RC24166 (C0701-020) January 29, 2007
Computer Science

IBM Research Report

Test Case Generation for Collaborative Real-time Editing Tools

Lian Yu*, Wenping Xiao, Changyan Chi, Lin Ma*, Hui Su
IBM Research Division
China Research Laboratory
Building 19, Zhouguancun Software Park
8 Dongbeiwang West Road, Haidian District
Beijing, P.R.C. 100094

*School of Software and Electronics
Peking University
Beijing, P.R.C. 102600

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Test Case Generation for Collaborative Real-time Editing Tools

Lian Yu[1]
School of Software and Electronics
Peking University
Beijing, 102600, PRC

Wenping Xiao
China Research Center
IBM
Beijing, 100094, PRC

Changyan Chi
China Research Center
IBM
Beijing, 100094, PRC

Lin Ma
School of Software and Electronics
Peking University
Beijing, 102600, PRC

Hui Su
China Research Center
IBM
Beijing, 100094, PRC

Abstract

Collaborative real-time editing tools (CRETs) present advanced editing features, and bring great challenge as well for verifying them. Test case generation is the key task of testing. Generating test cases efficiently with high quality is the goal of this paper. Collaboration is defined the core requirements of CRETs, from which functional features and design tactics are derived. The paper proposes a time-line diagram to visually model timing aspects and collaborative conflicts and ACDATE language to formally specify the corresponding test scenarios. The change of testing parameters and conflict resolution policy will incur significant time and effort for modifying the time line diagrams and programs. The paper presents an algorithm which allows configuring test parameters and collaboration policy on the fly; automatically generates textual test cases corresponding to the timeline diagram and test scripts in ACDATE language. A prototype shows the promising results of automatically generating test cases, both txtual and visual versions consistently.

Keywords: collaboration, time-line diagram, conflict region, ACDATE, automatically generating test cases.

1. Introduction

Real-time collaborative editing tools (CRETs) provide more exciting features than classical single-user editing tools. They allow in real-time fashion through Internet the collaboration among several participants, who are writing a same document using different computers from different places. Colleagues in a globalizing organization around the world can work on a same design artifact concurrently and see other's editing results instantly; a faculty in Beijing who is writing an NSFC proposal, can see what his partner on a business trip in New York is writing. CRETs impart better user experience for those kinds of situations that standalone editing tools can not afford.

The Web 2.0 techniques combined with the growing availability and speed of broadband internet access have caused an explosion of interest in browser-based document editing tools. For example, Google Docs & Spreadsheets [1] is a real time web based spreadsheet application and rich text editor; Synchroedit (rich text) [2] and MobWrite [3] projects are open-source attempts to do genuine real-time collaborative editing within a browser. Approaches including turn-taking protocols, locking or serialization-based protocols had been proposed to meet real-time and high concurrency requirements [4]; and various operational transformation algorithms have been proposed to maintain consistency [5][6].

At present, much time and effort are put on the design and development of CRETs, few attentions are paid to the research on quality assurance. The quality of CRETs either from vendors or open source community holds the key to success. Testing is considered as a core means to achieve the quality of assurance, and test cases generation is the key task of testing. Generating test cases efficiently with high quality for CRETs is the goal of this paper. Existing CRETs have a diversity of specification/ definitions in terms of real-time, concurrency algorithms, synchronous mechanisms, and conflict reconciliation policy. This paper identifies collaboration as the key requirement goal of CRETs, from which functional features – awareness, reflection and action, are derived; and tactics – real time, concurrency and synchronization, are established. Test case generation in the paper is based on the identified requirements.

CRETs are a kind of event driven systems for which event-response approach is often used to model the requirements specification and derive test cases. Event-response is a black-box testing technique, capturing

[1] The research is supported by Visiting Scholar Program and Pergola Project, IBM China Research Center.

events or stimuli from an external user, and identifying intended responses to that user. CRETs are imposed with collaboration requirements, where an event will be handled by the client, propagated to server, and reflected to other collaborator(s). Conventional event-response approach is no longer fitting in the situation. Testers need to identify not only response to the user, but also response to other collaborators, and response from intermediaries along the way. Gray box testing blends structural and functional testing methods throughout the testing procedure. This paper adopts gray box testing approach to generating the test cases for CRETs.

