Towards Reliable E-Government Systems with the OTS/CafeOBJ Method*

Weiqiang KONG†, Nonmember, Kazuhiro OGATA†, Member, and Kokichi FUTATSUGI†, Nonmember

SUMMARY System implementation for e-Government initiatives should be reliable. Unreliable system implementation could, on the one hand, be insufficient to fulfill basic system requirements, and more seriously on the other hand, break the trust of citizens on governments. The objective of this paper is to advocate the use of formal methods in general, the OTS/CafeOBJ method in particular in this paper, to help develop reliable system implementation for e-Government initiatives. An experiment with the OTS/CafeOBJ method on an e-Government messaging framework proposed for providing citizens with seamless public services is described to back up our advocacy. Two previously not well-clarified problems of the framework and their potential harm realized in this experiment are reported, and possible ways of revisions to the framework are suggested as well. The revisions are proved to be sufficient for making the framework satisfy certain desired properties.

key words: e-Government messaging framework, formal methods, the OTS/CafeOBJ method, falsification, verification

1. Introduction

One major responsibility of e-Government initiatives is to provide citizens with public services electronically through information and communication technologies. It is extremely important and necessary for the system implementation for e-Government to be reliable. Unreliable system implementation could, on the one hand, be insufficient to fulfill basic system requirements, and more seriously on the other hand, break the trust of citizens on governments.

Formal methods[1] have been extensively applied to specify and analyze design of software/hardware systems (e.g. Mondex Electronic Purse[2], Analog and Mixed-Signal Circuits[3]). The benefits of using formal methods to the design of such systems stem from two aspects. Firstly, formal specifications could help eliminate ambiguities that usually reside in informal design descriptions, and secondly, formal analysis could either uncover subtle logical errors, or prove correctness of the specifications. The use of formal methods to e-Government research and practice has recently been advocated by Jim Davies, et al. in[4], and the relevance and opportunities of such usage have been explored as well.

The objective of this paper is to make a same advocacy as the one by Jim Davies, et al., but we back up this advocacy with an experiment on using a specific formal method – the OTS/CafeOBJ method[5] – to model and analyze an e-Government messaging framework[6]–[8]. The framework is proposed for providing citizens with seamless public services†, and its key idea is to support asynchronous message exchange among registered members (government agencies) through dynamically created and subscribed channels. In[8], a formal specification of the framework using the RAISE specification language (RSL[9]) has been provided but without formal analysis.

In our experiment, the framework is first modeled as an OTS (Observational Transition System)[5], a kind of state transition system that could be straightforwardly written in terms of equations. The OTS is then specified in CafeOBJ, an algebraic specification language[10], [11]. The result specification of the framework is consequently analyzed with both falsification and verification mechanisms supported by CafeOBJ system. We use the term falsification to denote finding logical errors, and verification to denote that systems enjoy some properties. With falsification, we found two previously not well-clarified problems of the framework. We analyzed potential harm that could be caused by the problems and suggested possible ways of revisions to the framework. Furthermore, with verification, we proved that the revised framework satisfies certain desired properties. Therefore, based on the results of this experiment, we make a more specific argument (compared to[4]) that e-Government initiatives would benefit from formal falsification and verification (i.e. formal analysis) activities.

Organization: Section 2 outlines the e-Government messaging framework. Section 3 briefly introduces the OTS/CafeOBJ method. Sections 4 and 5 describe our experiment: to model, specify, and analyze the messaging framework. Section 6 discusses related work, and Sect. 7 concludes the paper.

---

*Seamless public services allow citizens to specify a need and obtain a service to fulfill this need without knowing which agency or level of government should be contacted[7]. Such a service is usually delivered by collaboration of multiple government agencies.
2. An e-Government Messaging Framework

We outline the primary part of the framework and refer readers to [6]–[8] for more detailed information. Key idea of the framework is to support asynchronous message exchange among registered members (government agencies) through dynamically created and subscribed channels. The framework comprises three main components: (1) core of the framework that fulfills the basic message exchange functionalities, called G-EEG-CORE, (2) extensions that enable additional functionalities to the core, called G-EEG-EXTEND. Example extensions include logging of messages passing through a channel, and encryption/decryption of messages etc, and (3) a development methodology for building other new extensions, called G-EEG-PROGRAM. We only focus on G-EEG-CORE in our experiment as [8] did.

An illustration of G-EEG-CORE is shown in Fig. 1. We informally outline the functionalities of G-EEG-CORE as follows:\(^1\):

- `register(i)` – a government agency registers a member with \(i\) as the member’s identifier (hereafter, for simplicity, we will call it member \(i\)). Its precondition requires that the identifier \(i\) is currently not in use by other members. Each member has a pair of inbox and outbox containing lists of outgoing and incoming messages, respectively, for this member, which are initially empty.
- `unregister(i)` – the government agency (that owns member \(i\)) unregisters member \(i\). Its precondition requires that member \(i\) exists and does not own any channels.
- `create(i, x)` – member \(i\) creates a channel with \(x\) as the channel’s identifier (hereafter, for simplicity, we will call it channel \(x\)). Member \(i\) will, by default, be the owner of channel \(x\), and also be a subscriber to channel \(x\). Each channel maintains a record of identifiers who subscribed to the channel. Its precondition requires that member \(i\) exists and the identifier \(x\) is not currently in use by other channels.
- `destroy(i, x)` – member \(i\) destroys channel \(x\). Its precondition requires that channel \(x\) exists, member \(i\) is the owner of channel \(x\), and channel \(x\) does not have any other subscribers except \(i\).
- `subscribe(i, x)` – member \(i\) subscribes to channel \(x\). \(i\) will be added into channel \(x\)’s subscriber record. Its precondition requires that member \(i\) and channel \(x\) exist, and \(i\) is not already in channel \(x\)’s subscriber record.
- `unsubscribe(i, x)` – member \(i\) unsubscribes to channel \(x\). \(i\) will be removed from channel \(x\)’s subscriber record. Its precondition requires that member \(i\) and channel \(x\) exist, and \(i\) is in channel \(x\)’s subscriber record.
- `send(i, msg)` – member \(i\) sends a message \(msg\) to member \(i\)’s inbox. Its precondition requires that member \(i\) exists, and \(i\) is recorded in \(msg\) as its sender. Each message contains information of its sender, the channel (through which the message will be delivered to its receivers), and its receivers (a set of member identifiers).
- `deliver(i)` – a message in member \(i\)’s inbox is sent out to the message’s receivers’ outboxes through a channel requested by the message. The information about the channel and the receivers could be obtained from the message. Its precondition requires that member \(i\) exists, its inbox is not empty, the channel exists, member \(i\) is a subscriber of the channel, the receivers exist, and the receivers are subscribers of the channel. A sent message will be removed from the sender’s outbox.

