java codes are generated automatically by Xtend.

Generating DEVS models through the diagrams of SoaML as described table 1 may follow a certain order. First, we transform data perspective diagrams into DEVS entities; second, we select the structural perspective diagram of components which generates the corresponding DEVS coupled structure and class definition diagrams that generate variables in atomic model; at last, the behavior of each atomic model is defined by behavioral perspective diagrams which are augmented with more information as per DEVS requirements. The critical process of using Xpand is the translation from an XMI file containing various diagrams into E-DEVSML. Besides mapping rules given in table 1, the relation between source diagrams and extended parts of DEVSML is also needed. The guard conditions of transitions in SMD certainly are mapped onto the phase guard. The message receiving function, transition actions are derived from transition effects, and the message sending function is derived from state exit effects, *Figure 15* gives a template that teach the code generator how to translate source diagrams into E-DEVSML. Note that, when we create SOA models, we can embed multiform codes (e.g. C++, java, OCL and Natural language) into diagrams, and E-DEVSML and other platform codes may be used together to realize multi-platform executable models generation. *Figure 16* gives a comparison between E-DEVSML source file (.fns) and java source file of GOS.

```
deltext{
    receive [recieveConfirmParked1] response
    receive [recieveConfirmParked2] response
    receive [recieveDockingRequest] flight_message
    S: passive{
        when ("flight_message != null")
        {   goto preparing  }
        when("flight_message == null && !response.isEmpty()")
        {   goto send_parked }
    }
}                                                          GOS.fns
public void deltext(double e, message x)                   GOS.java
{   Continue(e);
    for (int i=0; i< x.size();i++)
    {  if (messageOnPort(x,"recieveConfirmParked1",i))
        {   entity val = x.getValOnPort("recieveConfirmParked1",i);
            DockingDone job = (DockingDone)val;
            response.add(job);  }
        if (messageOnPort(x,"recieveConfirmParked2",i))
        {  entity val = x.getValOnPort("recieveConfirmParked2",i);
            DockingDone job = (DockingDone)val;
            response.add(job);  }
        if (messageOnPort(x,"recieveDockingRequest",i))
        {   entity val = x.getValOnPort("recieveDockingRequest",i);
            DockingRequest job = (DockingRequest)val;
             flight_message=job; }
    }
    if(phaseIs("passive"))
    {   if (flight_message != null)
        {   holdIn("preparing",30.0);return;}
        if (flight_message == null && !response.isEmpty())
        {  holdIn("send_parked",0.0);return;}
    }
}
```

Figure 16. Comparison between E-DEVSML and java source file of GOS

6.5 Experimental Frame Design and Setting

EF is used to simulate the dynamic view of SOA to obtain output data produced by the system under specified conditions. It actually represents the external environment that interacts with the evaluated architecture. There are three major elements in the EF (Zeigler, B. P., 2000): generator which provides input segments to the system, transducer which observes and analyzes the system output segments, and acceptor that monitors experiments to verify the desired conditions. In this paper we only set some generators as providers, and some transducers as measurement calculators, where these two types of elements were enough to be adapted to the context of the aircraft docking. Design of EF depends on FR and NFR of SOA, because we need to validate the behavior of the system and calculate the corresponding QoS when the modeled SOA interacts with the EF. In this paper, we choose performance as illustrative metric related to quality attributes that can be measured at runtime in the simulation environment. For an airport, the throughput of flights is a critical concern during the design

```
...
«DEFINE Behavior FOR Model»  «EXPAND main FOREACH allOwnedElements().typeSelect(StateMachine)» «ENDDEFINE»
«DEFINE main FOR StateMachine» «FILE name.toString() + ".fns"-» «EXPAND reg FOREACH region» «ENDFILE» «ENDDEFINE»
«DEFINE reg FOR Region-» «REM»«EXPAND pseudo FOREACH subvertex.typeSelect(uml::Pseudostate)» «ENDREM»
state-time-advance{ «EXPAND subv_advance FOREACH subvertex.typeSelect(State)-» }
state-machine{
    start in «EXPAND start_in FOREACH transition.select(e|e.name == 'start in')»
    deltint{ «EXPAND state_in FOREACH subvertex.typeSelect(State)-» }
    deltext{ «EXPAND state_ext FOREACH subvertex.typeSelect(State)-» }
    outfun{ «EXPAND subv_out FOREACH subvertex.typeSelect(State)-»  }
«ENDDEFINE»
...
«DEFINE trans_in FOR Transition-»       S: «source.name»{
            «IF guard.specification != null-»   when("«guard.specification.stringValue()»")
            { «EXPAND effect FOR effect-»   goto «target.name»  }   «ENDIF»
            «EXPAND target FOREACH target.outgoing.typeSelect(Transition)»      }
«ENDDEFINE»
...
```