The rest of paper is organized as follows: Section 2 identifies the core requirements of CRETs. Section 3 proposes a timeline diagram to model test cases of CRETs visually, and ACDATE language to specify test cases formally. Section 4 presents an algorithm to automatically generate test cases. Section 5 describes the tool support. Section 6 provides a survey on related work. Finally, Section 7 concludes the paper and sketches future work.

2. CRETs' Requirements and Its Modeling

Collaboration distinguishes CRETs from the single-user editing tools. To model CRET requirements, we start from refining collaboration requirements.

Collaboration Requirements

In a social community, collaboration is considered as sharing of concepts that are nurtured, reworked and implemented according to group consensus. Collaboration requires active participation from all group members as well as consistent and open dialogue to avoid and resolve any conflict during the collaboration. Research in [7] summarizes the three main cornerstone of collaboration for a social community: awareness, reflection and action.

Entailed with collaboration, CRETs requirements have awareness, reflection and action as three functional features, which in turn are achieved by combining the tactics of real-time, concurrency, and synchronization.

- Awareness: CRETs should facilitate collaborators aware of others' work and others aware of local work in real-time, concurrent, and synchronous fashion.
- Reflection: The sensed information should be reflected to collaborators in real-time and consistent manner.
- Action: RECTs should smooth the progress for collaborators to take actions, including accept, reject, pend, or negotiate, to resolve any inconsistencies.

CRETs Requirements Modeling

Internet-based CRETs consist of a variety of elements, including collaboration server (CS), collaboration client (CC), network, and collaborators, and their relations are shown in Figure 1. After loading collaboration client program (e.g., JavaScript files) on browser, the collaborators edit and collaborate with others through network and collaboration server. CC takes care of awareness and reflection, collaborators command the action and dialog with other collaborators, while the CS synchronizes the global data.

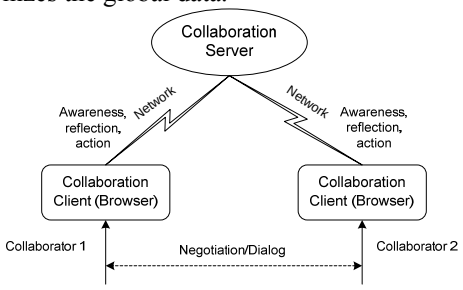

Figure 1: Architecture of Internet-based CRET Modeling

The Figure 2 shows the CC requirement model using Statechart diagram, which is composed of five orthogonal superstates, CC_Pulling, CC_Pushing, CC_Checking, CC_Warning and CC_Editing. When CC gets started, it loads CC program from CS into browser, and afterwards pulls updated data from CS in certain period of time, called *pull interval*. If there is any *conflict*, modify the pulled data and renders the results locally. The non-conflict pulled data are simply display on browser. In Section 3, we will give the explicit definition for "conflict". CC_Pulling and CC_Pushing model the behaviors of CC, in order to enable others knowing local changes and being aware others' changes respectively.

CC_Warning reflects conflict status by displaying on browse the conflict pending list if any, and allowing the collaborators to resolve detected conflicts by issuing commands of "accept", "reject", and "pend" on the pending list. CC_Editing provides primary features for CRETs, which facilitate any editing functions, such as add, delete, and modify. Changing sections or taking resolution actions are the events to trigger push action.

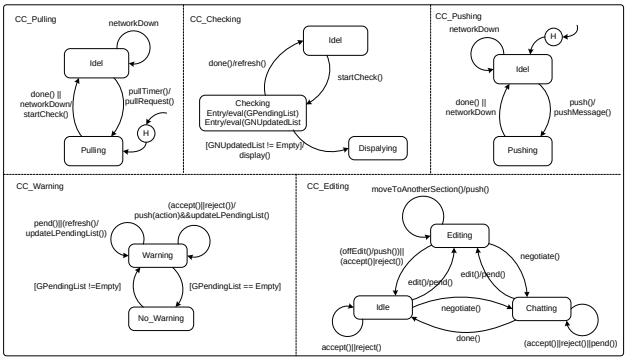

Figure 2: Requirements Model of Collaboration Client

Figure 3 demonstrates the behaviors of CS, which has two orthogonal superstates: CS_Monitoring and CS_Checking. In CS_Monitoring, CS starts from Monitoring substate when CS turns on. In CS_Checking, CS starts from Dequeueing substate when CS turns on. If the length of

push-message queue is not empty, CS enters Checking substate, evaluates any new changes against existing global data or pending lists, updates them if any, and otherwise returns to Dequeueing substate.