G-EEG-CORE maintains records of existing members and channels (their identifiers). When register (unregister) a member, the member’s identifier is added into (removed from) the member record. Similar change happens to the channel record when create (destroy) a channel. Besides, as shown in Fig. 1, there is a special member `admin`, which owns a channel for each member (`admin-i` and `admin-j`, respectively). `admin` is responsible for administrating the framework, such as registering new members etc. `admin` is not considered in this paper for simplicity.

3. The OTS/CafeOBJ Method

3.1 Observational Transition Systems (OTs)

An OTS [5] is used as the formal model in the OTS/CafeOBJ method. We assume that there exists a universal state space called \(\mathcal{Y}\), and that data types used, including the equivalence relation (denoted by \(=\)) for each data type, have been defined in advance. An OTS \(S\) consists of \((O, I, T)\):

- \(O\): A finite set of observers. Each \(o \in O\) is a function \(o : \mathcal{Y} \rightarrow D\), where \(D\) is a data type and may differ from observer to observer. The equivalence relation between

\(^1\)Our description (and its CafeOBJ specification in Sect. 4) does not follow exactly the description and its RAISE specification given in [8]. However, the counterexamples and their potential harm we discovered here (to be introduced in Sect. 5) are counterexamples, and respectively, will do the same harm to G-EEG-CORE of [8], as been confirmed by the authors of [8] during our discussions.
two states $v_1, v_2 \in \mathcal{V}$ wrt $S$ (denoted by $v_1 =_S v_2$) is defined as $\forall o \in O, a(v_1) = a(v_2)$.

- $I$: The set of initial states such that $I \subseteq \mathcal{V}$.
- $\mathcal{T}$: A finite set of conditional transitions. Each $t \in \mathcal{T}$ is a function $t: \mathcal{V} \rightarrow \mathcal{V}$, provided that $t(v_1) =_S t(v_2)$ for each $[v] \in \mathcal{V}$ and each $v_1, v_2 \in [v]$. $t(v)$ is called the successor state of $v \in \mathcal{V}$ wrt $t$. The condition $c_t$ of $t$ is called the effective condition. For each $v \in \mathcal{V}$ such that $\neg c_t(v), v =_S t(v)$.

Observers and transitions may be parameterized. Generally, observers and transitions are denoted by $o_{i_1,...,i_n}$ and $t_{i_1,...,i_n}$, provided that $m, n \geq 0$ and there exist corresponding data types $D_k(k = x_1, \ldots, x_m, y_1, \ldots, y_n)$.

The formal analysis in the OTS/CafeOBJ method is essentially analysis of the reachable states wrt $S$, which are inductively defined as follows: (1) each $v_0 \in I$ is reachable, and (2) for each $t \in \mathcal{T}$, $t(v)$ is reachable if $v \in \mathcal{V}$ is reachable. All the properties that we considered in this experiment are invariant (properties). An invariant wrt $S$ is a state predicate $p: \mathcal{V} \rightarrow \text{Bool}$ such that it holds in all reachable states wrt $S$.

3.2 Specifications of OTSs in CafeOBJ

In the OTS/CafeOBJ method, an OTS is described in CafeOBJ[10],[11]. CafeOBJ is an algebraic specification language and system based on order-sorted algebras, hidden algebras, and preorder algebras (corresponding to rewriting logic of Maude). Data types can be specified in terms of order-sorted algebras. State machines such as OTSs are specified in terms of hidden algebras for verification, while they are specified in terms of preorder algebras for falsification. A CafeOBJ visible sort denotes an abstract data type, and a hidden sort denotes the state space of an abstract state machine. Both the abstract data types and the state machine are described as modules, which are the basic building blocks of CafeOBJ specifications. There are two kinds of operators in hidden sorts: action and observation operators. An action operator can change a state of an abstract state machine, and only observation operators can be used to observe the inside of an abstract state machine. Declarations of observation and action operators start with $\text{bop}$, and those of other operators with $\text{op}$. Operators are defined in equations. Declarations of equations start with $\text{eq}$, and those of conditional equations with $\text{ceq}$. The CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules.

The universal state space $\mathcal{V}$ is denoted by a hidden sort, say $H$. An observer $o_{i_1,...,i_n}$ is denoted by a CafeOBJ observation operator and declared as $\text{bop} o: H V_{i_1} \ldots V_{i_n} \rightarrow V$, where $V_{i_1}, \ldots, V_{i_n}$ and $V$ are visible sorts.

Any state in $I$ (namely any initial state) is denoted by a constant, say $\text{init}$, which is declared as $\text{op} \text{init} : \rightarrow H$. The equation expressing the initial value of $o_{i_1,...,i_n}$ is as follows:

\[ \text{eq } o(\text{init}, X_{i_1}, \ldots, X_{i_n}) = f(X_{i_1}, \ldots, X_{i_n}) . \]

$X_k$ is a CafeOBJ variable of $V_k$, where $k = x_1, \ldots, x_m$, and $f(X_{i_1}, \ldots, X_{i_n})$ is a CafeOBJ term denoting the initial value of $o_{i_1,...,i_n}$.

