A Method of Requirement Inconsistency Analysis

Abstract

A new requirement inconsistency analysis method is presented to solve the problem of requirement inconsistent description in the process of requirement analysis. In the method, the requirement model is of two sets: state set and event set, and is described as reduced state machine to maintain global context information. A unified requirement model is reached by using three-valued operator to remove the inconsistency in the inconsistent models. In addition, the implementation steps of the method are given and a tool named SRIA (Simple Requirement Inconsistency Analyzer) based on the method is introduced. Finally, an example is used to illustrate the validity of this method.

1. Introduction

With the globalization of software development, software requirement elicitation is also becoming distributed. The distributed requirement elicitation achieves larger elicit basis, and leads to more accurate requirement analysis result. However, distributing elicitation also gives rise to more requirement inconsistencies, which makes researches of requirement inconsistency more significant.

There are mainly two methods in requirement inconsistency research nowadays, logic-based method and goal-based method. The logic based method uses classical logic or quasi-classical logic to describe requirements, in which the requirement inconsistency is discovered with theorem proofing. The goal based method treats requirements as goals, by which the requirement inconsistency is resolved through the goal decomposition.

In the logic-based method, a model is divided into individual propositions which are analyzed respectively. One drawback of this method is that the propositions are separate from their original context, which tends to result in the loss of their context information. This makes the analysis results of the propositions probably are not appropriate on the global basis. The goal-based method solves this problem by introducing the concept of goal. However, there is no a general method guiding the process of inconsistency analysis. As a result, the heuristic rules have to be revised to fit different cases of requirement inconsistency analysis.

In this paper, we propose a multi-value logic based state machine. The state machine provides the ability to specify requirements globally, whereas the multi-value logic is the basis of automatic inconsistency analysis and makes it possible to realize different policies for inconsistence analysis.

The new method will be introduced as follows: in section 2, state machine based requirement description is presented. Section 3 describes the new requirement inconsistency analysis method, section 4 introduces an implemented tool based on the method and the implementation process of our method, section 5 shows the process of our method with an example, and the paper is conclude in section 6.

2. Requirement description

Requirement descriptions are the intended reaction of the target system in a specified environment. The reaction can be described as state transitions. So we can describe requirements as states, among which systems transit and events which cause the state to transit.

2.1. Requirement model of state machine

Requirement description is of three parts: start state, state set and event set. State transitions are caused by events. We reduce classical state machine because some software, like OS, has not a terminal state and explicit input, and output is not indispensable. We define requirement model as a triple tuple $M$:

\[ M = (s_0, S, E) \]

where $s_0$ is start state, $S$ is state set and $E$ is event set.

\[ S = \{s_0, s_1, s_2, \ldots\} \]

\[ E = \{e_0, e_1, e_2, \ldots\} \]

$s_i$ and $e_i$ above represent a state and event respectively.

**def 2.1:**\[ V = \{v_0, v_1, v_2, \ldots v_i\} \]

The description of a state consists of a set of variable $v$.

**def 2.2** \[ stat(v, s) \]

The value of variable $v$ in state $s$.

**def 2.3** \[ source(e) \]

The starting state of event $e$.
2.2. State equivalence

Requirements describe the process of target system transiting among different states to accomplish specified operation. The first step of requirement analysis is to determine the equivalence relationship of inconsistency requirement models. There are three types of equivalence relationship: one to one equivalence, one to many equivalence and many to many equivalence.