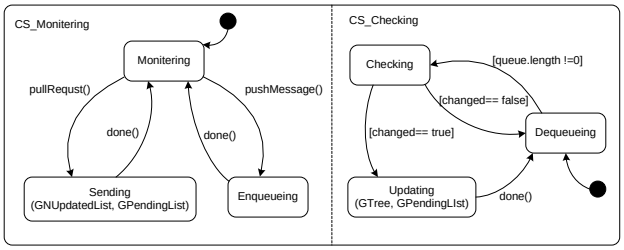

Figure 3: Requirements Model of Collaboration Server

Utilizing Statechart diagrams, Figure 2 and Figure 3 are drawn based on general collaboration requirements independent of any specific CRETs design approaches and implementation languages, and can be used as generic models for CRETs. Statechart diagrams are good at modeling system *behavior*, but not fit in modeling timing and collaboration issues as to *conflict*. We propose a time-line diagram to model these aspects of collaboration.

3. CRETs' Test Case Modeling

Test cases for single-user editing tools just need to prepare single user inputs and expected response of the tools. The difficulties with generating test cases for CRETs come from the functional features derived from collaboration, and the associated design tactics. This section describes a time-line diagram to annotate visually test cases and ACDATE language to specify the test cases formally for CRETs.

Figure 4 shows a Scenario *A* of CRET using time-line diagram, in which two collaborators are editing a same document. For each collaborator, we define a *baseline*, below which there are *external event/activity lines* indicating collaborator's inputs, above which there are *pull-lines* indicating that CC pulls global updates periodically, checks any conflicts with local updates, displays or refreshes information on its browser; *push lines* indicating that CC pushes the local updates upon the arrival of an external event or the completion of external inputs; *time window* indicating the period of time for pushing message on CC, wiring message over network, processing the message from CC. The beginning or end points of lines are projected with *time reference line* to *Time line* at the bottom of diagram.

Using narrative language, Scenario *A* can be described as "At t_0, Collaborator 1 (C_1) starts editing section S_1 and finishes up at t_3, which immediately triggers push(S_1) event; after the period of the time window at t_6, in the next pull run, CC_2 pulls S_1 from CS and displays it locally at t_7; at t_1, Collaborator 2 (C_2) starts editing section S_3 and finishes up at t_2, which immediately trigger push(S_3) event; after the period of time window at t_4, in the next pull run, CC_1 pulls S_3 from CS and displays it locally at t_5; at t_4, C_1 starts editing section S_2 and finishes up at t_8, which immediately triggers push(S_2) event; after the period of the time window at t_9, in the next pull run, CC_2 pulls S_2 from CS and displays it locally at t_{10}." The time of points, t_0 through t_{10}, satisfies time sequential order."

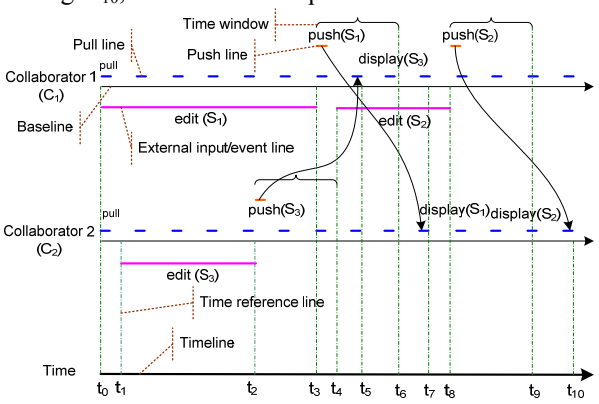

Figure 4: Scenario *A* of a CRET using time-line diagram

The time-line diagram helps capturing collaborators' motions and corresponding consequences as time goes on: one collaborator initiates an event/activity at a moment or during a period time, causes collaboration client responses at certain time; server ends up processing at other time; later on the result reflects to other collaborator(s). The time-line diagram facilitates human being to communicate but is not suitable for machine to execute. The following section introduces ACDATE language to specify the test cases annotated in the time-line diagram.