A transition $t_{i_1,...,i_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator and declared as $\text{bop} t: H V_{i_1} \ldots V_{i_n} \rightarrow H$, where $V_{i_1}, \ldots, V_{i_n}$ are visible sorts. $t_{i_1,...,i_n}$ may change the value returned by $o_{i_1,...,i_n}$ if it is applied in a state $v$ such that $c_{t_{i_1,...,i_n}}(v)$, which can be written generally as follows:

\[ \text{ceq } o(t(S, X_{i_1}, \ldots, X_{i_n}), X_{i_1}, \ldots, X_{i_n}) = e(t(S, X_{i_1}, \ldots, X_{i_n}, X_{i_1}, \ldots, X_{i_n}) ) 
\] 
\[ \text{if } c(t(S, X_{i_1}, \ldots, X_{i_n})) . \]

$S$ is a CafeOBJ variable of sort $H$ and all the $X$s being as parameters of $o$ and $t$ are CafeOBJ variables of corresponding visible sorts. $t(S, X_{i_1}, \ldots, X_{i_n})$ denotes the successor state of $S$ wrt $t_{i_1,...,i_n}$. $c(t(S, X_{i_1}, \ldots, X_{i_n}, X_{i_1}, \ldots, X_{i_n})$ denotes the value returned by $o_{i_1,...,i_n}$ in the successor state. $c(t(S, X_{i_1}, \ldots, X_{i_n})$ denotes the effective condition $c_{t_{i_1,...,i_n}}(v)$, which can be generally written as:

\[ \text{ceq } t(S, X_{i_1}, \ldots, X_{i_n}) = S \text{ if not } c(t(S, X_{i_1}, \ldots, X_{i_n}) . \]

3.3 Analysis of OTSs wrt Invariants

3.3.1 Falsification with Search Command

CafeOBJ system provides a search command that can be used to falsify (find logical errors of) invariants of OTSs specified in CafeOBJ specification language. General idea of using this command for falsification is trying to explore the reachable state space of an OTS from its initial state to a state in which an invariant does not hold.

To use the CafeOBJ search command, we need to (1) give an explicit state structure for an OTS $S$, and then obeying this state structure, to (2) give, for each action (representing a transition of $S$) defined in equations, additional state transition expressions. Assume that the state structure of an OTS is decided to be $< v >$, then an additional state transition expression for the action expression described in Sect. 3.2 (the second equation) can be generally written in a module $\text{TRANS}$ (which imports the module where $S$ is defined) as follows:

\[ \text{ctrans } < S > = < t(S, i_1, \ldots, i_n) > \text{ if } c(t(S, i_1, \ldots, i_n)) . \]

$\text{ctrans}$ is a keyword to declare state transition expressions. $i_1, \ldots, i_n$ are CafeOBJ constants of corresponding variables $X_{i_1}, \ldots, X_{i_n}$. These constants are determined and chosen by human analyzers with the purpose to make the state space to be explored smaller (generally the state space of an OTS is infinite). Therefore according to the number of constants selected for each variable, several such state transition expressions may exist for each action of $S$.

A CafeOBJ search command is of the form:
red < init > = (m, n) => * < S > suchThat pred(S).

The command returns true if a state S satisfying the condition pred(S) is reached, via 0 or more applications of transitions from the initial state init. Otherwise, it returns false. n is the upper bound of the depth from the initial state, and m is the upper bound of the number of such state S. For falsification purpose, we could put negation of the invariant to be analyzed in the position of pred(S).

3.3.2 Verification with Proof Scores

To verify invariants of an OTS S, we generally need to do induction on the reachable state space of S, namely to check (1) Base Case: the predicate to be proved invariant holds on initial state of S, and to check (2) Inductive Case: the predicate is preserved by the execution of each transition of S. We describe how to prove a predicate p1 is invariant to S using induction by writing proof scores [5] in CafeOBJ. The proof that p1 is invariant to S often needs other predicates. We suppose that p2, . . . , pn are such predicates. We then prove p1 ∧ . . . ∧ pn is an invariant to S. Let X1, . . . , Xn of types D1, . . . , Dn be all free variables in p1 (i = 1, . . . , n) except for S whose type is T.

We first declare operators denoting p1, . . . , pn. Their defining equations in a module INV (which imports the module where S is defined) are written as:

```
op inv : H V1i . . . Vmi -> Bool
eq inv(S, X1i, . . . , Xni) = p(S, X1i, . . . , Xni).
```

where i = 1, . . . , n. Vk (k = i1, . . . , in) is a visible sort denoting Dk, and Xk is a CafeOBJ variable whose sort is Vk. p(S, X1i, . . . , Xni) is a CafeOBJ term denoting pi. In INV, we declare a constant x1 denoting an arbitrary value of V1. We then declare operators denoting basic formulas to show in the inductive cases and their defining equations in a module ISTEP (which imports INV) as follows:

```
op instep : V1i . . . Vmi -> Bool
eq instep(X1i, . . . , Xni) =
inv(S, X1i, . . . , Xni) implies inv(S', X1i, . . . , Xni).
```

where i = 1, . . . , n. s and s' are constants of sort H, denoting an arbitrary state and a successor state of s.

Let init denotes any initial state of the OTS S. All we have to do to show that pi holds on any initial state is to write a proof as follows:

```
open INV
red inv(init, X1i, . . . , Xni).
close
```

The CafeOBJ command red reduces a term denoting a proposition to its truth value. open creates a temporary module that imports a module given as an argument, and close destroys the temporary module. Parts enclosed with open and close are basic units of proof scores, which are called proof passages.