2.2.1. One to one equivalence

\[ s_{A_1} \approx s_{B_1} \]

\[ \iff \neg \exists v: \text{stat}(v, s_{A_1}) \neq \text{stat}(v, s_{B_1}) \]

In two inconsistent requirement models \( M_A \) and \( M_B \), we call two state sets \( \{ s_{A_1}, s_{A_2}, \ldots, s_{A_t} \} \) and \( \{ s_{B_1}, s_{B_2}, \ldots, s_{B_u} \} \) are equivalent if there does not exist any \( v \) that has different values in \( s_{A_1} \) and \( s_{B_1} \). For example:

\[ V_A = \{ \text{attribute1}, \text{attribute2} \} \]
\[ \text{stat}(\text{attribute1}, s_1) = \text{stat}(\text{attribute1}, s_2) \]
\[ \text{stat}(\text{attribute2}, s_1) = \text{stat}(\text{attribute2}, s_2) \]
\[ \text{then } s_1 \approx s_2 \]

2.2.2. One to many equivalence

\[ s_{A_1} \approx [s_{B_1}, s_{B_2}, \ldots] \]

\[ \iff s_{A_1} \approx s_{B_1} \land \ldots \land s_{A_1} \approx s_{B_k} \land s_{B_k} \neq \ldots \neq s_{B_u} \]

In two inconsistent requirement model \( M_A \) and \( M_B \), we call \( s_{A_1} \) is equivalent to multiple different states, \( s_{B_1}, s_{B_2}, \ldots \) at the same time. For example:

\[ V_A = \{ \text{attribute1} \} \]
\[ V_B = \{ \text{attribute1}, \text{attribute2} \} \]
\[ \text{stat}(\text{attribute1}, s_{A_1}) = \text{stat}(\text{attribute1}, s_{B_1}) \]
\[ \text{stat}(\text{attribute2}, s_{A_1}) = \text{stat}(\text{attribute2}, s_{B_2}) \]
\[ s_{A_1} \approx [s_{B_1}, s_{B_2}] \]

2.2.3. Many to many equivalence

\[ \{ s_{A_1}, s_{A_2}, s_{A_3} \} \approx [s_{B_1}, s_{B_2}] \]

\[ \iff \exists v: \text{stat}(v, s_{A_1}) = \text{stat}(v, s_{A_2}) \land s_{A_3} \approx [s_{B_1}, s_{B_2}] \]

In two inconsistent requirement model \( M_A \) and \( M_B \), we call two state sets \( \{ s_{A_1}, s_{A_2}, s_{A_3} \} \) and \( \{ s_{B_1}, s_{B_2} \} \) are equivalent if \( s_{A_1}, s_{A_2} \) and \( s_{A_3} \) have one to many equivalence relationship with \( \{ s_{B_1}, s_{B_2} \} \) respectively. For example:

\[ V_A = \{ \text{attribute1}, \text{attribute2} \} \]
\[ V_B = \{ \text{attribute1}, \text{attribute3} \} \]
\[ \text{stat}(\text{attribute1}, s_{A_1}) = \text{stat}(\text{attribute1}, s_{B_1}) \]
\[ \text{stat}(\text{attribute2}, s_{A_1}) = \text{stat}(\text{attribute1}, s_{B_2}) \]
\[ s_{A_1} \approx [s_{B_1}, s_{B_2}] \]
\[ s_{A_2} \approx [s_{B_1}, s_{B_2}] \]
\[ s_{A_3} \approx [s_{B_1}, s_{B_2}] \]

2.3. Multi-valued logic

Inconsistencies occur in most of the practical requirements analysis. As the classical logic can’t handle inconsistent information, multi-valued logic has been proposed and developed rapidly. Ternary logic \([3]\) is one of the most researched logic theories.

In ternary logic, three truth values, true \( t \), false \( f \) and unknown \( u \), are used to specify the judgment of a proposition. As the meaning of the words, \( t \) meaning the positive judgment, \( f \) means the negative judgment and \( u \) means the undetermined judgment. Ternary logic has unary, binary and trinary operator, as listed in table 1.