ACDATE Specification Language

ACDATE stands for the acronym of *A*ctor, *C*ondition, *C*onstraint, *D*ata, *A*ction, *T*iming and *E*vent, which are used as model elements. The semantics of ACDATE model can be represented as state-transition: if an actor is in the *pro-condition*, the actor performs an executable computation and transits to post-condition when the actor receives the trigger *event* of the transition and if the *guard* condition, if any, is satisfied. The computation may directly act on the actor that owns the state machine, and indirectly on other actors that are visible to the actor by sending out event(s).

ACDATE language is used for test case specification, and consists of two parts: testing definition and testing logic. Testing definition part creates instances of the six elements. List 1 shows testing definition of Scenario *A* in Figure 4 where six instances of actors are created, CS, CC_1, CC_2, C_1, C_2, and NW (network); initial condition is prepared; three events and timings are defined; data/states along time are specified; three timing patterns and three actions are identified.

Testing logic part stipulates testing algorithm. List 2 shows the testing logic for test case in Figure 4. ACDATE

language provides stimuli-verify template for testing steps: *stimuli* section specifies inputs and/or events in some timing styles to the system under test (SUT); *verify* section specifies sequential responses from local client, CS and other collaborative clients in some other timing characteristics; captures the actual results and compared with the expected results prepared in the testing definition section.

List 1: Testing Definition of Test Scenario *A*

```
@Actor:
    CS; CC1; CC2; C1; C2; NW;
@Condition:
    initCondition{CS.start()#before(t0); CC1.start()#before(t0);
                  CC2.start()#before(t0); NW.start()#before(t0);}
@Event:
    C1.edit(S1).moveMouse(S1, S2)#at(t3);
    C2.edit(S3).moveMouseOff(S3)#at(t2);
    C1.edit(S2).moveMouseOff(S2)#at(t8);
@Data:
    (CC1,S1,t0); (CC1,S1,t3); (CS,S1,t6); (CC2,S1,t7);(CC2,S3,t1); (CC2,S3,t2);
    (CS,S3,t4); (CC2,S3,t4);(CC1,S2,t4); (CC1,S2,t8);  (CS,S2,t9); (CC2,S2,t10);
    pullTime; pullRate; pushTime; transferTime; processingTime;
    (t7−t6) : between (pullTime, pullTime+1/pullRate);
    (t6−t3) : equal (pushTime+transferTime+processingTime );
@Timing:
    during(startPoint, endPoint); at(timeOfPoint); before(timeOfPoint);
@Action:
    push(); update(); display();
```

List 2: Testing Logic of Test Scenario *A*

```
1. setup:
        initCondition;
2. stimuli:
        C1.edit(S1)#during (t0, t3); C1.edit(S1).moveMouse(S1, S2)#at(t3);
3. verify:
        CC1.push(S1) .start() # at(t3); CS.update(S1).done()#at(t6);
        CC2.display(S1).done()#at(t7);
4. stimuli:
        C2.edit(S3) #during (t1, t2); C2.edit(S3).moveMouseOff(S3)#at(t2);
5. verify:
        CC2.push(S3).start() # at(t2); CS.update(S3).done()#at(t4);
        CC1.display(S3).done()#at(t5);
6. stimuli:
        C1.edit(S2) #during (t4, t8); C1.edit(S2).moveMouseOff(S2)#at(t8);
7. verify:
        CC1.push(S2).start()#at(t8); CS.update(S2).done()#at(t9);
        CC2.display(S2).done()#at(t10);
```

The separation of testing definition and testing logic enables the reuse of testing logic part and even in the case of changes of testing data change, but does not pledge the reuse. As we will see in the subsection that when the timing of events or actions changes to some extend such that collaboration conflict incurs, the test steps have to be changed correspondingly. The following section identifies the threshold.