The proof of each inductive case often requires case analysis. Let us consider the inductive case where it is shown that H1−→ s preserves pi. Suppose that the state space is split into l sub-spaces for the proof of the inductive case, and that each sub-space is characterized by a predicate casek (k = 1, . . . , l) such that (case1 ∨ . . . ∨ casek) ⇒ true. Also suppose that visible sorts V1, . . . , Vn correspond to data types D1, . . . , Dn of the parameters of H1−→ s. The proof passage for case casek is shown as follows:

```
open ISTEP
-- arbitrary objects denoted by constants e
op e1 := V1i . . . op en := Vni.
-- assumptions
Declarations of equations denoting casek.
-- successor state
eq s' = (s, e1, . . . , en).
-- check if the predicate is true
red SIHk implies instep(x1i, . . . , xni).
```

where i = 1, . . . , n. A comment starts with -- and terminates at the end of the line. SIHk is a CafeOBJ term denoting what strengthens the inductive hypothesis invi(s , x1i, . . . , Xni) and can be the concatenation of different predicates ranging from inv1( . . . ) to invn( . . . ).

4. Modeling and Specifications

We start our experiment on modeling G-EEG-CORE as an OTS, called Score, and then specifying Score in CafeOBJ. The data types used in Score are:

- Bool for truth values.
- Mname, Cname, and MsgID for names of members and channels, and identifiers of messages, respectively.
- SetMname and SetCname for sets of member names and channel names, respectively. Types Mname and Cname are declared as sub-types of them, respectively. The constructors of SetMname (SetName) are (1) empty – for an empty set, and (2) _ _ _ _ – for concatenating two SetMname (SetName) elements. Membership (overloading) operator /in is defined for both SetMname and SetCname.
- Message for messages, where mkMsg(id:MsgID, m:Mname, c:Cname, ms:SetMname) denotes a message (with identifier id) that is to be sent by sender m through channel c to receivers ms. Projection functions mid, sender, chn, and receivers are defined to extract corresponding components of a message, respectively. For example, mid(mkMsg(id, m, c, ms)) = id.
- MsgList for a list of messages. Membership operator /in is also defined.
- Member for members, where mkMem(m:Mname,
cs1:SetCname, cs2:SetCname, ib:MsgList, ob:MsgList) denotes a member with name m, who owns a set of channels cs1, subscribed to a set of channels cs2, and has inbox ib and outbox ob for messages. Projection functions mid, owns, subed, inbox, and outbox are defined to extract corresponding components of a member, respectively. invalidM denotes an uneexisting member.

- Channel for channels, where mkChn(c:Cname, m:Mname, ms:SetMname) denotes a channel with name c, which is owned by member m and is subscribed by a set of members ms. Projection functions cid, owner, and subs are defined to extract corresponding components of a channel, respectively. invalidC denotes an uneexisting channel.

- MemChn denotes a pair of member name set and channel name set, where [s1:SetMname, s2:SetCname] denotes that currently there exist a set s1 of members (names) and a set s2 of channels (names). Projection functions mSet and cSet are defined to extract corresponding components of a pair, respectively.

\[
S_{core} \equiv (O_{core},I_{core},T_{core}) \text{ such that:}
\]

\[
O_{core} \equiv \{member_{m,Mname} : T \rightarrow Member, \ 
channel_{c,Cname} : T \rightarrow Channel, \ 
record : T \rightarrow MemChn\}
\]

\[
I_{core} \equiv \{\forall \in T \} member(\forall_{m,Mname}) = invalidM \land \ 
channel(\forall_{c,Cname}) = invalidC \land \ 
record(\forall_{m,Mname}) = [empty,empty]\}
\]

\[
T_{core} \equiv \{register_{m,Mname} : T \rightarrow T, \ 
unregister_{m,Mname} : T \rightarrow T, \ 
create_{c,Mname,c,Cname} : T \rightarrow T, \ 
destroy_{c,Mname,c,Cname} : T \rightarrow T, \ 
subscribe_{m,Mname,c,Cname} : T \rightarrow T, \ 
unsubscribe_{m,Mname,c,Cname} : T \rightarrow T, \ 
send_{m,Mname,\forall:Msg} : T \rightarrow T, \ 
receive_{m,Mname} : T \rightarrow T, \ 
deliver_{m,Mname} : T \rightarrow T\}
\]

\[
S_{core} \text{ is specified as a module called MESSAGING in CafeOBJ specification language. We assume that all the data types used such as Member, Channel, and Message, etc. have been defined as modules in advance, and imported by the module MESSAGING. The signature of the module MESSAGING is as follows:}
\]

-- any initial state
op init : -> Sys
-- observation operators
bop member : Sys Mname -> Member
bop channel : Sys Cname -> Channel
bop record : Sys -> MemChn
-- action operators
bop register : Sys Mname -> Sys
bop unregister : Sys Mname -> Sys
bop create : Sys Mname Cname -> Sys
bop destroy : Sys Mname Cname -> Sys
bop subscribe : Sys Mname Cname -> Sys
bop unsubscribe : Sys Mname Cname -> Sys
bop send : Sys Mname Message -> Sys
bop receive : Sys Mname -> Sys
bop deliver : Sys Mname -> Sys

Sys is the hidden sort denoting the state space T of \(S_{core}\). Constant init denotes an arbitrary initial state of \(S_{core}\). The three observation operators correspond to the observers, and the nine action operators correspond to the transitions. In this paper, we show the CafeOBJ specifications for init and actions unregister as demonstration examples. The remaining actions could be defined in a similar way.

-- for any initial state init
eq member(init,P) = invalidM.

-- for unregister
op c-unregister : Sys Mname -> Bool
eq c-unregister(S,P) = P /in mSet(record(S)) and owns(member(S,P)) = empty.

-- unregister a member with a given name P
op c-unregister : Sys Mname -> Bool
eq c-unregister(S,P) = P /in mSet(record(S)) and owns(member(S,P)) = empty.

