Algebraic Method to Model Secure IoT

Yeongbok Choe and Moonkun Lee

Chonbuk National University
664-1 Deokjin-Gu Deokjin-Dong 1Ka
Jeonju, Jeonbuk 561-756, Republic of Korea
e-mail: moonkun@jbnu.ac.kr

Abstract. Process algebra can be considered to be one of the best methods to model IoT systems since it can represent the main properties of things in the systems: communication, movements, deadlines, etc. The best known algebras are *π-calculus* and *mobile ambient*. However there are some limitations to model the different types of movements of the things with secure requirements. π-calculus passes the name of ports for indirect movements unrealistically, and mobile ambient uses ambient to synchronize asynchronous movements forcefully and unnaturally. This paper presents new process algebra, called *δ-calculus*, to model the different types of such synchronous movements for the things in IoT over some target geographical space. A process can be nested in another process, and their configuration will be changed by these movements. Any violation of the secure movements can be detected and prevented by the properties of the movements: synchrony, priority and deadline. To demonstrate the feasibility, a tool, called SAVE, was developed on the ADOxx meta-modeling platform with an emergency medical system, which is one of the best suitable application domains for IoT.

1 Introduction

Internet of Things (IoT) is one of the most promising IT domains supported by smart devices: It consists of things connected together by Internet in order to provide users with intelligent services over some geographical space(A. Whitmore et al. 2015). Since it deals with human lives and assets, as well as environment, it must provide them with complete security and privacy, namely *secure requirements*(B. Larcom, 2014), to meet stakeholders' security needs. Any violation of the requirements may cost human lives and assets as well as destruction of environment. Therefore it is critical to specify and verify all the requirements at the time of modeling the IoT systems with formal methods, as recommended by a number of international standards organizations(IoT-GSI).

Generally formal methods can be classified into the following three mathematical structures: Logic, state machine and process algebra(C Heitmeyer and D. Mandrioli, 1996). Among these, process algebra can be more suitable than other

types to model the behavior of the IoT systems due to formal representation of things as processes, movements as interactions, and time and priority as IoT-related properties, with the support of algebraic equivalences for analysis of behaviors. The most well-known process algebras are π-*Calculus* (R. Milner et al. 1992) and *Mobile Ambient* (L. Cardelli and A. Gordon, 1998) with their evolved versions(H. Lin, 2005).

π-calculus is designed by Robin Milner by extending the basic notion of CCS (R.Milner, 1982) to model the movement of processes based on the notion of value passing. Instead of representing the actual movements of a process (actor) from one process (source) to another process (target), the name of a port of the actor is passed from the source to the target to represent the movement of the actor. It can be considered as the imaginary indirect representation of the actual direct movements.

Mobile Ambient is designed by Luca Cardelli and Andrew D. Gordon with the new notion of ambient, which assists the movements of processes synchronously. Ambient provides a means of controlling asynchronous movements in synchronous manner, but it increases the complexity of specification with the additional dimension of the ambient. Consequently the specification and verification of the movements become very complex and complicated. Further some processes can fall into some deadlock state since no movement is possible for nested ambient.

These algebras and their evolved ones have the limitations to represent additional properties of IoT, such as geographical space of IoT, different types of movements of the things with different modes in time with priority, etc. In order to overcome the limitations, this paper introduces new process algebra, namely, δ-*calculus*(Y. Choe and M. Lee, 2015), with the notion of the following geographical space and the distinctive properties of the movements:

- Geographical space: A geographical space can be defined as a system space, where all of its processes will be both processes and their child processes
- Movement types: There are two types of movements: *In* and *out*. The *in* is an interaction for one process to move into another process; the *out* is an interaction for one process to move out of its parent process.
- Movement modes: There are two modes of movements: *active* and *passive*. The *active* is an autonomous movement of a process to move in its sibling process or to move out of its parent process. The *passive* is a heteronomous movement for a process to be moved into its sibling process or to be moved out of its parent process.
- Time: The temporal properties of the movements are *ready time*, *interaction time*, *execution time*, and *deadline*. In addition, the proper actions for the violations of deadlines can be specified.
- Priority: A discrete level of priority can be assigned to each process. Therefore it is possible to control the accessibility of a process to another process.

These properties will provide the basic features to model the behavior of IoT on a specific geographical space.

The steps of modeling IoT are as follows:

- Specification: The first step is to specify IoT with δ-calculus. All the things in IoT will be specified on a geographical space with the detailed actions, especially the movement actions with time and priority.
- Execution: The next step is to generate the execution tree for the specification. It shows all possible execution paths for the specification.
- Simulation: The next step is to simulate each execution path from the execution tree. It generates a *Geo-Temporal Space* (GTS) diagram for the simulation of each path.
- Verification: The last step is to verify all the secure requirements by model-checking (E. M. Clarke and E. A. Emerson, 1981) on all the simulation diagrams.

The secure requirements in the last step include dependencies among processes/things and their actions, especially prioritized temporal movements. They describe under what restrictions things or processes in IoT should behave.

The modeling approach based on δ-calculus can be considered as one of the most suitable methods to model IoT and verify its secure requirements. In order to demonstrate the justification and feasibility of the approach, a tool, called SAVE (Y. Choe et at. 2015), has been implemented on ADOxx (H. Fill and D. Karagiannis, 2013).

The organization of the paper is as follows. Section 2 introduces the basic notions of δ-calculus, including syntax, semantics and algebraic laws with a simple example. Section 3 represents the models for the calculus: specification, execution and simulation models. Section 4 presents the SAVE tool on ADOxx and demonstrates feasibility of the approach with an IoT example. Finally conclusions are made in Section 5.

2 Method Description: δ-Calculus

δ-calculus is a process algebra to model the behavior of processes by defining distribution of processes on a geographical space and their actions, especially movements in time and with priority.

2.1 Syntax

The syntax of δ-calculus is shown in Fig. 1. The description of each construct is as follows:

- Inaction (*nil*): No action for a process.
- Action (A): Actions performed by a process, such as empty, communication and movements.
- Priority ($P_{(n)}$): The priority of the process P represented by a natural number $n \geq 0$. The higher number represents the higher priority.