<table>
<thead>
<tr>
<th>( p )</th>
<th>( q )</th>
<th>( \neg p )</th>
<th>( p \land q )</th>
<th>( p \lor q )</th>
<th>( p \rightarrow q )</th>
<th>( p \leftarrow q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u )</td>
<td>( u )</td>
<td>( u )</td>
<td>( u )</td>
<td>( t )</td>
<td>( t )</td>
<td>( t )</td>
</tr>
<tr>
<td>( u )</td>
<td>( t )</td>
<td>( u )</td>
<td>( u )</td>
<td>( t )</td>
<td>( t )</td>
<td>( u )</td>
</tr>
<tr>
<td>( f )</td>
<td>( u )</td>
<td>( f )</td>
<td>( f )</td>
<td>( u )</td>
<td>( u )</td>
<td>( u )</td>
</tr>
<tr>
<td>( t )</td>
<td>( f )</td>
<td>( t )</td>
<td>( t )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( f )</td>
<td>( f )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
<td>( f )</td>
<td>( t )</td>
</tr>
<tr>
<td>( f )</td>
<td>( t )</td>
<td>( f )</td>
<td>( f )</td>
<td>( t )</td>
<td>( u )</td>
<td>( u )</td>
</tr>
</tbody>
</table>

In this paper, ternary logic is used to specify the probability of an event. \( p(e) = t \) is used to describe a positive event \( e \) which should happen. \( p(e) = f \) is used to describe a negative event \( e \) which should not happen. \( p(e) = u \) is used to describe an undetermined event \( e \) which might or might not happen. Unknown \( u \) is the
default value in describing event, which means when we do not explicitly specify a truth value to an event, the truth value of this event is $u$. In this way, we only need to specify events which have $t$ or $f$ value from $n \times n$ events of a requirement model which has $n$ states.

Figure 1
Diagram of event representation

Figure 1 shows the truth value of an event. When $p(e) = t$, a directed line is drawn from the starting state of $e$ to the ending state of $e$. When $p(e) = f$, a directed line with a vertical short line is drawn from the starting state of $e$ to the ending state of $e$. When $p(e) = u$, no explicit line is drawn. In the example of figure 1, there are altogether $3 \times 3 = 9$ events, which are as follows:

- $p(e_{s00}) = u$
- $p(e_{s01}) = t$
- $p(e_{s02}) = t$
- $p(e_{s10}) = u$
- $p(e_{s11}) = u$
- $p(e_{s12}) = f$
- $p(e_{s20}) = u$
- $p(e_{s21}) = u$
- $p(e_{s22}) = u$

We give an example to describe our requirement model as a whole. A log-in model has two states: $s_0$ as logged-in state and $s_1$ as not-logged-in state. There is one variable ifLoggedIn to describes state. In state $s_0$, the value of ifLoggedIn is true. In state $s_1$, the value of ifLoggedIn is false. There are $2 \times 2 = 4$ events in the model. We can describe this model as follows:

- $S = \{ s_0, s_1 \}$
- $\text{stat}(\text{ifLoggedIn}, s_0) = \text{false}$
- $\text{stat}(\text{ifLoggedIn}, s_1) = \text{true}$
- $E = \{ e_{s00}, e_{s01}, e_{s10}, e_{s11} \}$
- $p(e_{s00}) = t$
- $p(e_{s01}) = t$
- $p(e_{s10}) = t$
- $p(e_{s11}) = u$

$M = (S, \Sigma, E)$

Figure 2 shows the diagram of the requirement of the log-in module.

3. Requirement inconsistency analysis

As the requirement model is composed of two sets, the source of inconsistency is also from these two sets, the state set and the event set. In another word, inconsistent requirement models are two requirement models of the same target system with different state set and/or event set. The goal of requirement inconsistency analysis is to determine a unified requirement model over inconsistent models, and thus the process of the requirement inconsistency analysis is the process of building the unified requirement model. For the purpose of conciseness, we will shortly write $M_{UM}$ as the unified model in the rest of this paper.

3.1. State inconsistency analysis

State equivalence relationship describes the relationship that different states in inconsistent model have the similar description. Requirement inconsistency analysis is based on the premise that the equivalence relationship between states of the inconsistent model is determined. The purpose of state analysis is to determine the equivalence relationship. A different analyzing step is carried out according to the different type of equivalence.