-- unregister a member with a given name P
op c-unregister : Sys Mname -> Bool
eq c-unregister(S,P) = P /in mSet(record(S)) and owns(member(S,P)) = empty.

The variables P and X are of sorts Mname and Cname, denoting an arbitrary member name and an arbitrary channel name, respectively. The meaning of the equations is that in the initial state, there exist no members and channels, and thus the record is a pair of empty sets of members and channels.

The effective condition of unregister demands that in state S, the to-be-unregistered member P should exist and it has no owned channels. If the effective condition holds, execution of unregister will (1) change the observed value of member to invalidM (namely a member with the name P does not exist); (2) the return value of channel remain unchanged; and (3) remove the name P from the member name set of the record (by the function del). If the effective condition does not hold, nothing changes. Note that the variable P denotes an arbitrary member to be unregistered, and variable Q denotes an arbitrary member that the observer member is “observing” on, and thus we need to compare if they are the same member.

5. Analysis

We give the informal descriptions of four basic desired properties of \(S_{core}\) that we have analyzed in our experiment. All of these properties are safety properties, more precisely in-
variant properties\(^\dagger\).

**Property 1:** All subscribers to a channel are members.

**Property 2:** The owner of a channel is a member.

**Property 3:** The owner of a channel is subscribed to the channel.

**Property 4:** Messages are delivered only to receivers.

The four properties are specified in a module `INV` (which imports the module `MESSAGING` defining `S_{core}`) as follows:

```plaintext
op prop1 : Sys Mname Cname -> Bool
eq prop1(S,P,C) = C /\in cSet(record(S)) and P /\in subs(channel(S,C)) 
  implies P /\in mSet(record(S)) .

op prop2 : Sys Cname -> Bool
eq prop2(S,C) = C /\in cSet(record(S)) 
  implies owner(channel(S,C)) /\in mSet(record(S)) .

op prop3 : Sys Cname -> Bool
eq prop3(S,C) = C /\in cSet(record(S)) 
  implies owner(channel(S,C)) /\in subs(channel(S,C)) .

op prop4 : Sys Mname Message -> Bool
eq prop4(S,P,M) = P /\in mSet(record(S)) and M /\in outbox(member(S,P)) 
  implies P /\in receivers(M) .
```

5.1 Falsification of `S_{core}`

Following the method described in Sect. 3.3.1, we first prepare additional state transition expressions in a module `TRANS` (which imports the module `INV` and thus `MESSAGING` is consequentially imported) for each of the nine actions.

We choose two constants \(i\) and \(j\) denoting two arbitrary member names, and two constants \(x\) and \(y\) denoting two arbitrary channel names. The state transition expressions for `unregister` are shown as demonstration examples as follows, in which we give names to the transitions by adding labels such as “[unregister-\(i\)]”:

```plaintext
ctrans [unregister-\(i\)] : 
  < S > ==\< unregister(S,i) > if c-unregister(S,i) .

ctrans [unregister-\(j\)] : 
  < S > ==\< unregister(S,j) > if c-unregister(S,j) .
```

We then try to falsify the four properties by inputting the following search commands into the CafeOBJ system.

```plaintext
red < init > ==\((1,*))==\(\times S; Sys\) 
  suchThat (not prop1-\(i\)\(j\)y(S)) .
red < init > ==\((1,*))==\(\times S; Sys\) 
  suchThat (not prop2-\(i\)xy(S)) .
red < init > ==\((1,*))==\(\times S; Sys\) 
  suchThat (not prop3-\(i\)xy(S)) .
red < init > ==\((1,*))==\(\times S; Sys\) 
  suchThat (not prop4-\(i\)\(j\)x(S)) .
```

In the above commands, terms like `prop1-\(i\)\(j\)y(S)` are instantiated versions of corresponding properties. For example, `prop1-\(i\)\(j\)y(S)` = `(prop1(S,\(i\),\(x\)) and prop1(S,\(i\),\(y\)) and prop1(S,\(j\),\(x\)) and prop1(S,\(j\),\(y\)))`. For each property, we expect one result and allow to apply infinite number of transitions (denoted by the upper bounds (1,*)).

5.1.1 For Property `prop1`

CafeOBJ system returns true for the first command, which means that a state satisfying the negation of `prop1`, namely a counterexample to `prop1`, has been found. This means that property `prop1` does not hold for `S_{core}`. Following the report of CafeOBJ, such a state (named as `s5` here) can be reached from the initial state `init` as follows:

```plaintext
init ==[register-\(i\)]==\(\times s1 ==[register-\(j\)]==\(\times s2 
==[create-\(ix\)]==\(\times s3 ==[subscribe-\(jx\)]==\(\times s4 
==[unregister-\(j\)]==\(\times s5)
```

The sequence of behaviors can be read as: a successor state is reached from its previous state by applying the state transition written in between them. For example, `s3` is reached from `s2` by member \(i\) creating a channel \(x\).

After checking the specification of action `unregister`, we can understand the cause of this unexpected sequence of behaviors: when unregister a member which has subscribed to a channel, the member’s name is not removed from the channel’s subscriber record (since the return value of observer `channel` remains unchanged).

We now analyze what potential harm could possibly be caused by this problem. We know that, essentially, a member can send (receive) messages to (from) a channel only if the member is a subscriber to the channel (see the precondition of action `deliver`). So, if the name of an unregistered member still exists in a channel that this member previously subscribed to, whether this may cause some unexpected behaviors of G-EEG-CORE? Let us see Fig. 2.