- Nesting ($P[Q]$): P contains Q.
- Channel ($P\langle r \rangle$): A channel r of P to communicate with other processes.
- Choice ($P + Q$): Only one of P and Q will be selected nondeterministically for execution if their priorities are same. If the priorities are different, the process with the higher will be selected.
- Parallel ($P \mid Q$): Both P and Q are running concurrently.
- Sequence ($A \bullet P$): P follows after action A.
- Empty action ($\varnothing$): Do nothing for 1 unit time.
- Communication ($r(msg)$; $r(\overline{msg})$): P communicates with other process connected with the channel r to pass the message msg. The mode of passing is indicated by $\overline{msg}$ for sending and msg for receiving.
- Movement request ($m_t^p(k)\ P$): A request for a movement to or from a target process with a key. Here t, p and k represent the time, priority, and the password for the movement, respectively. The timing properties are described in detail in Section 2.4.
- Movement permission ($P\ m_t^p(k)$): The permission for the movement from the above request. The timing properties are described in detail in Section 2.4.

$$
\begin{aligned}
&P ::= nil \mid A \mid P_{(n)} \mid P[Q] \mid P\langle r_t \rangle \mid P + Q \mid P \mid Q \mid A \bullet P \\
&A ::= \varnothing \mid r_t(\overline{msg}) \mid r_t(msg) \mid M \\
&M ::= m_t^p(k)\ P \mid P\ m(k)_t^p \\
&m ::= in \mid out \mid get \mid put
\end{aligned}
$$

Fig. 1 Syntax of δ-Calculus.

Basically the movements are synchronous: The request to enter or move out requires permission from its target process in the autonomous case, and, similarly, the request to make another process to enter into or move out of itself has to require permission from the target process in the heteronomous case, too.

For example, the *Producer-Buffer-Consumer* (*PBC*) system with three main processes *Producer* (*P*), *Buffer* (*B*) and *Consumer* (*C*) with an additional process *Resource* (*R*) can be represented, in δ-Calculus, in order to pass *R* to *C* through *B*, as follows:

$PBC1 := \underline{P[R]} \mid B \mid C;$
$P := PB(\overline{send}) \bullet put\ R \bullet exit;$
$B := PB(\underline{send}) \bullet get\ R \bullet CB(\overline{need}) \bullet put\ R \bullet exit;$
$C := CB(\underline{need}) \bullet get\ R \bullet exit;$
$R := P\ put \bullet B\ get \bullet B\ put \bullet C\ get \bullet exit;$

In *PBC*, *P* sends the *send* message to *B* through the *PB* channel to inform *B* to *put* *R*, and *put*s *R* out of its space. After receiving the message, *B get*s *R* and waits for the *need* message from *C* before *put*ting *R*. After receiving the message, *B put*s *R* out of its space. C sends the *need* message to *C* to *get R*. If *C* receives the message, *C get*s *R*. *R* has been properly transported to *C* through *B*.

2.2 Semantics: Transition Rules

The semantics of δ-calculus are defined in Tables 1 and 2 as transition rules. The description of each rules in Table 1 are as follows:

- Action: It is a transition rule for the execution of a communication action $r(a)$ on a channel r with a message a. It does not require any premise for the transition, after which P will be executed next.
- Choice: *ChoiceL* and *ChoiceR* transition rules can be equally chosen under the same premise. *ChoiceP* rule is determined by priority.
- Parallel: The transition of a process in *ParIL* and *ParIR* rules does not influence its parallel process. But *ParCom* rule requires that two parallel processes must synchronously interact with each other in order to make their corresponding transitions to be occurred.

Table 1 Communication Semantics of δ-calculus

Action	$\frac{-}{r(a)\bullet P \xrightarrow{r(a)} P}$	ChoiceL	$\frac{P \xrightarrow{A} P'}{P+Q \xrightarrow{A} P'}$
ChoiceR	$\frac{Q \xrightarrow{A} Q'}{P+Q \xrightarrow{A} Q'}$	ParlL	$\frac{P \xrightarrow{A} P'}{P \mid Q \xrightarrow{A} P' \mid Q}$
ParlR	$\frac{Q \xrightarrow{A} Q'}{P \mid Q \xrightarrow{A} P \mid Q'}$	ParCom	$\frac{P \xrightarrow{A} P', Q \xrightarrow{\bar{A}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$
NestO	$\frac{P \xrightarrow{A} P'}{P[Q] \xrightarrow{A} P'[Q]}$	NestI	$\frac{Q \xrightarrow{A} Q'}{P[Q] \xrightarrow{A} P[Q']}$
Nest-Com	$\frac{P \xrightarrow{A} P', Q \xrightarrow{\bar{A}} Q'}{P[Q] \xrightarrow{\tau} P'[Q']}$		

The description of each rules in Table 2 are as follows:

- Movement: *In*, *Out*, *Get* and *Put* transition rules represent the general synchronous movements. Each movement must have its corresponding co-movement. Similarly, *InP*, *OutP*, *GetP*, *PutP* rules are for the movements with priority.

The timing requirements for the movement transition rules are described in detail in Section 2.4.

Table 2 Movement Semantics of δ –Calculus

In	$\dfrac{P \xrightarrow{in\ Q} P', Q \xrightarrow{P\ in} Q'}{P \mid Q \xrightarrow{\delta} Q'[P']}$	Out	$\dfrac{P \xrightarrow{out\ Q} P', Q \xrightarrow{Q\ out} Q'}{Q[P] \xrightarrow{\delta} P' \mid Q'}$
Get	$\dfrac{P \xrightarrow{get\ Q} P', Q \xrightarrow{P\ get} Q'}{P \mid Q \xrightarrow{\delta} P'[Q']}$	Put	$\dfrac{P \xrightarrow{put\ Q} P', Q \xrightarrow{Q\ put} Q'}{P[Q] \xrightarrow{\delta} P' \mid Q'}$
InP	$\dfrac{P_{(n)} \xrightarrow{in^P\ Q} P_{(n)}'}{P_{(n)} \mid Q_{(m)} \xrightarrow{in^P\ Q} Q_{(m)}[P_{(n)}']} (n > m)$		
OutP	$\dfrac{P_{(n)} \xrightarrow{out^P\ Q} P_{(n)}'}{Q_{(m)}[P_{(n)}] \xrightarrow{out^P\ Q} P_{(n)}' \mid Q_{(m)}} (n > m)$		
GetP	$\dfrac{P_{(n)} \xrightarrow{get^P\ Q} P_{(n)}'}{P_{(n)} \mid Q_{(m)} \xrightarrow{get^P\ Q} P_{(n)}'[Q_{(m)}]} (n > m)$		
PutP	$\dfrac{P_{(n)} \xrightarrow{put^P\ Q} P_{(n)}'}{P_{(n)}[Q_{(m)}] \xrightarrow{out^P\ Q} P_{(n)}' \mid Q_{(m)}} (n > m)$		
InN	$\dfrac{P \xrightarrow{in\ Q} P', Q \xrightarrow{P\ in} Q'}{P \mid Q[R] \xrightarrow{\delta} Q'[P' \mid R]}$	GetN	$\dfrac{P \xrightarrow{get\ Q} P', Q \xrightarrow{P\ get} Q'}{P[R] \mid Q \xrightarrow{\delta} P'[R \mid Q']}$