Figure 15. Comparison between E-DEVSML and java source file of GOS

process of airport systems, therefore, we calculate Average Turnaround Time (ATT) and throughput for the aircraft docking.

In an aircraft docking process, FIS actually plays a role of generator which is the source of stimulus to this SOA, because it sends docking request to the GOS before aircrafts' landing. We set periodic requests generated by FIS (e.g. one request every five minutes). The transducers are observers that keep a count of flights as well as docked ones. To compare performance of different DGS, we set several transducers as performance calculators: transducer1 for the DGS1, transducer2 for the DGS2, and transducer as total performance indicator for the entire GOS. As shown in Figure 14, after GOS receives docking requests, it will allocate these tasks to two sets of DGS averagely. The time advance values of different phases of DGS and OP are set asrandomly distributed numbers between a minimum value and a max value according to experts' experience and prediction. Aircaft1 and Aircrat2 simulate different flights to be guided by the corresponding DGS during the docking process. We assume that DGS2 is installed in a location which is further from the runway than where DGS1 is installed, thus it may take more time for an aircraft guided by DGS2 to finish the docking task.

6.6 DEVS Simulation and Discussion

When the final generated codes are executed in DEVS-Suite, we can get visualization of models and animation of the simulation. Their animations are supported by "SimView" function illustrated in *Figure 17*. It shows this aircraft docking case in the DEVS-Suite interface. There are four main sections to this screen: the Model Viewer, Simulator Control, SimView, and Tracking Window. The Model Viewer in the top left corner is populated with a list of the components. Immediately below the component list is a box that lists the predefined variables pertaining to the model selected by the user. In this figure, we can see that the component "OperatorPannel2" which has two input ports and two output ports is staying at "checking" phase and have an event that will occur at 30 seconds later. The SimView window on the top right displays the model visually, including any hierarchical components. Below that is the Tracking Window, which contains the standard output console by default, and we can see that after three aircrafts finish docking process, ATT and throughput calculated by the transducer are illustrated. Finally, in the lower left, there is the Simulator Control. From here, the user can control the actions of the simulator.

Run-time display of message on ports and trajectory of variables as two dimensional plots are supported by "TimeView" function. This view in *Figure 18* represents the component "GOS" and it contain multiple graphs pertaining to this component including the state variable "phase" and message on each port. For more precise tracking of variables, detailed data sets can be illustrated in "Tracking