The scenario illustrated in Fig. 2 can be explained as: (1) a government agency `A` registers G-EEG-CORE using a name \(i\), and another agency `B` registers G-EEG-CORE using a name \(j\), (2) member \(i\) creates a channel \(x\), and member \(j\) subscribes to channel \(x\), (3) member \(i\) sends a message `msg` to member \(j\) through channel \(x\), where \(j\) as one of the receivers and \(x\) as the intended channel are recorded in `msg`, (4) agency `B` unregisters member \(j\), (5) a third government agency `C` registers G-EEG-CORE using the name \(j\), (6) member \(j\) (representing agency `C` rather than agency `B`) receives

\(^\dagger\)Although the OTS/CafeOBJ method could also be used for verifying liveness properties [12], we focus on invariant ones in this paper since they are more basic and usually used as lemmas for proving liveness properties.
msg from channel x. An assumption for this scenario is that a government agency can register G-EEG-CORE using any name as long as the name is currently not in use by other members.

The above scenario could actually be reported by the search command of CafeOBJ when we try to check another property. The property, named as Property 5, says that: if a member m received a message from a channel c and member m never unsubscribed channel c, then member m should have subscribed and still be subscribing to channel c. To express and analyze this property, we need to expand the CafeOBJ specifications of the messaging framework by adding an action-history, which is a list of names of actions that have been executed so far from init. In addition, one more observer called history is added to return the current contents of the action-history in a given state. We omit the specification of the search command for Property 5 and the sequence of behaviors of G-EEG-CORE, i.e., the counterexample, reported by CafeOBJ (same as the above six steps shown in Fig. 2).

5.1.2 For Property prop2

CafeOBJ returns nothing for the second command and keeps running. The reason for this is that even if we chose constants i, j, x and y to try to make the reachable state space of $S_{core}$ smaller, the restricted state space is still too large (actually still infinite) for CafeOBJ system to explore. However, if we set the upper bound of the depth, say 20, CafeOBJ system will return false, which means that within this upper bound, no states that satisfy the defined condition have been found.

One thing that we should keep in mind is that: even if CafeOBJ system returns false, since CafeOBJ system has not explored the entire reachable state space of $S_{core}$, we could not say that prop2 holds for $S_{core}$.

5.1.3 For Property prop3

CafeOBJ system returns true for the third search command, which means that a state satisfying the negation of prop3, namely a counterexample to prop3, has been found. This means that property prop3 does not hold for $S_{core}$. Following the report of CafeOBJ, such a state (named as s3 here) can be reached from the initial state init as follows:

init ==[register-i]==> s1
==[create-ix]==> s2 ==[unsubscribe-ix]==> s3

We now analyze what potential harm could possibly be caused by this problem. Let us see Fig. 3.

Figure 3 illustrates a scenario in which the extensions of G-EEG-CORE such as logging the messages passing through a channel, and encryption/decryption of messages, are employed. As illustrated, when the extensions are to be employed, the messages sent by member i through channel x (whose owner is member k) to member j are not directly sent to member j, but firstly sent to member k, and member k will enable those extensions. Therefore, if member k unsubscribes channel x, then no extensions mentioned above could ever be enabled, which is definitely not as expected.

5.1.4 For Property prop4

Similar to Prop2, CafeOBJ could not find any counterexample for prop4. Note that to use CafeOBJ search command for prop4, we should make the structure of a message clear. For example, mkMsg(mid,i,x,(i,j)), rather than a constant m of type Message, is used to denote a message, where (i,j) denotes the receivers of the message. We declare an additional constant mid of type MsgID to denote the identifier of the message.

Recall that we use prop4 to state “Messages are delivered only to receivers”. Here by receivers, we mean members, rather than the government agencies that register G-EEG-CORE as those members. Therefore there is no contradiction between the search result for prop4 with the unexpected scenario that we analyzed for prop1 in Sect. 5.1.1. If we want, however, to use this receivers to denote government agencies, then we have to model government agencies as well, rather than only model members. In other words, taking the action register(gc, i) as an example, we may have to change it as register(gc, i) to denote “government agency gc registers G-EEG-CORE as a member i.”

5.2 Verification of Revised $S_{core}$

We give our suggested revisions to the framework G-EEG-CORE. To avoid the potential harm mentioned in Sect. 5.1.1, actually there are several possible ways, for example, (1) register the framework using fresh names, and disallow using a previously used name, (2) use extensions such as authentication, and (3) when unregister a member, remove the member’s name from the channel that this member has subscribed to. We choose the third one. In the CafeOBJ specifications for action unregister, the return value of observer channel is changed as follows:

ceq channel(unregister(S,P),Y) = (if (P /in subs(channel(S,Y))) then mkChn(Y,owner(channel(S,Y))), del(P,subs(channel(S,Y)))) else channel(S,Y) fi) if c-unregister(S,P) .
The revised specification says that: if the name of the to-be-unregistered member P exists in a channel Y’s subscriber record, then the name is removed from that record, and other components of the channel remain unchanged.

To avoid the potential harm mentioned in Sect. 5.1.3, we simply disallow the owner of a channel to unsubscribe the channel by changing the effective condition of action unsubscribe as follows:

\[
\text{eq \ c\text{-}unsubscribe}(S, P, X) = P \in \text{mSet}(\text{record}(S)) \Rightarrow X \in c\text{Set}(\text{record}(S)) \Rightarrow \text{not}(\text{member}(\text{record}(S), P)) \\
\]

Following the introduction in Sect. 3.3.2, we verify that properties \text{prop1}, \text{prop2}, \text{prop3}, and \text{prop4} are invariants of the revised specifications. Parts of the proof score of \text{prop2} are described in this paper as a demonstration example.