- **One to one equivalence**
  Given inconsistent model $M_A$ and $M_B$, by applying the principle of 2.2.1, we can conclude the equivalence set $ES$. A state set of the unified model can be reached by the equation below
  \[ S_{UM} = (S_A \cup S_B) - ES \]
  This means the state set of the unified model includes all states in the inconsistent models but no two states are equivalent.

- **One to many equivalence**
  We first describe the self-conflict situation in the one to many equivalence.
  Given $s_{Ai} \approx \{ s_{Bj}, s_{Bk}, \ldots \}$
  \[ \exists s_{Bx}: p(e_{BjBx}) \neq p(e_{BkBx}) \]
  This means when we treat $\{ s_{Bj}, s_{Bk}, \ldots \}$ as a single state in $M_B$, there will be two event from $\{ s_{Bj}, s_{Bk}, \ldots \}$ to state $s_{Bx}$ with different truth value, it’s obviously unreasonable. In such situation, we no longer treat $\{ s_{Bj}, s_{Bk}, \ldots \}$ as equivalence to $s_{Ai}$ but try one to one equivalence relationship between $s_{Ai}$ and every element in the set $\{ s_{Bj}, s_{Bk}, \ldots \}$

- **Many to many equivalence**
  We can convert many to many equivalence to one the many equivalence according to the definition 2.2.3.

3.2. Event inconsistency analysis
After determined state set of the unified model, we can use ternary logic operation on the truth value of the event of the inconsistent models to determined event set $E_{UM}$ of the unified model as follows:

$$
\begin{align*}
    & \text{if} \quad \text{source}(e_{UM_{sd}}) = s_{UMs} \\
    & \quad \text{destination}(e_{UM_{sd}}) = s_{UMd} \\
    & \quad \text{source}(e_{Asd}) = s_{As} \quad \text{destination}(e_{Asd}) = s_{Ad} \\
    & \quad \text{source}(e_{Bsd}) = s_{Bs} \quad \text{destination}(e_{Bsd}) = s_{Bd} \\
    & \quad s_{UMs} \approx s_{As} \approx s_{Bs} \\
    & \quad s_{UMd} \approx s_{Ad} \approx s_{Bd}
\end{align*}
$$

then

$$
p(e_{UM_{sd}}) = p(e_{Asd}) \ op \ p(e_{Bsd})
$$

$e_{UM_{sd}}$ represents the event from $s_i$ to $s_d$ in the event set $E_{UM}$ of the unified model. $e_{Asd}$ represents the event from $s_i$ to $s_d$ in the event set $E_A$ of $M_A$. $e_{Bsd}$ represents the event from $s_i$ to $s_d$ in the event set $E_B$ of $M_B$. $op$ represents ternary logic operator. The principle of this method is that the truth value of event $e$ of the unified model is calculated by using the specified operator to those events which have the equivalent starting state and ending state with $e$.

By using different ternary logic operators, different unified model can be reached. The conjunction operator $\land$ implements cautious inconsistency analysis, while disjunction operator $\lor$ implements optimistic inconsistency analysis. We can also use customized composite operations \textsuperscript{[4]} to implement complex requirement inconsistency analysis.

Some states could become isolated after the event inconsistency analysis.

$$
\exists \ s \forall \ e: \ s \neq s_0 \land \text{destination}(e) = s \land p(e) = u
$$

Because of no explicit event connection, such states should be removed from the unified model.

### 4. Implementation

#### 4.1. Overview

SRIA (Simple Requirement Inconsistency Analyzer) is a requirement inconsistency analyzing tool based on the method described above.

As showed in figure 3, SRIA has several functional modules. SRIA communicates with stakeholder’s XML file through ModelManager module, using UnitAnalyzer to carry out unit inconsistency analysis which only works on two inconsistent models. By using UnitAnalyzer recursively on all inconsistent models, the unified model is finally reached. TVL is a reduced ternary logic processor on which the event inconsistency analysis based.