Time-line diagram with Conflict – Scenario *B*

Scenario *B* in Figure 5 has different characteristics from Scenario *A* in Figure 4. In Scenario *B*, C_1 finishes up editing S_1 at t_2, however, *before* the editing result shows up on CC_2 at t_6, C_2 starts editing the *same* section S_1. The interval from t_2 to t_6 is the *conflict region*, which is associated with section S_1 among C_1 and C_2 and equal to, on average, the summation of time window, pullTime and 1/(2* pullRate). As a matter of fact, as long as the timing that CC_2 pushes S_1 falls into the conflict region, conflict occurs. As C_1's editing result reaches CS early than C_2's, CS uses C_1's S_1 to update the global data, and puts C_2's into the corresponding pending list. As long as the conflict is not resolved, whenever C_1 updates S_1, as does during t_8 and t_{10}, CS will use it to update the corresponding global data. Because CC_1's push(S_1) event occurs at t_{10} out of the conflict region of C_2 on S_1, there is no conflict between C_1 and C_2 associated S_1 during t_3 to t_9.

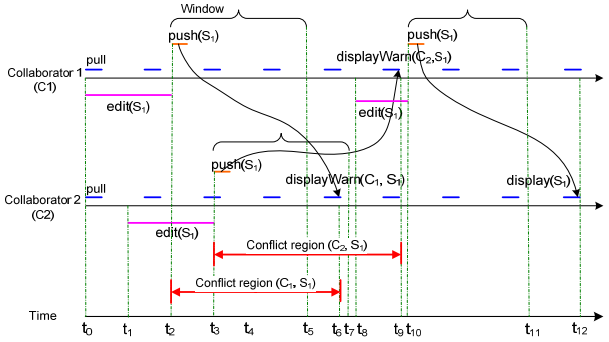

Figure 5: Scenario *B* of a CRET with conflict region

Definition: *Conflict occurs regarding to a section S if and only if one client's push event occurs at t_i within another client's conflict region t_j and t_k, i.e., $t_j \leq t_i \leq t_k, i \neq j \neq k$.*

Compared with List 1 of Scenario *A*, List 3 of Scenario *B* includes *pendList*s of CS, C1 and CC2 into @Data section, *check*() and *displayWarn*() into @Action section

List 3: Testing Definition of Test Scenario *B*

```
    ……//omit @Actor, @Condition and @Timing sections
@Data:
    CC2.pendList: (S1,C1,C2,t7); CS.pendList: (S1,C1,C2,t8);
    CC1.pendList: (S1,C1,C2,t9);CC2.pendList: (S1,C1,C2,t12);
    ……//omit  other test data in @Data section
@Action:
    push(); update(); check(); display(); displayWarn();
```

List 4: Testing Logic of Test Scenario *B*

```
1. setup:
        initCondition;
2. stimuli:
        C1.edit(S1)#during (t0, t2); C1.edit(S1).moveMouse(S1, S2)#at(t2);
3. verify:
        CC1.push(S1).start() # at(t2); CS.update(S1).done()#at(t6);
        CC2.check().conflictTrue()#at(t7);CC2.displayWarn(C1,S1).done()#at(t7);
4. stimuli:
        C2.edit(S1)#during (t1, t3); C2.edit(S3).moveMouse(S1,S2)#at(t3);
5. verify:
        CC2.push(S1).start()#at(t2); CS.addToPendList(C2,S1).done()#at(t8);
        CC1.displayWarn(C2,S1).done()#at(t9);
8. stimuli:
        C1.edit(S1)#during (t10, t12); C1.edit(S2).moveMouseOff(S1)#at(t12);
9. verify:
        CC1.push(S1).start()#at(t12);CS.update(global(S1)).done()#at(t15);
        CC2.display(pendList).done()#at(t16);//changed S1 in pendList
```

Correspondingly, in the testing logic of List 4, local conflict check of CC_2 on S_1 at t_7 is inserted before displaying contents pulled from CS, and identified conflict

with CC_1 is displayed together the pulled content on CC_2's browser. When S_1 of CC_2 arrives at CS, CS makes out the conflict and puts in the pendList. The rest of steps is similar to that of List 2, but appending the pendList on CS, CC_1 and CC_2.

Time-line Diagram Resolving Conflict-Scenario *C*

Whenever there is a conflict occurred to a section *S*, CS will maintain a global data of *S*, and associated pending list indicating which other client(*S*) had different version of *S*. The conflict information, the global data together with the pending list, is pulled and reflected in an intended timing on client browsers for collaborators to resolve. Collaborators can take three kinds of actions with conflict:

- Accept: one accepts another's editing results.
- Reject: one rejects another's editing results.
- Pend: a dummy action that a collaborator does anything than accept and reject regarding a section.