The basic formula to prove in each inductive case is declared in module ISTEP (which imports the module INV) as follows, where \(s, s', q\) are constants of sort \(\text{Sys}\) (denoting an arbitrary state and its successor state) declared in INV:

\[
\text{op \ istep2 : Cname } \rightarrow \text{Bool} \equiv \text{prop2}(s, C) \Rightarrow \text{prop2}(s', C).
\]

Let us consider the inductive case that action \text{unregister} preserves \text{prop2}. The proof passage of this inductive case is initially written as follows:

```
open ISTEP
-- arbitrary objects
op q : \rightarrow \text{Mname}.
-- successor state
eq s' = \text{unregister}(s, q).
-- check if \text{prop2} is preserved.
red istep2(c).
```

We input this proof passage into CafeOBJ system. CafeOBJ rewrites the term \(\text{istep2}(c)\) and returns neither true nor false, but a boolean term \(((c /\text{in cSet}(\text{record}(s, q)))) \text{ xor } \ldots ) : \text{Bool}\). What we need to do now is to conduct case-splitting. From the above term, especially the first operand of “and”, we could understand that CafeOBJ does not know how to reduce the term \(\text{record}(\text{unregister}(s, q))\) since it is defined using a conditional equation (see CafeOBJ specification for \text{unregister} in Sect. 4). Therefore, we split this case into two sub-cases, where the effective condition \(\text{c\text{-}unregister}(s, q)\) is true and false, respectively. Such kind of case-splitting is recursively conducted until CafeOBJ returns either true or false for all proof passages. Finally the inductive case \text{unregister} is split into 9 sub-cases based on the following 6 predicates:

\[
\text{bp1} \equiv q /\text{in mSet}(\text{record}(s)) \\
\text{bp2} \equiv \text{member}(s, q) = \text{empty} \\
\text{bp3} \equiv q /\text{in subs}(\text{channel}(s, c)) \\
\text{bp4} \equiv c /\text{in cSet}(\text{record}(s)) \\
\text{bp5} \equiv \text{owner}(\text{channel}(s, c)) /\text{in mSet}(\text{record}(s)) \\
\text{bp6} \equiv \text{owner}(\text{channel}(s, c)) = q
\]

where the constant \(q\) denotes an arbitrary member name, and \(c\) denotes an arbitrary channel name. The conjunction of \text{bp1} and \text{bp2} is the effective condition of action \text{unregister}. The 9 sub-cases are as follows:

\[
\begin{align*}
\text{sub-case1} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \\
\text{sub-case2} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \land \text{bp6} \\
\text{sub-case3} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \land \text{bp6} \\
\text{sub-case4} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \\
\text{sub-case5} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \land \text{bp6} \\
\text{sub-case6} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \\
\text{sub-case7} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \land \text{bp6} \\
\text{sub-case8} & \equiv \text{bp1} \land \text{bp2} \land \text{bp3} \land \text{bp4} \land \text{bp5} \land \text{bp6} \\
\text{sub-case9} & \equiv \neg(\text{bp1} \land \text{bp2})
\end{align*}
\]

Among the above sub-cases, CafeOBJ returns false for sub-cases 2 and 7, and true for the remaining ones. We need to do nothing if true is returned. There are two possibilities if false is returned (1) the predicate \text{prop2} to be proved is NOT an invariant, or (2) the arbitrary state characterized by the equations of the sub-case corresponding to the proof passage is not reachable wrt OTS \(S_{\text{core}}\), and this sub-case can be discharged by some lemmas. By observing the equations in sub-case 2 and 7, we found that \text{bp6} contradicts to \text{bp2}, namely that if the owner of a channel is a member \(q\) (bp6) then the set of channels owned by \(q\) should NOT be empty (contradiction to \text{bp2}). We thus come up with a lemma \text{lemma1} and sub-cases 2 and 7 can be proved (discharged) successfully by using it. We show the proof passage for sub-case 2 in which \text{lemma1} is used:

```
open ISTEP
-- arbitrary objects
op q : \rightarrow \text{Mname}.
-- assumptions characterizing a sub-case
eq q /\in \text{mSet}(\text{record}(s)) = \text{true}.
\text{eq \ owns}(\text{member}(s, q)) = \text{empty}.
\text{eq q /\in \text{subs}(\text{channel}(s, c)) = \text{true}.
\text{eq c /\in \text{cSet}(\text{record}(s)) = \text{true}.
\text{eq \ owner}(\text{channel}(s, c)) /\in \text{mSet}(\text{record}(s)) = \text{true}.
\text{eq \ owner}(\text{channel}(s, c)) = q.
-- successor state
eq s' = \text{unregister}(s, q).
-- check if \text{prop2} is true.
red \text{lemma1}(s, q, c) implies istep2(c).
```

Constant \(c\) has been declared in the module \text{INV} that is imported by \text{STEP}. \text{lemma1} is declared in module \text{INV} as:

```
\text{op \ lemma1 : \text{Sys} Mname Cname } \rightarrow \text{Bool}
\text{eq \ lemma1}(s, P, C) = \text{owner}(\text{channel}(s, C)) = P \Rightarrow \text{in owns}(\text{member}(s, P)).
```

The above mentioned ideas of case-splitting and finding lemmas are very general. More well summarized tips on these two issues can be found in paper [13]. There is a tool
called Buffet [14] that can conduct case-splitting automatically to some extent, and furthermore, another tool called Crème [15] can conduct both case-splitting and finding lemmas fully automatically, for invariant verification using the proof scores method. (e.g., Crème has successfully verified the NSLPK protocol [16] fully automatically.) However, neither of these two tools are mature enough to be applicable to complex applications. We therefore did not use them in our experiment and all the proof scores are written manually.

Note that we need also to prove that lemma1 is an invariant of the revised $S_{core}$. In addition, we have also proved and used several other lemmas to verify other inductive cases of prop2 and the other three properties. In this paper, we omit the descriptions of these lemmas and their proofs.

6. Related Work

Elsa Estevez, et al. informally addressed in [6] the reliability requirements of the messaging framework on various levels of design, development and application. The same authors formally specified the core part (G−EEG−CORE) of the messaging framework in [8] using the RAISE specification language. Formal analysis of the RAISE specification, considered as a future work, was not given in [8]. Our work of modeling and specification of G−EEG−CORE is inspired by the RAISE specification, but additionally, we conducted formal analysis (both falsification and verification) of the defined specification. Formal specification itself could contribute to develop reliable e-Government system implementation in the sense of providing rigorous and precise descriptions of the systems’ behaviors, and thus avoiding ambiguities often existing in informal descriptions. However, as indicated by the results (especially by the two realized problems of G−EEG−CORE) of our experiment, formal analysis could contribute more to achieve this reliability to e-Government systems.