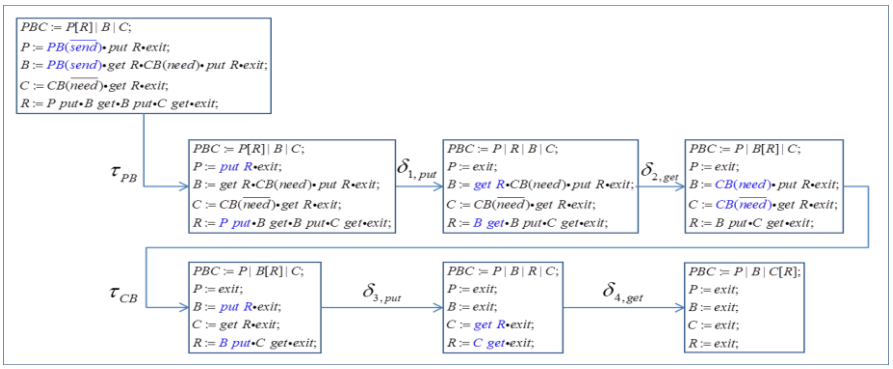


Fig. 2 A Sequence of Transitions for PBC Example Based on Semantics.

For example, Fig. 2 shows the execution of the *PBC* example by the transition rules of the communication (τ) in Table 1 and the movements (δ) in Table 2 as follows:

1) τ_{PB} : The communication between $P : PB(\overline{send})$ and $B : PB(send)$.
2) $\delta_{1,put}$: The *put* movement between $P : put\ R$ and $R : P\ put$.
3) $\delta_{2,get}$: The *get* movement between $B : get\ R$ and $R : B\ get$.
4) τ_{CB} : The communication between $C : CB(\overline{need})$ and $B : CB(need)$.

5) $\delta_{3,put}$: The *put* movement between $B:put\ R$ and $R:B\ put$.

6) $\delta_{4,get}$: The *get* movement between $C:get\ R$ and $R:C\ get$.

2.3 Laws

The algebraic laws are defined in Table 3. These laws are used to restructure the textual configuration of processes in a system or to reduce synchronous binary interactions, that is, communication and movements, between two interactive processes as a means of execution.

For example, the execution of the *PBC* example is supported by these algebraic laws as follows. Note that ':' implies 'of', such that $P:PB(\overline{send})$ implies the $PB(\overline{send})$ action of *P*:

$$
\begin{aligned}
PBC &= P[R] \mid C \mid B \\
&= P[R] \mid (C \mid B) && \text{(by associative law on |)} \\
&= (P[R] \mid B) \mid C && \text{(by commutative \& associative law on |)} \\
&= (P:PB(\overline{send}) \mid B:PB(send)) \mid C && \text{(by binary communication operation)} \\
&= (P:put\ R[R:P\ put] \mid B) \mid C && \text{(by binary } put \text{ movement Operation)} \\
&= P \mid (R:B\ get \mid B:get\ R) \mid C && \text{(by associative laws \& } get \text{ operation)} \\
&= P \mid (B:CB(need)[R] \mid C:CB(\overline{need})) && \text{(by binary communication operation)} \\
&= (P \mid B:put\ R[R:B\ put]) \mid C && \text{(by binary } put \text{ movement Operation)} \\
&= P \mid B \mid (R:B\ get \mid C:get\ R) && \text{(by associative laws \& } get \text{ operation)} \\
&= P \mid B \mid C[R]
\end{aligned}
$$

Table 3 Laws of δ-calculus

Choice(1)	$P+P \equiv P$	Choice(2)	$P+Q \equiv Q+P$
Choice(3)	$(P+Q)+R \equiv P+(Q+R)$	Parallel(1)	$P \mid nil \equiv P$
Parallel(2)	$P \mid Q \equiv Q \mid P$	Parallel(3)	$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$
Nesting(1)	$P[nil] \equiv P$	Nesting(2)	$R[P]+R[Q] \equiv R[P+Q]$
Distributive (1)	$P \mid (Q+R) \equiv (P \mid Q)+(P \mid R)$	Distributive (2)	$(A_1+A_2).P \equiv A_1.P+A_2.P$

2.4 Time Property

There are four types of timing specifications in δ-calculus: system, process, communication and movements, as follows:

- System: The time properties for *S* are defined as a tuple (*r*, *e*, *d*) in the form of subscript to *S*, that is, $S_{(r,e,d)}$, where *r*, *e* and *d* imply *ready time*, *execution*

time and *deadline* for S, respectively. Restrictions are $r \le e$, $e \le d$, and $r + e \le d$. Note that all numbers are natural.

- Process: The properties for *P* are defined as $P_{(r,e,d)}$, same as those of *S*.
- Communication (τ): It is a synchronous interaction with a *send* action and its *receive* action. The time properties for both actions are as follows:
  - Send: The properties are defined as $send_{([l_s,u_s],e_s,d_s)}$, where $[l_s,u_s]$, e_s and d_s imply *time period* for synchronization, *execution time* and *deadline*, respectively. Restrictions are $l \le u$, $(u-l) \le e$ and $l + e \le d$.
  - Receive: The properties are defined as $receive_{([l_r,u_r],e_r,d_r)}$, same as those of *send*.
- Movements (δ): Since it is a synchronous interaction as communication, the specifications are same as those of *send*, for the movement actions that request movements (*requester*), and *receive*, for the movement actions that permit the request (*permitter*). The additional time for the movement is defined as $move_{([l,u],[e,e_m],d_s)}$, where e_m is the execution time for the movement. It is only applicable to the process that actually moves, namely, *mover*.

For example, the following is a *PBC* example with timing properties. There are the timing specifications for system, processes and actions. Note that the actual time for the movement of *R* is 8 in total, which is less than the execution time 10 of *R*.

$$PBC1_{[0,15,25]} = \underline{P_{[0,10,20]}}[R_{[0,10,15]}] \mid B_{[0,15,20]} \mid C_{[0,10,20]};$$
$$P_{[0,10,20]} = PB(\overline{send})_{([0,2],1,-)} \bullet put\ R_{([0,2],1,-)} \bullet exit;$$
$$B_{[0,15,20]} = PB(\overline{send})_{([1,3],1,-)} \bullet get\ R_{([0,2],1,-)} \bullet CB(need)_{([1,3],1,-)} \bullet put\ R_{([0,2],1,-)} \bullet exit;$$
$$C_{[0,10,20]} = CB(\overline{need})_{([0,2],1,-)} \bullet get\ R_{([0,2],1,-)} \bullet exit;$$
$$R_{[0,10,15]} = P\ put_{([0,2],[1,2],4)} \bullet B\ get_{([0,2],[1,2],4)} \bullet B\ put_{([0,2],[1,2],4)} \bullet C\ get_{([0,2],[1,2],4)} \bullet exit;$$

3 Method Conceptualization: Models for δ-Calculus

There are three different types of models for δ-calculus: specification, execution and simulation.