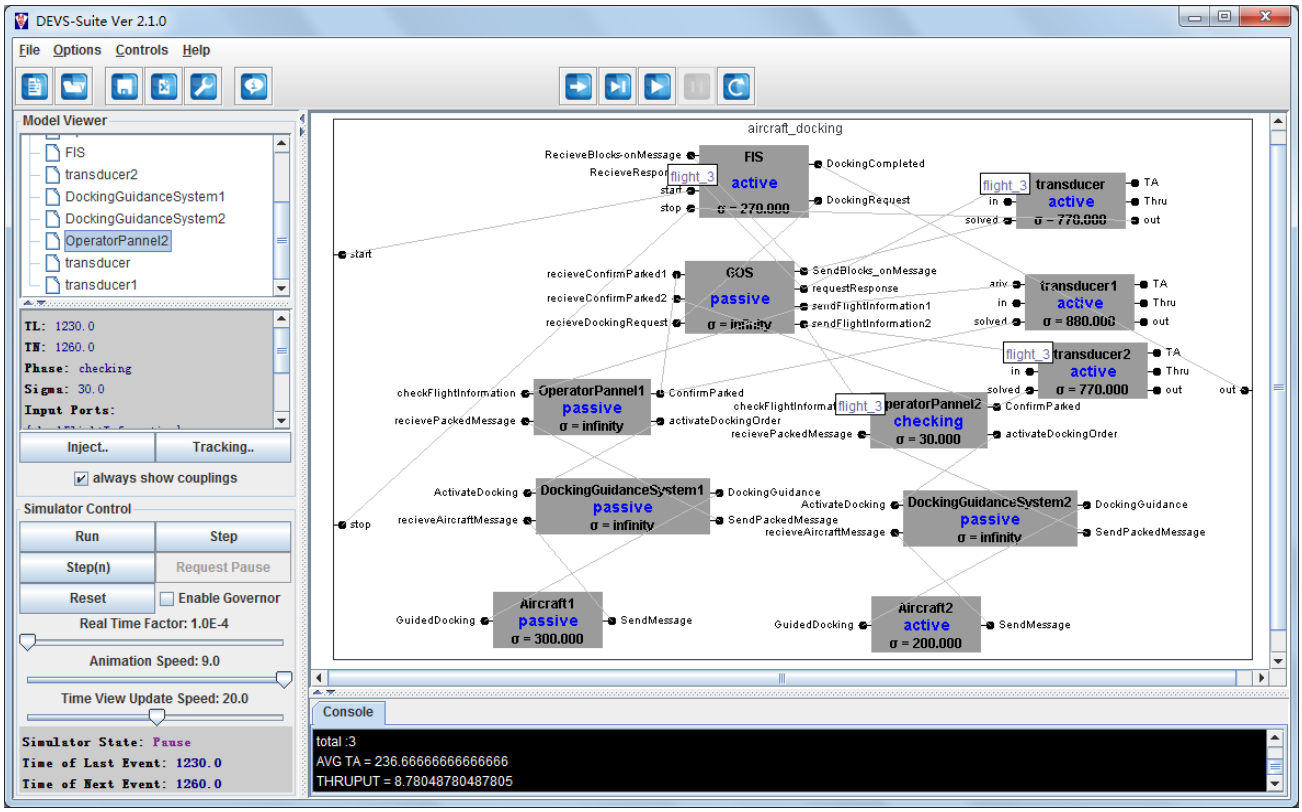


Figure 17. Simulation animation of aircraft docking

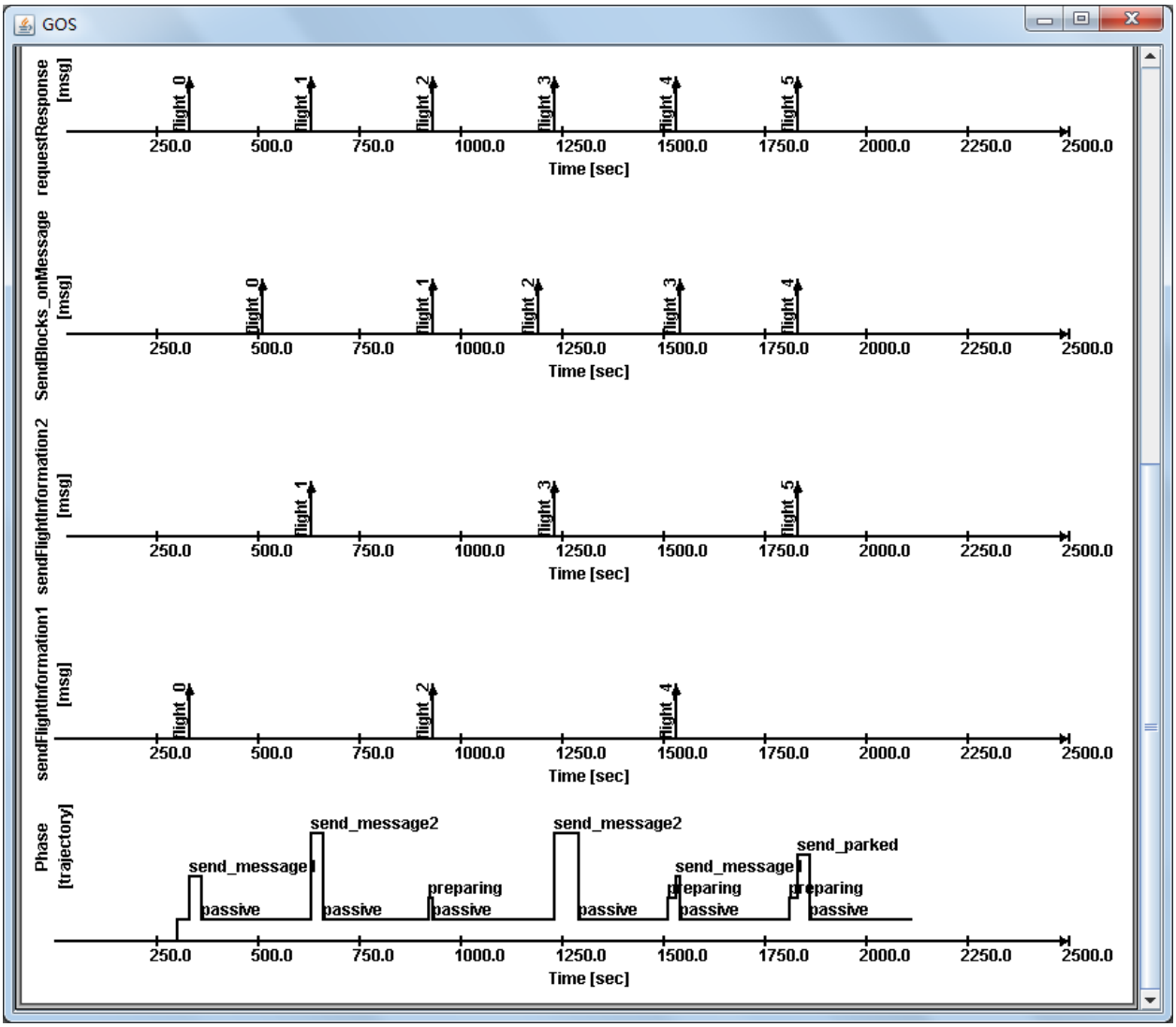


Figure 18. Tracking window of the component GOS

Console | Tracking Log

Time [sec]	1800.0	1810.0	1830.0	1830.0	1830.0	1860.0
GOS	**Phase:** passive **Input Ports:** recieveConfirmParked1: {flight_4} recieveDockingRequest: {flight_5} recieveConfirmParked2: **Output Ports:** sendFlightInformation1: requestResponse: SendBlocks_onMessage: sendFlightInformation2:	**Phase:** preparing **Input Ports:** recieveConfirmParked1: recieveDockingRequest: recieveConfirmParked2: **Output Ports:** sendFlightInformation1: requestResponse: SendBlocks_onMessage: sendFlightInformation2:	**Phase:** preparing **Input Ports:** recieveConfirmParked1: recieveDockingRequest: recieveConfirmParked2: **Output Ports:** sendFlightInformation1: requestResponse: SendBlocks_onMessage: sendFlightInformation2:	**Phase:** send_message2 **Input Ports:** recieveConfirmParked1: recieveDockingRequest: recieveConfirmParked2: **Output Ports:** sendFlightInformation1: requestResponse: {flight_5} SendBlocks_onMessage: sendFlightInformation2: {flight_5}	**Phase:** send_parked **Input Ports:** recieveConfirmParked1: recieveDockingRequest: recieveConfirmParked2: **Output Ports:** sendFlightInformation1: requestResponse: SendBlocks_onMessage: {flight_4} sendFlightInformation2:	**Phase:** pas **Input Port** recieveCon recieveDoc recieveCon **Output Po** sendFlightI requestRes SendBlocks sendFlightI