Jim Davies, et al. advocate in [4] the use of formal methods to e-Government research and practice by exploring the relevance and opportunities for such usage. Such an exploration is conducted in [4] by mapping various general application scenarios of formal techniques to the identified challenges peculiar to e-Government development. Some studies involving formal elements that exemplify this map are also mentioned in [4]. For example, [17] gives a formal model of threats, vulnerabilities and controls for enhancing government risk assessment, [18] explains how to define semantic frameworks that could help provide rigorous descriptions of e-Government system implementation. Our work could be considered as another exemplification study, which involves various formal elements of this map ranging from modeling, through specifications, to analysis.

Kazuhiro Ogata, et al. have applied the OTS/CafeOBJ method to modeling and analysis of various applications, including non-trivial ones such as SET payment protocol [19], Mondex Electronic Purse [20], etc. Regarding e-Government systems, Xiaoyi Chen, et al. have recently used this method to try to clarify and provide formal definitions to transparency properties in public administration domain, and to analyze (verify with proof scores) whether a given e-Government system design modeled as an OTS satisfies these properties [21]. Since the emphasis of [21] is to give formal definitions of transparency, formal analysis is not described in detail. Besides, falsification with the search command is not applied in this work.

7. Conclusion and Discussion

We described an experiment on using a specific formal method – the OTS/CafeOBJ method – to formally model and analyze a messaging framework proposed for providing citizens with seamless public services. In this experiment, two previously not well-clarified problems of the framework have been realized using the search command, and the potential harm that could possibly be caused by them is analyzed. We also suggested possible revisions to the framework, and proved by writing and executing proof scores that the revised specifications of the framework satisfies some desired properties. Based on this experiment, we advocated the use of formal methods, in particular formal analysis, to e-Government initiatives to help develop reliable e-Government systems.

Recently, we have known that in a refined/updated specification of the RSL one presented in [8], Elsa Estevez, et al. have corrected the two problems found in our experiment. In the refined version, a globally unique/fresh name is assigned by admin to a newly registered member, and the owner of a channel is disallowed to unsubscribe the channel.

In this paper, falsification with the search command is fully automatic, and is used for finding logical errors by exploring part of the system’s behaviors (therefore even if no errors are found, the correctness is not proved); verification by writing and executing proof scores needs much more human verifiers’ effort (for preparing the proof scores), and is used for showing that the system enjoys some desired properties by exploring its entire behaviors. Therefore there is a balance between the uses of the two techniques. Considering the fact that e-Government systems demand extremely high reliability, we believe that the effort paid for proving correctness is worthwhile, and thus verification is treated as the primary formal analysis technique, while falsification as an assistant technique, in the OTS/CafeOBJ method. Falsification can be used to discover errors (within bounded depth) of both the properties to be analyzed, and the intermediate lemmas prepared by users to verify these properties. Supporting both falsification and verification for one (almost) same system specification, is one of the primary benefits of the OTS/CafeOBJ method. In [22], techniques about how to combine falsification and verification to facilitate each other within the OTS/CafeOBJ method are proposed and discussed in detail.

One of our future work is to use the OTS/CafeOBJ method to model extensions to G−EEG−CORE, and analyze...
desired properties of the extensions. Since certain cryptographic mechanisms will be used in the extensions, it will then become necessary for us to take into account existence of malicious intruders to e-Government systems, as has usually been considered for secure protocols, which are mission-critical systems. In the experiment of this paper, we did not consider intruders since we focused on G-EEG-CORE as did in [8] and before we could model potential behaviors of the intruders, we would have to assume the existence of a certain cryptographic mechanism. One (among others) interesting property in the extensions is related to anonymity. Intuitively this means to prevent the disclosure of information like who did what. Such kind of properties may be of great importance for e-Government systems, in which information highly related to privacy may be frequently involved. Last, another more general future work for us is to apply the OTS/CafeOBJ method to model and analyze the design of other real-life e-Government systems.

Acknowledgments

This research is conducted as a program for the “21st Century COE Program” in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology. We would like to thank Elsa Estevez and Tomasz Janowski for providing us with fundamental materials of the messaging framework, and giving useful comments to this work. We would also like to express our appreciation to the anonymous reviewers and the associated editor for giving us insightful and valuable comments for revising the paper.

References


Weiqiang Kong is a researcher at Fukuoka Industry, Science & Technology Foundation (Fukuoka IST). He received his Ph.D. in information science from Japan Advanced Institute of Science and Technology (JAIST) in 2006. He was a post-doc researcher at JAIST from 2006 to 2008. His research interests include software engineering and formal verification.
Kazuhiro Ogata is an associate professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his Ph.D. in engineering from Graduate School of Science and Technology, Keio University in 1995. He was a research associate at JAIST from 1995 to 2001, a researcher at SRA Key Technology Laboratory, Inc. from 2001 to 2002, a research expert at NEC Software Hokuriku, Ltd. from 2002 to 2006, and a research associate professor at JAIST from 2006 to 2009. Among his research interests are software engineering, formal methods and formal verification.

Kokichi Futatsugi is a professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). Before getting a full professorship at JAIST in 1993, he was working for ETL (Electrotechnical Lab.) of Japanese Government and was assigned to be Chief Senior Researcher of ETL in 1992. His research interests include formal methods, system verification, software requirements/specifications, language design, concurrent and cooperative computing. His primary research goal is to design and develop new languages which can open up new application areas, and/or improve the current software technology. His current approach for this goal is CafeOBJ formal specification language. CafeOBJ is multi-paradigm formal specification language which is a modern successor of the most noted algebraic specification language OBJ.