3.1 Specification Model: System and Process

A system model is defined as the following tuple: $S = (P, C, I, T)$, where *P* is a set of processes described by δ–calculus, *C* is a set of channels between processes, *I* is the set of inclusion relations among the processes in *P*, and *T* is the global

clock. The graphical icons for System View are shown in Table 4, with its meta-modeling definition.

Table 4 Process Modeling Definition with Icons.

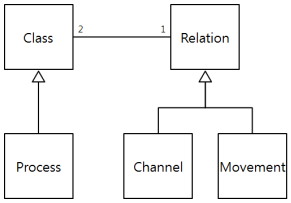
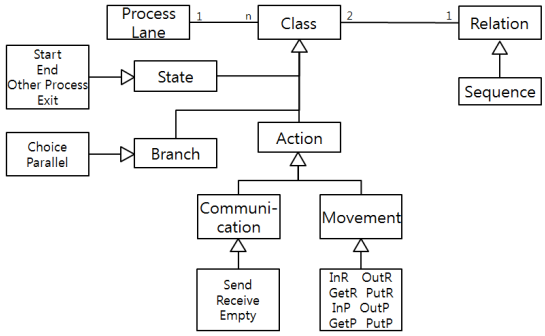
Meta-Model	Icon	
	Process	
	Channel	
	Movement	

Table 5 Process Model Definition with Icons.

Meta-Model						
						
Icon						
Process Lane		Start		End		Other Process	
Exit		Choice		Parallel		Send	
Receive		Empty		In R		Out R	
Get R		Put R		In P		Out P	
Get P		Put P		Sequence			

System View is the view with processes interacting with each other over some geographical space. In the view, a process will be represented as a node and a channel as an edge between nodes. A node can be nested in another node. Any movement can be represented as a movement edge, and it changes the configuration of the view.

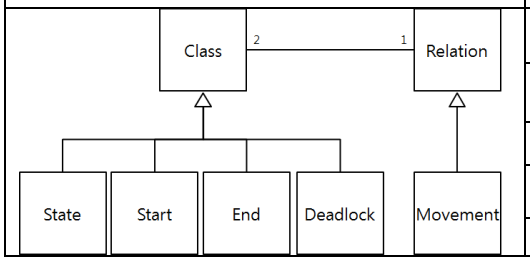
A process model is defined as a sequence of actions: $P = (a_1 \bullet a_2 \bullet \cdots \bullet a_n, t)$, where a_i's are the actions in order defined in δ–calculus and t is a local clock. The

graphical icons for Process View are shown in Table 5, with its meta-modeling definition.

As stated, System and Process Views represent graphically a system and its processes, respectively. For the complete specification of a system, there should be 1-1 correspondence between each node in System View and its Process view. Further for each interaction, there should be 1-1 correspondence between a synchronizing action of a synchronizer and the synchronized action of its synchronizee. These correspondences guarantee the proper structural condition for syntactical completeness. If not, the proper execution of the system cannot be performed due to syntactic inconsistence.

Process View shows the detailed actions of each process in System View. There is 1-1 correspondence between each action of process in δ-calculus and Process View icon, except *start* and *end* icons.

Table 6 Execution Model Definition with Icons.

Meta-Model	Icon
	Start  End  State  Deadlock  Sequence

3.2 Execution Model: Labelled Transition System

The execution model for the system is based on the notion of system state and its transition. Since the system consists of processes, its state is defined as a set of states of its processes with the two additional state variables: a set of inclusion relations among processes and a global clock. The inclusion relations are due to the changes of the system configuration as a result of the movement actions, and every action consumes time. In order to control such a temporal synchrony of actions, it is necessary to define two types of clocks: 1) the global clock for the system and 2) the local clocks for each processes. The former is named as the global clock (T) and the latter is named as local clocks (t_i) for each process P_i.

A *process state* is $p_i = (\bullet_i, t_i)$, where $\bullet_i$ is the position just after a_i in the sequence of actions in P and t_i is the time of the local clock. There are two special states: *start* and *final*.

A *system state transition* is $p_{j-1} \xrightarrow{a_j^t=(a_j,t_j)} p_j$, where a transition occurs from p_{j-1} to p_j by an action a_j, which takes the time t_i.

The transition is based on the rules defined in the semantics of δ-calculus. For example, if $p_0 = (0,0)$ and the first action of sending a message on the channel c in P occurs in the time 2, then $p_1 = (1,2)$ by $p_0 \xrightarrow{c(\bar{m}),2} p_1$.

The *execution model* for processes in δ-calculus is $p_0 \xrightarrow{a_1^{t_1}} p_1 \xrightarrow{a_2^{t_2}} p_2 \longrightarrow \cdots \longrightarrow p_f$, where each p_i is a process state, which is transited to the next state after performing an action a_i in the time t_i. There are two special states, s_0 and s_f, for the *start* and *final* states, respectively.

It is possible not only to have multiple transitions from one process state, but also to one process state. And it is possible not only to have no *final* state, but also to have multiple *final* states. However there should be a single *start* state.

A *system state* is $s_i = \left((p_{1,i}, p_{2,i}, \cdots, p_{n,i}), I_i, C_i, T_i\right)$, where each $p_{j,i}$, I_i, C_i and t_i represent the state of each process P_i, a set of the inclusion relations among processes, a set of channels and the global time, respectively.

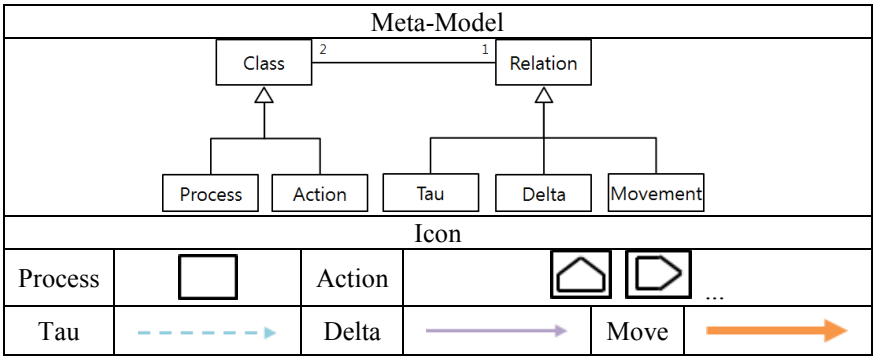
A *system state transition* between one system state to another state is $s_j \xrightarrow{i_j^t} s_{j+1}$, where $i_j = \left((P_a : a, P_{\bar{a}} : \bar{a}), t_j\right)$. Here s_j and s_{j+1} are system states, i_j^t is a synchronous interaction between processes P_a and $P_{\bar{a}}$ with the action a of P_a and the action $\bar{a}$ of $P_{\bar{a}}$ in the time t_j. The transition is based on the rules defined in the semantics of δ-calculus.