Figure 19. Tracking log of the component GOS

Log" as *Figure 19* is showing. At the time of 1800 seconds, the component "GOS" receive two inputs on different ports simultaneously. In this situation, according to the rule defined beforehand in E-DEVSML, GOS will transit into phase "preparing" prior to phase "send_parked". This log shows that after GOS staying in phase "preparing" for 20 seconds an internal transition to phase "send_message2" is triggered because internal transitions are also prior to external transitions, and finally the phase "send_parked" is reached and a message on port "SendBlocks_onMessage" is

sent off. These simulation tracking results in Figure 17,18,19 are all very useful to validate the FR of the system, and also prove the applicability of our approach and extended features in E-DEVSML.

Next, we'll focus on the simulation results of NFR. After 16 aircrafts docked with different processing time, NFR statistics including ATT and throughput are shown in Figure 20 and 21. With the increase of number of docked aircrafts, ATT and throughput also gradually stabilize. We can see that DGS1 has better performance than DGS2, the total ATT of GOS calculated by transducer is always less than 300 seconds. So this docking system may have the possibility to cope with the requirement of 5 flights per minutes. In *Figure 21*, total throughput of GOS is less than the throughput sum of two DGS, that's because GOS usually need some preparing time, and sequential processing of messages lead to the delay of GOS's turnaround time.

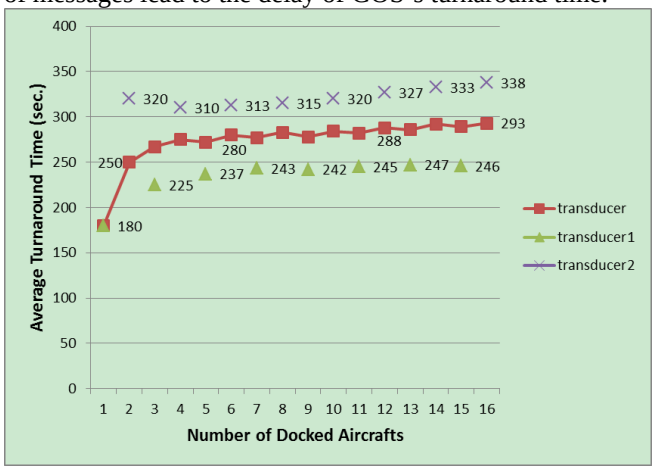


Figure 20. ATT with increasing number of docked aircrafts

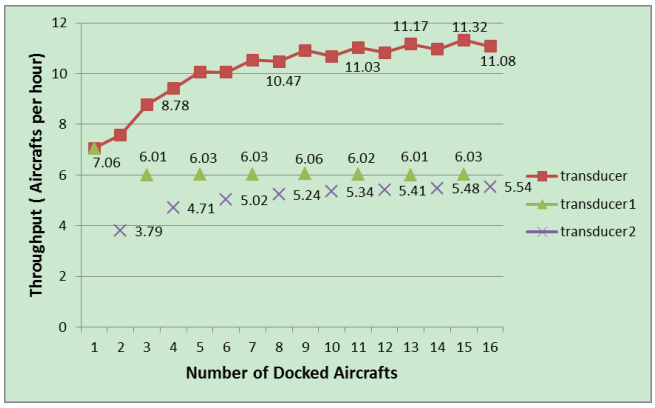


Figure 21. Throughput with increasing number of docked aircrafts

Figure 22 and *23* show the measure defined in the scenario and the values of these metrics obtained in 10 simulation runs. As can be seen, the total ATT ranges from 260 to 300 during all the simulation runs, and the same metric captured by transducer2 fluctuate more wildly than others; however for the results of throughput, all the transducers represent quite stable data. According to these simulation results of throughput, we are sure about that the max throughput of this docking system is 11 aircrafts per hour. If this performance can't satisfy the real requirement, adjustments and improvements on this SOA design should be made. To sum up, all these features of DEVS framework precisely in earlier design phase.