#### 4.2. Requirement description

The premise of using our method is to convert stakeholder’s requirement specifications to our state machine based on requirement model. Among a variety of requirement specifications, XML is a good choice to represent stakeholder’s requirement specifications. XML is a popular markup language, which has several advantages of being used as requirement description.

- **General purpose**
  - XML is a popular markup language, which is widely used in the software field and therefore be supported by lots of tools, rules, or specifications, like the famous Document Type Definition (DTD) and XML Schema Definition (XSD)
- **Easy to understand**
  - XML is a language that is easy to understand, this is an important characteristic to stakeholder’s who have different background.
- **Customizable**
  - We can define customized tags in XML for requirement inconsistent analysis.

SRIA utilizes XML to communicate with stakeholders: stakeholders send their requirement specifications as XML file to SRIA, and receive the unified model specifications as XML file. We define a set of elements and related attributes for XML file used as requirement specifications.

- **Element <stetes> represent state set**
- **Element <state> represent a particular state.**
- **Element <param> represent state variable, attribute name represent the name of the state variable, attribute value represent the value of the state variable.**
1. Element <events> represent event set
2. Element <event> represent a particular event, attribute name represent the name of the event, attribute from represent the starting state of the event, attribute to represent the ending state of the event, attribute value represent the ternary logic truth value of the event.

The DTD listed below is used to check the validity of an XML requirement description.

```xml
<!DOCTYPE root [
<!ELEMENT root (states, events)>
<!ELEMENT states (state)>
<!ATTLIST state name CDATA #REQUIRED>
<!ELEMENT events (event)>
<!ATTLIST event name CDATA #REQUIRED >
<!ATTLIST event from CDATA #REQUIRED >
<!ATTLIST event to CDATA #REQUIRED >
<!ATTLIST event value CDATA #REQUIRED >]
```

For development considerations, the start state is not explicitly defined in the XML requirement descriptions. We define the state described in the first <state> as the start state.

### 4.3. Process of inconsistency analysis

1. Read requirement model from XML file
   SRIA can parse the XML file and build the requirement model accordingly. SRIA is designed to contain multiple requirement models for inconsistency analysis
2. Analysis state equivalence relationship and build state set of the unified model
   Determining the state set of the unified model is the core process of SRIA because it's the basis of all later process. SRIA builds an empty unified model at start of the analysis, and then add equivalent state until the state set is determined.
3. Choose the ternary operator to build the event set of the unified model
   Once the state set is determined, the event set can be calculated by using ternary operation. Because the event of the unified model is influenced by variety of factors, SRIA does not generate any operator automatically, but expose interface which allow user to define operator.
4. Export the unified model by output XML file
   SRIA converts the generated the unified model to XML file. In the way, stakeholders and requirement analysts can evaluate the result of the inconsistency analysis and choose better ternary logic operation for better result.

### 5. Example

Communication between the users is a very important feature of an Internet community. Users can get to know each other even make friends by supporting the similar viewpoint or having the same interests. In the process of making friends in such cyber world, there are two typical requirement descriptions.

#### 5.1. Requirement descriptions

- The requirement model ModelA of the person A who looking for friends
  Person A thinks he/she can make friends with B only if he/she wants.
  The XML requirement of ModelA are described below.

```xml
<root>
  <states>
    <state name="sA1">
      <param name="friendMade" value="F" />
    </state>
    <state name="sA2">
      <param name="friendMade" value="T" />
    </state>
  </states>
  <events>
    <event name="fireFriendA" from="sA1" to="sA2" value="T" />
  </events>
</root>
```