The system execution model for δ-calculus is the labelled transition system: $s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_2 \longrightarrow \cdots \longrightarrow s_f$. Each s_j represents a system state, and i_k does a system state transition. There are two special states, s_0 and s_f, for *first* and *final* states, respectively.

It is possible not only to have multiple transitions from one system state, but also to one system state. And it is possible not only to have no *final* state, but also to have multiple *final* states. However there should be a single *start* state.

The graphical icons for Execution View are shown in Table 6, with its meta-modeling definition. The view is represented as *Execution Tree* (ET).

Table 7 Simulation Model Definition with Icons.



3.3 Simulation Model: GTS Diagram

The *simulation model* for system *S* is represented by the following tuple: $\mathbf{S} = (\mathbf{P}, \mathbf{I})$, where $\mathbf{P} = \{\mathbf{P}_1, \cdots, \mathbf{P}_n\}$ with $\mathbf{P}_i$ for each process P_i in the system *S*, and $\mathbf{I} = \{\mathbf{i}_1^{t_1}, \cdots, \mathbf{i}_m^{t_m}\}$ with $\mathbf{i}_j^{t_j}$ for each interaction in the system transitions. Further each $\mathbf{P}_i$ is represented by the following sequence: $\mathbf{P}_i = \left\langle \mathbf{a}_{j,1}^{t_{j,1}}, \cdots, \mathbf{a}_{j,j_n}^{t_{j,j_n}} \right\rangle$, where each $\mathbf{a}_i$ is the simulated action in the time t_i.

The graphical icons for Simulation View are shown in Table 7, with its meta-modeling definition. The view is represented as *Geo-Temporal Space* (GTS) diagram, where all the processes, their actions and the interactions among them are represented as GTS blocks with the following restrictions:

- Syntactic restriction:
  - Action blocks in P_i cannot be overlapped in any space at any time in the GTS block of P_i, that is, $\mathbf{P}_i$.
  - If $P \mid Q$, there is no overlap in their GTS blocks, but an overlap in time.
  - If $P[Q]$, there is an overlap of the *Q* GTS block over the *P* GTS block during the period of time for inclusion.
- Semantics restrictions for communication:
  - The sender *P* GTS block and the receiver *Q* GTS block must be in the same time period in their GTS blocks.
- Semantic restriction for movements:
  - For the active *in* movement, the mover *Q* GTS block must be in the same space with the target *P* GTS block at the same period of time.
  - For the active *out* movement, the mover *Q* GTS block must be in the target *P* GTS block at the same period of time.
  - For the passive *get* movement, the mover *Q* GTS block must be in the same space with the target *P* GTS block at the same period of time.
  - For the passive *put* movement, the mover *Q* GTS block must be in the target *P* GTS block at the same period of time.

4 Proof of Concept

4.1 SAVE Tool

A tool, called SAVE (*Specification, Analysis, Verification Environment*), for δ-calculus has been developed on ADOxx (H. Fill and D. Karagiannis, 2013), as shown in Fig. 3. It consists of four basic components as follows:

- Modeler: It provides capability to specify System and Process Views.
- EM Generator: It generates an execution model, in *Execution Tree* (ET), for

the views and makes each path of the model to be selected for simulation.

- Simulator: It generates a model for the selected simulation, in a GTS diagram.
- Verifier: It verifies the secure requirements of the system by model-checking on the diagrams.

The graphical representations of the models in SAVE are designed by the ADOxx Development Tool, and the procedures of its components are built from the ADOxx libraries. The detailed logics of the procedures are programmed in the ADOScript language. ADOxx provides three layers to implement mechanisms and algorithms for SAVE:

- First layer: The pre-defined functionality, a basic set of features most commonly used by modeling tools.
- Second Layer: Approximately 400 APIs for the generation of objects, editing of their properties, etc.
- Third layer: Ways of interaction to outside of ADOxx. The simple interaction is by exporting and importing XML files.

SAVE uses the functionalities of the first layer to implement the graphical elements and attributes of the graphic models, and it uses those of the second layer to implement Modeler, EM Generator, Simulator and Verifier.

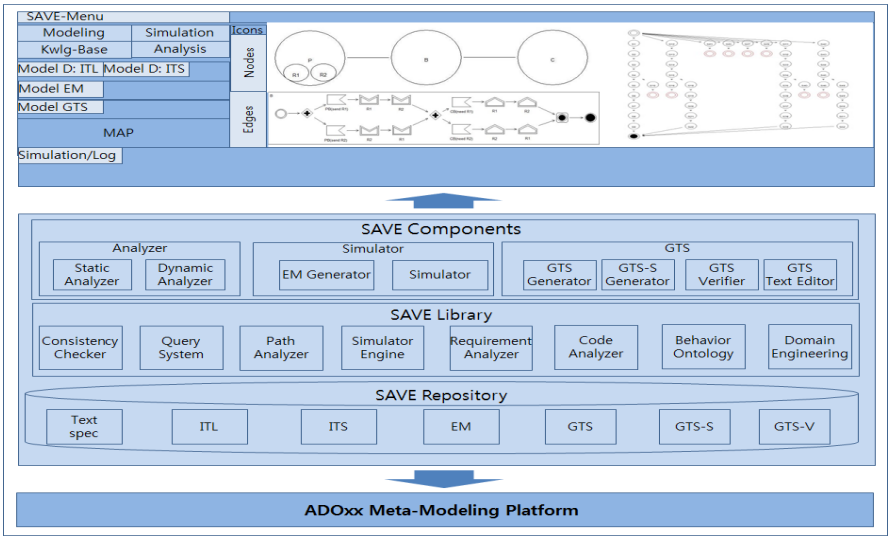


Fig. 3 SAVE Architecture.

4.2 IoT Application: EMS Example

Emergency Medical System (EMS) is one of the best applications for IoT(A. Whitmore et al. 2015). The system provides drivers with smart emergency medical services at the time of accidents. The services are realized by integrated intelli-

gent systems between the 911 and the medical information centers on smart emergency calls from smart cars or phones.

$$
\begin{aligned}
&EMS := Car[Drv] \mid 911[Amb] \mid Hsp \mid Mdc;\\
&Car := \left(CS1(\overline{crash})_{[0,60]} \bullet Drv\ out + CS2(\overline{crash})_{[61,-]} \bullet AC(open) \bullet put\ Drv\right);\\
&911 := \left(EC(call) \bullet AM(\overline{call}) + EC(autocall) \bullet AM(\overline{auto}) \bullet ME(\overline{pi, hi})\right) \bullet Amb\ out \bullet EH(pi);\\
&Amb := \left(AM(call) \bullet out\ 911 \bullet get\ Drv + AM(auto) \bullet out\ 911 \bullet AC(\overline{open}) \bullet get\ Drv \bullet MA(pi)\right) \bullet\\
&\quad AS(\overline{em}) \bullet in\ Hsp \bullet put\ Drv;\\
&Mdc := \left(\varnothing + ME(pi, hi) \bullet MA(\overline{pi}) \bullet MH(\overline{pi})\right) \bullet MH(pi);\\
&Drv := \left(CS1(crash) \bullet EC(\overline{call}) \bullet out\ Car + CS2(crash) \bullet EC(\overline{autocall}) \bullet Car\ put\right) \bullet Amb\ get \bullet\\
&\quad AS(em) \bullet Amb\ put \bullet HS(tr);\\
&Hsp := \left(\varnothing + MH(pi)\right) \bullet Amb\ in \bullet HS(\overline{tr}) \bullet EH(\overline{pi}) \bullet MH(\overline{pi});
\end{aligned}
$$