Figure 6 shows a Scenario *C*, where collaborators intend taking action to resolve the conflicts. The initial condition of the scenario is "CC_1 has a conflict on Sections S_1 and S_2 with CC_2, and CC_1's results as global data while CC_2's put into pending lists." In Scenario *C*, C_1 accepts C_2's S_2 at t_7, however, *before* the action result shows up on CC_2 at t_{11}, C_2 accepts C_1's S_2 at t_9, incurring the conflict.

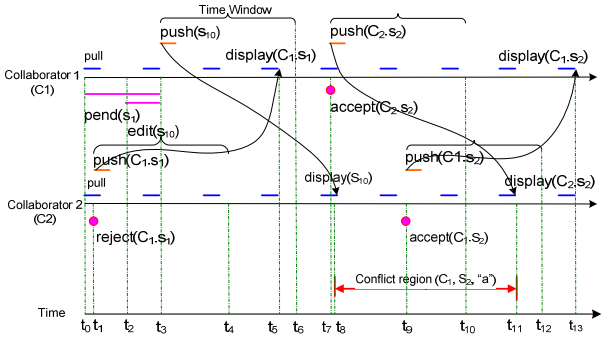

Figure 6: Scenario *C* with conflict resolution actions

The ways to resolve conflict change from application to application. Assume that there are two collaborators, C_1 and C_2; C_1 and C_2 have conflict on section *S*, and C_1's result as global, and C_2's as *pendList* entry. Table 1 shows an example of conflict resolution policy for 9 situations. The first 5 situations indicate no conflict occurs, while the last 4 situations incur conflicts of resolution actions. "Pend" in the table means the corresponding collaborator does anything but accepts or rejects the counterpart's results on section *S*. In contrast to List 3, List 5 adds *dataCondition* in the @Condition section, *accept*() and *reject*() in the @Event section. Correspondingly, List 6 shows up the events in *stimuli* portions.

Constructing the time-line diagram and writing the language program is time consuming. Moreover, test parameters and collaboration policy may change or even be unknown beforehand; any of the changes will incur significant time and effort for modifying the time-line diagrams and the programs.

Table 1: An Example of Conflict Resolution Policy

id	C_1 Action	C_2 Action	Server Policy
1	pend	pend	No change
2	pend	accept C_1's	1) Updates C_2.accept(C_1, S) after a window time 2) Delete pending entry of C_2 from the list after "2*window time + 1/C_1.pullingRate"
3	pend	reject C_1's	1) Updates C_2.reject(C_1, S) after a window time 2) Replace/swap C_2's as Global, C_1's as pending entry after "2*window time + 1/C_1.pullingRate"
4	accept C_2's	pend	1) Updates C_1. accept(C_2, S) after a window time 2) Replace/swap C_2's as Global, C_1's as Pending entry after "2*window time + 1/C_2.pullingRate"
5	reject C_2's	pend	1) Updates C_1. reject(C_2, S) after window time 2) Delete pending entry of C_2. from the list after "window time + 1/C_2.pullingRate + window time"
6	accept C_2's	accept C_1's	1) Receives the two messages within the window time, 2) No change to the global, update pending list's flag
7	reject C_2's	accept C_1's	1) Receives the two messages within the window time, 2) No change to the global, delete pending entry of C_2
8	accept C_2's	reject C_1's	1)Receives the two messages within the window time, 2)Replace with C_2 as global, delete pending entry of C_2
9	reject C_2's	reject C_1's	1) Receives the two messages within the window time, 2) No change to the global, update pending list's flag

List 5: Testing Definition of Test Scenario *C*

```
//omit  @Actor section and conditions and events similar to those in Lists 1 or 2.
@Condition
      dataCondition{(CS.pendList !=empty)#before(t0);}
@Event:
      C2.reject(C1,S1)#at(t1);C1.accept(C2,S2)#at(t7);C2.accept(C1,S2)#at(t8);
@Data:
      CS.pendList: (S1,C1,C2.reject(C1),t4), (S2,C1,C2,t4);
      C1.pendList: (S1,C1,C2.reject(C1),t5), (S2,C1,C2,t5);
      CS.pendList: (S1,C2,C1,t9), (S2,C1.accept(C2),C2,t9);
      CC2.pendList: (S1,C2,C1,t10), (S2,C1.accept(C2),C2.accept(C1),t10);
      CS.pendList: (S1,C2,C1,t11), (S2,C1.accept(C2),C2.accept(C1),t11);
      CC1.pendList: (S1,C2,C1,t12), (S2,C1.accept(C2),C2.accept(C1),t12);
//omit  @Timing and @Action sections similar to those in Lists 1 or 2.
```