The simulation of the SOA execution provided information that is needed to make decisions about the design of the system. This study focused on a concrete scenario related to the FR and NFR; the simulation shows the behavior of the system and calculates quantitative information to validate these performance attributes and to find the possible causes that can affect the measures. The reports show information not only to validate the FR specified in the previous SoaML models but also NFR statistics to analyze each component. This information helps the architects to determine if the architecture fulfills the quality requirements, and system measures can be used to decide between different design alternatives. By the way this simulation framework based on DEVS can also be used in other quality aspects such as availability and reliability (Pasqua Roberto, 2014).

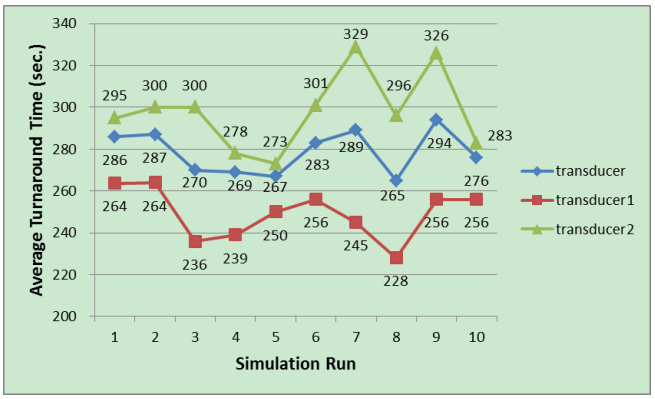


Figure 22. ATT in 10 simulation runs

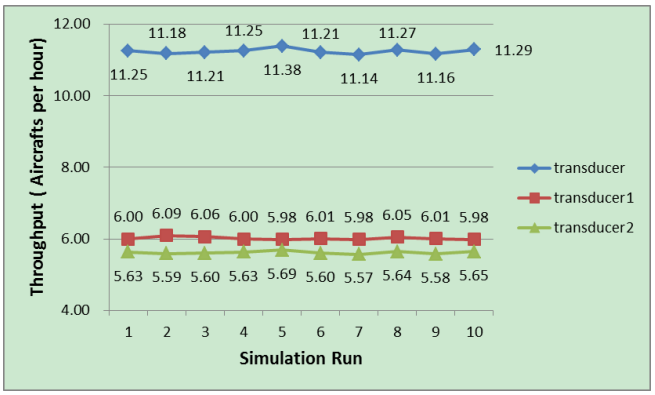


Figure 23. Thoughput in 10 simulation runs

When we started up this research case, we also tried to use other methods including CPN and DUNIP. The major

problem with CPN is that most model transformation processes are manual and the CPN-tool is less convenient and flexible than DEVS-suite. DUNIP provide a comprehensive solution for executable architecture construction from static architecture, but modeling using DEVSML with limitation of FD-DEVS always troubles us. For instance, the State Machine Diagram of GOS on *Figure 10* has complicated state transitions, and we could not directly transform it into DEVSML and Java codes because it may need a semi-automated transformation process with some codes added manually.

7. CONCLUSIONS AND FUTURE WORK

We have presented a model-driven approach to design and validation of SOA using SoaML in conjunction with E-DEVSML, which is a guide to be adapted in a generic service-oriented circumstance. E-DEVSML enhances its capability to describe complex behavior of systems without the limitation of FD-DEVS. It helps modelers to realize automated transformation from static SOA models into executable codes to conduct a systematic simulation of the architecture. Our approach features many advantages:

(1) It is flexible and comprehensive as it is suitable for validation of FR and NFR in SOA development. To fulfill these requirements, this approach based on DEVS framework integrate different perspectives into a more comprehensive evaluation of the designs, and it is adaptable to different contexts and quality objectives of an evaluation.

(2) It provides better usability with modular and hierarchical features than other formalisms such as Petri Nets or the Markov process. Furthermore, E-DEVSML encapsulates the complexity of the simulation technique and keeps the model in a higher level, while the EF captures the architect's objectives and how they impact the SOA model. These features and graphical representation tools could make this approach easier to be learned and used by architects.