Fig. 4 EMS Example in δ-calculus

4.2.1 Specification Model

The textual specification for the example is shown in Fig. 4. It consists of Car, Driver, 911, Ambulance, Medical Center and Hospital. The scenario for the smart medical service is as follows:

1) At the time of an accident, Car calls Driver.
2) If Driver receives the call in 60 unit times, Driver makes an emergency call *manually* to 911 to inform the accident and moves out of the car. If not, Driver makes the call *automatically* and will be moved out of car later by Ambulance.
3) When 911 receives a call from Driver, it informs Ambulance to go to the location of the accident to handle the 'manual' or 'automatic' call cases. In case of the automatic call, additionally, it sends Driver ID, with Hospital ID, to Medical Center to get the medical information of Driver.
4) Ambulance goes to Car and handles the situation. For the manual case, Driver takes on Ambulance and gets the first aid treatment. For the automatic case, Ambulance takes Driver on, gets Driver's medical information requested by 911, makes the information-based first aid treatment, and informs Medical Center of the treatment. After the treatment, it takes Driver to Hospital for the further medical treatment.
5) Medical Center receives the ID's of Driver and Hospital, it sends Driver's medical information to Ambulance and Hospital. And later, it will get Driver's medical status from Hospital after the medical treatment.
6) Hospital receives Driver's medical information from Medical center and treats Driver based on the information at the time of Driver's arrival by Ambulance. After the treatment, it informs both 911 and Medical Center of Driver's medical status.

Fig. 5 shows System View for EMS in SAVE on ADOxx. This is the first configuration of EMS before the accident. There are four main processes, that is, Car with Driver, 911 with Ambulance, Hospital and Medical Center.

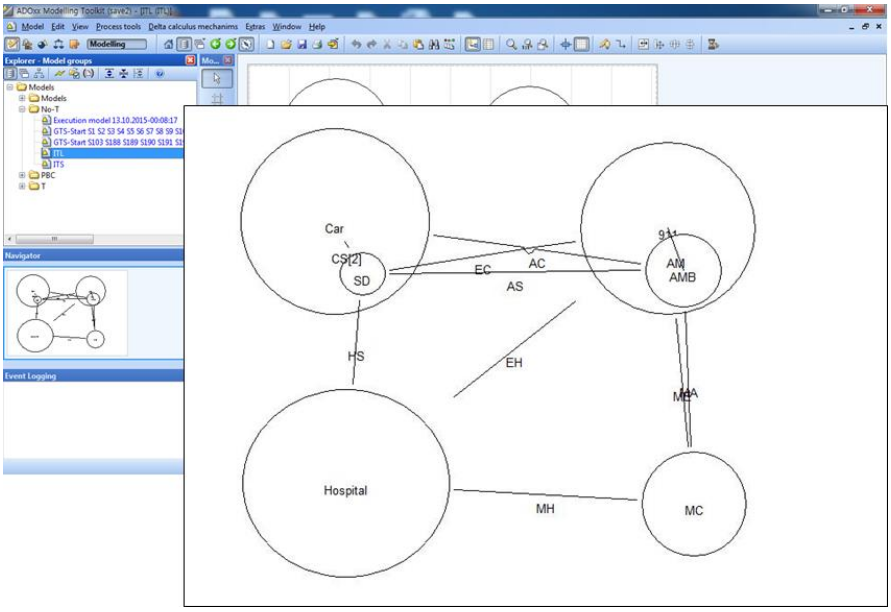


Fig. 5 System View for EMS Example.

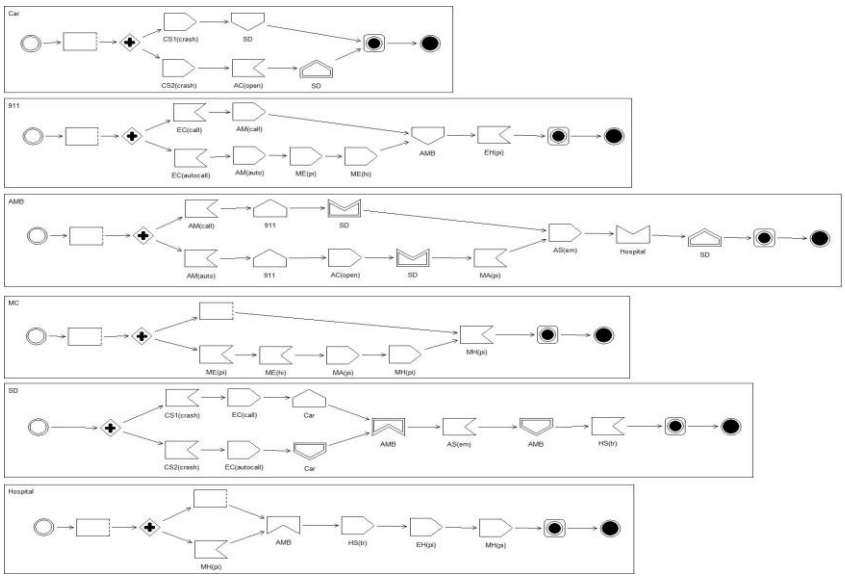


Fig. 6 Process Views for EMS Example.

Fig. 6 shows Process Views for EMS in SAVE on ADOxx. There are 6 processes. Each corresponds to each process in System View. There is 1-1 correspondence between each action of process in the EMS code and the node of the action in its Process View, except *start* and *final* nodes.