---

### Figure 4. Process sequence of SRIA

#### Figure 5.

State machine diagram of ModelA

- The requirement model ModelA of the person A who looking for friends
  Person A thinks he/she can make friends with B only if he/she wants.
  The XML requirement of ModelA are described below.

```xml
<root>
  <states>
    <state name="sA1">
      <param name="friendMade" value="F" />
    </state>
    <state name="sA2">
      <param name="friendMade" value="T" />
    </state>
  </states>
  <events>
    <event name="fireFriendA" from="sA1" to="sA2" value="T" />
  </events>
</root>
```
The requirement model ModelB of the person B who being searched by A thinks his/her opinion should be respected when A wants to make friends with him/her. The XML requirement of ModelB are described below.

```xml
<root>
  <states>
    <state name="sB1">
      <param name="friendMade" value="F" />
      <param name="requestSent" value="F" />
      <param name="requestAccepted" value="F" />
    </state>
    <state name="sB2">
      <param name="friendMade" value="F" />
      <param name="requestSent" value="T" />
      <param name="requestAccepted" value="F" />
    </state>
    <state name="sB3">
      <param name="friendMade" value="T" />
      <param name="requestSent" value="T" />
      <param name="requestAccepted" value="T" />
    </state>
  </states>
  <events>
    <event name="eB1B2" from="sB1" to="sB2" value="T" />
    <event name="eB2B3" from="sB2" to="sB3" value="T" />
    <event name="eB2B1" from="sB2" to="sB1" value="T" />
    <event name="eB1B3" from="sB1" to="sB3" value="F" />
  </events>
</root>
```

5.2. Process of requirement inconsistency analysis

5.2.1. State inconsistency analysis

\[ s_{A1} \approx s_{B1} \]
\[ \text{stat}(\text{friendMade}, s_{A1}) = \text{F} \]

5.2.2. Event inconsistency analysis

- **Analyzing event with disjunction operation**
  \[ E_{UM} = \{ e_{11}, e_{12}, e_{13}, e_{21}, e_{22}, e_{23}, e_{31}, e_{33} \} \]
  \[ p(e_{12}) = p(e_{B1B2}) = t \]
  \[ p(e_{21}) = p(e_{B2B1}) = t \]
  \[ p(e_{23}) = p(e_{A1A2}) \lor p(e_{B2B3}) = t \lor t = t \]
  \[ p(e_{31}) = p(e_{A1A2}) \lor p(e_{B3B1}) = u \lor t = t \]
  \[ p(e_{33}) = p(e_{B2B3}) = t \]

The unified model is shown in figure 7.

This unified model shows that A has two ways to make friends with B. One is directly making friends with B. The second is sending B a request, and making friends with B after B accepts the request. Because there is no dissolution information of friend relationship in ModelB, the unified model follows the process of ModelA. Consequently, A can dissolve directly the friend relationship.

Because the disjunction operation, the unified model retains as much information as it could. The problem of this operation is that it mask the event whose truth value is f. In this case, the limitations of ModelB are compromised.

- **Analyzing event with conjunction operation**
  \[ E_{UM} = \{ e_{11}, e_{12}, e_{13}, e_{21}, e_{22}, e_{23}, e_{31}, e_{33} \} \]
  \[ p(e_{12}) = p(e_{B1B2}) = t \]
  \[ p(e_{21}) = p(e_{B2B1}) = t \]
  \[ p(e_{23}) = p(e_{B2B3}) = t \]
  \[ p(e_{31}) = p(e_{A1A2}) \land u = f \]
  \[ p(e_{33}) = p(e_{A3A2}) \land u = u \]

The unified model is shown in figure 8.
This unified model describes that A first sends request to B, and makes friends with B after his/her request is accepted by B. A can’t directly make friends with B by using the request acknowledgement. With conjunction operation, the event whose truth value is F is emphasized. This operation is always used in the situation when some important information can’t afford to be lost. The problem is that it is sometimes too restrictive, and makes some information to be lost. In this case, the dissolution information of friend relationship in ModelA is lost.

6. Conclusions

State machine based requirement description can describe requirement context information via state transition, and thus has the ability of global description. By using ternary logic operation, the inconsistency analysis can be automated and the flexibility is also improved.

This method is still fragile to complex requirement inconsistency because of the absence of composite ternary logic operations. We will continue the research especially on the composite ternary logic operations in our future work.

Conferences