(3) It bridges the gap between MDSE and DEVS framework using E-DEVSML as a model transformation intermediary. It is innovative as it combines MDA, SOA and DEVS methodologies, taking advantage of each one in order to address complex system issues.

The configuration of SOA model and EF need the estimation of a set of parameters, and more accurate parameters will produce more precise results. In this way, our approach assumes that architects or experts are able to specify these initial parameters. This requirement may represent the main limitation of our approach because this information may not always be available. However, specific information can be estimated from historical data, similar SOA projects, expert judgment, etc. As the current status of E-DEVSML is still in a static-structure phase, our future work also includes description of DEVS dynamic-structure, which permits the structure of the coupled model to change over time.

8. ACKNOWLEDGMENT

The work described in this paper was supported by the National Natural Science Foundation of China under Grant No. 61232007, 91118004 and the Innovation Program of Shanghai Municipal Education Commission (No. 13ZZ023).

9. REFERENCES

Service oriented architecture Modeling Language (SoaML), Version 1.0. Object Management Group, Document ptc/2009-12-10 (2009). http://www.omg.org/spec/SoaML/

Ardagna, Claudio A., Ernesto Damiani, & Kouessi AR Sagbo (2013). Early assessment of service performance based on simulation, *Proceedings of the 2013 IEEE International Conference on Services Computing (SCC 2013)*, Santa Clara, CA, USA, 2013, pp. 33-40.

Ligang He, Kewei Duan, Xueguang Chen, Deqing Zou, Zongfen Han, Ali Fadavinia, & Stephen A Jarvis (2011). Modelling workflow executions under role-based authorisation control. *Proceedings of the 2011 IEEE International Conference on Services Computing (SCC 2011)*, Washiogton, DC, USA, 2011, pp.200-208.

Vangheluwe, & Hans LM (2000). DEVS as a common denominator for multi-formalism hybrid systems modelling. *Computer-Aided Control System Design, 2000. CACSD 2000. IEEE International Symposium on.* Anchorage, AK, 2000. pp. 129-134.

Sarjoughian HS, Kim S, Ramaswamy M, & Yau SS. A simulation framework for service-oriented computing systems(2008). *Simulation Conference, 2008. WSE 2008.* Austin, TX, 2008, pp. 845–853.

Muqsith MA, Sarjoughian HS, Huang D, & Yau SS(2010). Simulating adaptive service-oriented software systems. *Simulation 2010.* 87(11), pp. 915-931.

Madhoun R. Web service-based distributed simulation of discrete event models. Canada: Carleton University, 2006.

Wainer GA, Madhoun R, &Al-Zoubi K. Distributed simulation of DEVS and Cell-DEVS models in CD++ using Web-Services(2008). *Simulation Modelling Practice and Theory*. 16(9), pp.1266–1292.

Tsai WT, Fan C, Chen Y, & Paul R (2006). A service-oriented modeling and simulation framework for rapid development of distributed applications. *Simulation Modelling Practice and Theory.* 14(6), pp.725–739.

Tsai WT, Wei X, Cao Z, Paul R, Chen Y, &Xu J(2007). Process specification and modeling language for service-oriented software development. *In: 11th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS '07)*. Sedona, AZ, 2007, pp.181–188.

Jia L and Zhang HM. Research on service oriented distributed M&S framework. J Syst Simul 2007; 19: 4680–4684.

Narayanan S, &Mcilraith S(2003). Analysis and simulation of web services. *Computer Networks.* 42(5). pp. 675–693.

Mittal S, Risco-Martin JL, &Zeigler BP. DEVS/SOA: A Cross-platform framework for net-centric modeling and simulation in DEVS unified process. Simul Trans 2009; 85: 419–450.