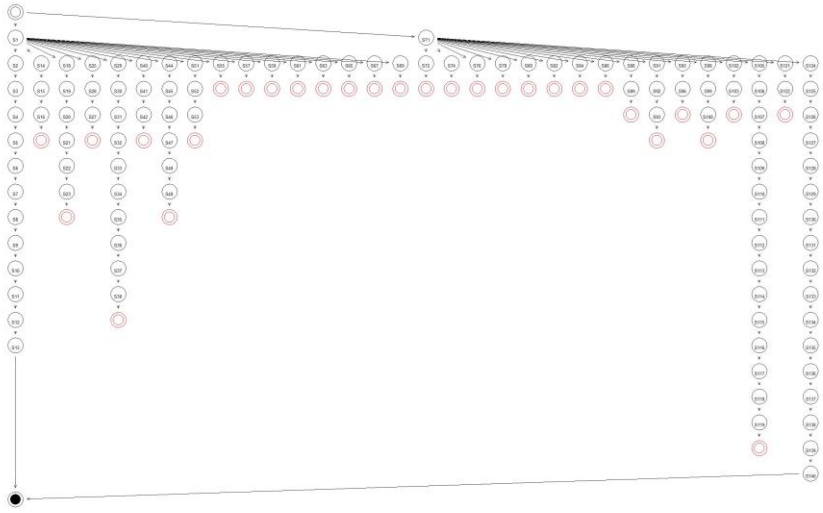


Fig. 7 Execution Tree for EMS Example.

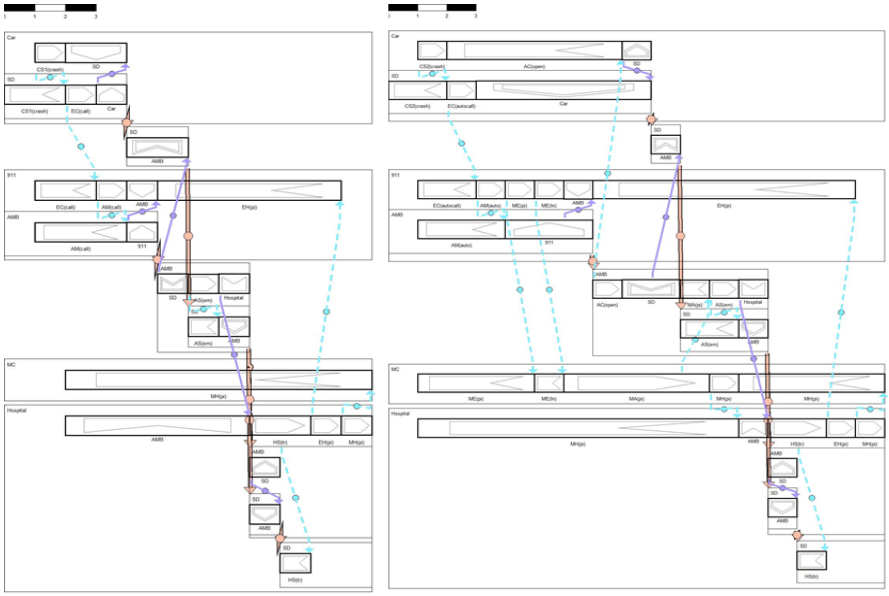


Fig. 8 Simulation Data for 2 Normal Execution Paths in EMS Example.

4.2.2 Execution Model

Fig. 7 shows the execution tree for EMS example. Each path in the tree represents one possible execution path for the example. There are total 32 possible execution paths in the tree. It is due to six choice operators from each processes in composition with two complemented alternatives, which means that there is no need to make composition of their respective choices: $2^{6-1} = 2^5 = 32$ case of the composition.

Further it shows that there are only two possible cases for normal termination: one for the manual call case on the left-most side of the figure and another for the automatic call case on the right-most. The rest of cases between the sides are deadlock.

4.2.3 Simulation Model

Fig. 8 shows the simulation data, in GTS diagrams, generated for two execution paths of the normal termination from the execution tree, based on the interactions listed in Table 8. The one on the right is for the manual call case and the one on the left is for the automatic call case. The communication and movement interactions in the table are represented as interaction edges between action blocks and their co-action blocks in the diagrams.

Both diagrams show that Ambulance moves from 911 to Car to take Driver on and transports Driver to Hospital in time. Both also show that the predefined calls are conducted among 911, Ambulance, Hospital and Medical Center in order to automate the smart EMS service.

4.3 Verification

The last step of modeling process is to verify the secure requirements of the EMS example by model-checking the simulation data in the GTS diagrams for the requirements. There can be a number of dependencies among interactions to be considered as the requirements. Some of the secure requirements for the example can be summarized as follows:

- R1: Accident must be notified to 911 before Ambulance leaves 911.
- R2: Type of Accident must be known before Ambulance arrives at the scene.
- R3: In case of the manual call, no driver's medical information is needed for first aid treatment by Ambulance.
- R4: In case of the automatic call, driver's medical information is needed for first aid treatment by Ambulance.
- R5: Driver should be treated at first by Ambulance before being arrived at Hospital.

- R6: In case of the manual call, no driver's medical information is needed for treatment at Hospital.
- R7: In case of the automatic call, driver's medical information is needed for treatment at Hospital.
- R8: After the treatment at Hospital, Hospital must inform the results to both 911 and Medical Center.
- R9: Driver must be treated first in 15 unit times by Ambulance.
- R10: Driver must be transported to Hospital in 30 unit times by Ambulance for the further medical treatment.

Table 8 List of Interactions from Simulations for Path 1 and 32 in Execution Tree.

Simulation for Path 1	Simulation for Path 32
$\tau_1 = \left((Car : CS1(\overline{crash}), Drv : CS1(crash)), 1\right)$ $\tau_2 = \left((Drv : EC(\overline{call}), 911 : EC(call)), 2\right)$ $\delta_1 = \left((Drv : out\ Car, Car : Drv\ out), 3\right)$ $\tau_3 = \left((911 : AM(\overline{call}), Amb : AM(call)), 3\right)$ $\delta_2 = \left((911 : Amb\ out, Amb : out\ 911), 4\right)$ $\delta_3 = \left((Amb : get\ Drv, Drv : 911\ get), 5\right)$ $\tau_4 = \left((Amb : AS(\overline{em}), Drv : AS(em)), 6\right)$ $\delta_4 = \left((Amb : in\ Hsp, Hsp : 911\ in), 7\right)$ $\delta_5 = \left((Amb : put\ Drv, Drv : 911\ put), 8\right)$ $\tau_5 = \left((Hsp : HS(\overline{tr}), Drv : HS(tr)), 9\right)$ $\tau_6 = \left((Hsp : EH(\overline{pi}), 911 : EH(pi)), 10\right)$ $\tau_7 = \left((Hsp : MH(\overline{pi}), Mdc : MH(pi)), 11\right)$	$\tau_1 = \left((Car : CS2(\overline{crash}), Drv : CS2(crash)), 1\right)$ $\tau_2 = \left((Drv : EC(\overline{autocall}), 911 : EC(autocall)), 2\right)$ $\tau_3 = \left((911 : AM(\overline{auto}), Amb : AM(auto)), 3\right)$ $\tau_4 = \left((911 : ME(\overline{pi, hi}), Mdc : ME(pi, hi)), 4\right)$ $\delta_1 = \left((Amb : out\ 911, 911 : Amb\ out), 6\right)$ $\tau_7 = \left((Amb : AC(\overline{open}), Car : AC(open)), 7\right)$ $\delta_2 = \left((Car : put\ Drv, Drv : Car\ put), 8\right)$ $\delta_3 = \left((Amb : get\ Drv, Drv : Amb\ get), 9\right)$ $\tau_5 = \left((Mdc : MA(\overline{pi}), Mdc : MA(pi)), 10\right)$ $\tau_6 = \left((Mdc : MH(\overline{pi}), Mdc : MH(pi)), 11\right)$ $\tau_8 = \left((Amb : AS(\overline{em}), Drv : SA(em)), 11\right)$ $\delta_4 = \left((Amb : in\ Hsp, Hsp : 911\ in), 12\right)$ $\delta_5 = \left((Amb : put\ Drv, Drv : 911\ put), 13\right)$ $\tau_9 = \left((Hsp : HS(\overline{tr}), Drv : HS(tr)), 14\right)$ $\tau_{10} = \left((Hsp : EH(\overline{pi}), 911 : EH(pi)), 15\right)$ $\tau_{11} = \left((Hsp : MH(\overline{pi}), Mdc : MH(pi)), 16\right)$

Table 9 Secure Requirements for Simulations 1 and 2.

Requirements	Simulation of Path 1	Simulation of Path 32
R1	$\tau_2 < \delta_2$	$\tau_2 < \delta_1$
R2	$\tau_2 < \delta_2$	$\tau_2 < \delta_1$
R3	$\tau_2 \rightarrow (T \rightarrow \tau_5)$	N/A
R4	N/A	$\tau_2 \rightarrow (\tau_5 \rightarrow \tau_8)$
R5	$\tau_4 < \delta_5$	$\tau_8 < \delta_5$
R6	$\tau_2 \rightarrow (T \rightarrow \tau_5)$	N/A
R7	N/A	$\tau_2 \rightarrow (\tau_5 \rightarrow \delta_5)$
R8	$\tau_5 < (\tau_6 \wedge \tau_7)$	$\tau_9 < (\tau_{10} \wedge \tau_{11})$
R9	$((T(\tau_4) - (T(\tau_1)) \leq 15$	$((T(\tau_8) - (T(\tau_1)) \leq 15$
R10	$((T(\delta_5) - (T(\tau_1)) \leq 30$	$((T(\delta_5) - (T(\tau_1)) \leq 30$

These requirements can be represented as the dependency relations among the interactions in Table 8 for both simulations, as shown in Table 9.

By checking the simulation data for each case, it can be verified if all the requirements in Table 9 are satisfied by the relations among the interactions and their timing properties in Table 8.

Finally it can be concluded that the secure requirements for the example is valid for the specification in δ-calculus.

5 Conclusion

This paper presented a formal method to specify the IoT systems in a process algebra, called δ-calculus, and verify the secure requirements by generating all the possible execution paths from the execution tree for the specification, simulating all the cases of the executions from the tree and model-checking the results of all the cases to verify the secure requirements.

In order to realize the method, a tool, called SAVE, has been developed on the ADOxx meta-modeling platform and the approach in the method has been demonstrated with an EMS example in SAVE. SAVE consists of four components: Modeler, EM Generator, Simulator and Verifier. For the example, Modeler allows to specify System View for EMS and Process Views for each processes in EMS. After specification, EM generator produces an execution tree for EMS, which consists of 32 possible execution paths: 30 abnormal cases as deadlock and 2 cases for normal termination. From the tree, the 2 normal cases were selected for the simulation and generated the GTS diagrams as a result of the simulation. Finally, Verifier verified the secure requirements for EMS by model-checking the diagrams to see whether the requirements are satisfied or not in the diagrams.

Recently, the case study for a more complex EMS example was reported in (W.Choi et at. 2015) to demonstrate capability of reducing a considerable amount of execution paths in the execution tree with complement and conjunctive choices, from 15823 paths to 6 paths. It will be a challenging work to apply the reduction method to the real industrial examples.

δ-calculus can be one of the best suitable methods to model IoT systems and SAVE can be one of the practical tools to realize the method in practice.

Acknowledgments This work was supported by Basic Science Research Programs through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2010-0023787), and the MISP(Ministry of Science, ICT and Future Planning), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2015-H8501-15-1012) supervised by the IITP(Institute for Information & communications Technology Promotion), and Space Core Technology Development Program through the NRF(National Research Foundation of Korea) funded by the Ministry of Science, ICT and Future Planning(NRF-2014M1A3A3A02034792), and Basic Science Research Program

through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(NRF-2015R1D1A3A01019282).

References

R. Milner, J. Parrow and D. Walker, A calculus of mobile processes (i-ii), *Information and Computation*, 100 (1992), pp.1-77.

L. Cardelli and A. Gordon, Mobile Ambients, In: Nivat, M. (ed.) ETAPS 1998 and FOSSACS 1998, *LNCS*, vol. 1378, pp. 140–155, Springer, Heidelberg (1998).

R. Milner, *A Calculus of Communicating Systems*, Springer-Verlag, New York, 1982.

H. Fill and D. Karagiannis, On the Conceptualisation of Modeling Methods Using the ADOxx Meta Modeling Platform, Enterprise Modeling and Information Systems Architectures 8(1), pp.4-25, 2013.

A. Whitmore , A. Agarwal and Li Da Xu, The Internet of Things – A survey of topics and trends, *Information Systems Frontiers*, vol 17, issue 2, pp 261-274, Springer, April 2015.

Internet of Things Global Standards Initiative(IoT-GSI), http://www.itu.int/en/ITU-T/gsi/iot/Pages/default.aspx

C. Heitmeyer and D. Mandrioli, Formal Methods for Real-Time Computing: An Overview, Formal Methods for Real-Time Computing, pp1-32, John Wiley & Sons, 1996.

Y. Choe, W. Choi, G. Jeon and M. Lee, A Tool for Visual Specification and Verification for Secure Process Movements, eChallenges e-2015, November 2015.

W. Choi, Y. Choe and M. Lee, A Reduction Method for Process and System Complexity with Conjunctive and Complement Choices in a Process Algebra, IEEE 39th Annual International Computer, Software & Applications Conference, 2015.

E. M. Clarke and E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, vol. 131 of *Lecture Notes in Computer Science*, pp. 52-71, Springer-Verlag, 1981.

B. Larcom, Secure Requirement, https://www.frisc.no/arrangementer/finse-winter-school-2014/, 2014.

H. Lin, Predicate μ –Calculus for Mobile Ambients, *Journal of Computer science and Technology*, vol. 20, issue 1, pp. 95-104, 2005.

Y. Choe and M. Lee, δ -Calculus: Process Algebra to Model Secure Movements of Distributed Mobile Processes in Real-Time Business Application, 23rd European Conference on Information Systems, 2015.