Jianpeng Hu, Linpeng Huang, Bei Cao, &Xuling Chang(2014). Executable Modeling Approach to Service Oriented Architecture Using SoaML in Conjunction with Extended DEVSML. *SCC 2014 – Proceedings of the*

11th IEEE International Conference on Services Computing, Anchorage, AK, 2014, pp.243-250.

Zeigler, B. P., T. G. Kim, and H. Praehofer. Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems Second Edition: Academic Press. 2000.

Mittal, S, Zeigler, BP, &Hwang, MH (2013), XFDDEVS: XML-Based Finite Deterministic DEVS, Retrieved Jan 2013 from http://www.duniptechnologies.com/research/xfddevs/.

Hong. KJ, &Kim, TG (2006), DEVSpecL-DEVS specification language for modeling, simulation and analysis of discrete event systems, *Information and Software Technology*, 48(4), pp.221 - 234, Apr., 2006

Mittal, S., &S. A. Douglass (2012). DEVSML 2.0: The Language and the Stack. *Symposium on Theory of Modeling and Simulation, Spring Simulation Multiconference.* Orlando, FL: SCS. 2012.

Hwang, Moon Ho, & Bernard P. Zeigler(2009). Reachability graph of finite and deterministic DEVS networks. *Automation Science and Engineering, IEEE Transactions on*. 6(3), pp.468-478.

Elvesæter, B., Carrez, C., Mohagheghi, P., Berre, A. J., Johnsen, S. G., & Solberg, A. (2011). Model-driven service engineering with SoaML. *In Service Engineering, Springer Vienna.*pp.25-54.

The Arizona Center for Integrative Modeling and Simulation (ACIMS), DEVS-Suite, Retrieved Jan 2014 from http://sourceforge.net/projects/devs-suitesim/

Pasqua, Roberto, et al. FROM SEQUENCE DIAGRAMS UML 2. x TO FD-DEVS BY MODEL TRANSFORMATION. European Simulation and Modelling (2012).

Verónica Bogado, Silvio Gonnet, & Horacio Leone, Modeling and simulation of software architecture in discrete event system specification for quality evaluation [J], Simulation January 27, 2014.

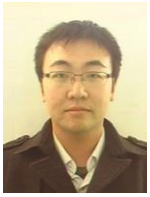
Authors

Jianpeng Hu received his BS and MS degrees from East China University of Science and Technology (ECUST), Donghua University(DHU) in 2003 and 2006, respectively. He is a lecturer of computer science in the College of Electrical and Electronic Engineering, Shanghai University of Engineering Science (SUES). Also, he is currently working toward the PhD degree in the department of computer science and engineering at the Shanghai Jiao Tong University (SJTU). His research interests include software engineering, formal verification techniques, modeling & simulation, software architecture and system of systems.

Linpeng Huang received his MS and PhD degrees in computer science from Shanghai Jiao Tong University in 1989 and 1992, respectively. He is a professor of computer science in the department of computer science and engineering, Shanghai Jiao Tong University. His research interests lie in the area of distributed systems, formal verification techniques, architecture-driven software development, system of systems, big data analysis and in-memory computing.

Renke Wu received the BS degree in Software Engineering, the MS degree in Computer Software and Theory from the School of Computer at Hangzhou Dianzi University (HDU), China, in 2011 and 2014, respectively. He is currently working toward the PhD degree in the department of computer science and engineering at the Shanghai Jiao Tong University (SJTU). His research interests include software engineering, formal verification techniques, architecture-driven software development, system modeling languages, software architecture and system of systems.

Bei Cao received the BSc degree in computer science and technology from Huazhong University of Science and Technology (HUST) in 2012. He is currently working toward the master degree with the Department of Computer Science and Engineering at Shanghai Jiao Tong University (SJTU). His research interests include software engineering and system of systems.



Xuling Chang received the BSc degree in software engineering from Jilin University (JLU) in 2012. He is currently working toward the MSc degree with the School of Electronic Information and Electrical Engineering at the Shanghai Jiao Tong University (SJTU). His research interests include software engineering, coloured petri nets, semantics and verification of UML/SysML, transformation algorithms between system modeling languages and system of systems modeling and simulation.