List 6: Testing Logic of Test Scenario C

```
1. setup:
      initCondition; dataCondition;
2. stimuli:
      C2.reject(C1,S1)#at(t1);
3. verify:
      CC2.push(C2.reject(C1,S1)).start()#at(t1);
      CS.updatePendList(S1,C1,C2.reject(C1)).done()#at(t4);
      CC1.display(CC1.pendList).done()#at(t5);
…...//omit atomic scenario – C1 edits S10.
6. stimuli:
      C1.accept(C2,S2)#at (t7);
7. verify:
      CC1.push(C1.accept(C2,S2)).start()#at(t7);
      CS.updatePendList(S2,C1.accept(C2,S2),C2).done()#at(t9);
      CC2.check().conflictTrue()#at(t10);
      CC2.displayWarn(C1.accept(C2,S2)).done()#at(t10);
8. stimuli:
      C2.accept(C1,S2)#at (t8);
9. verify:
      CC2.push(C2.accept(C1,S2)).start()#at(t8);
      CS.updatePendList(S2,C1.accept(C2,S2),C2.accept(C1,S2)).done()#at(t11);
      CC1.display(CC1.pendList).done()#at(t12);
```

Although ACDATE language enables separating testing data and testing logic, certain thresholds of testing data changes will incur the change of testing logic as discussed in Section 3. The following section presents an algorithm

which allows configuring test parameters and collaboration policy on the fly, and automatically generates textual test cases corresponding to the time-line diagram and test scripts in ACDATE language.

4. Automation of Test Case Generation

This section presents the algorithm for automatically generating test cases of CRETs and analyzes the generated results.

Algorithms to Generate Test Cases

We define as *atomic* test scenario a pair of *stimuli* and *verify* in testing logic with ACDATE language, corresponding to event of a collaborator and associated responses of local client, CS and collaborative clients in a time-line diagram. One atomic test scenario (ts_1) affects or is affected by another atomic test scenario (ts_2) if *push*() action of ts_1 falls into the conflict region of ts_2 or vice versa. *Complex* test scenario consists of more than one atomic test scenario in a consistent way, i.e., detecting conflicts correctly. Figures 4 through 6 are complex test scenarios. To formally present the algorithm of test case generation, we have the following definitions:

- A Set of actors triggering events and taking responses, A={CS, {CC_i } (i = 1, …p), {C_i}, NW}
- O Set of conditions including data conditions and actor conditions
- D Set of data, D_1={S}: edited sections, D_2={P}: pendList of CS and CC
- E Set of events
- Π Set of responses of CC_j, CS, CC_k, j≠k
- T Set of timing patterns
- Δ Set of conflict regions
- Γ Set of test scenario types
- Λ Set of atomic test scenarios, Λ={(Φ, E, Π, T, Δ)}
- X Set of complex test scenarios, X={(Γ, Λ_1,…, Λ_m)}

According to ACDATE model, atomic test scenario can be specified as: under conditions {o}⊆O, upon an event of a collaborator $e \in E$ with timing pattern $t_i \in T$, CCs and CS take responses {π}⊆Π with timing patterns {t} ⊆ T, and data conditions change to {d}∈D. In Table 2, we call as factors sets that affect test case generation, and instances of sets as levels.

List 7 shows the pseudo-code of algorithm for automatically generating test scenarios, composing of three parts. The first part, configuration, prepares relevant parameters, conflict resolution policy and scenario type. The second part delineates the steps to generate atomic test scenario including generating pre-condition, incoming event from a collaborator, verifying collaborative clients and server's response following timing constraints. When generating test scenario with conflict, CS and other collaborators need to check the global and local conflicts, respectively. The third part aggregates a certain number of atomic test scenarios to generate complex test scenario.

Table 2: A List of Factors of Levels

ID	Factors	Levels
1	Scenario type	{With Conflict, Without conflict}⊆ Γ
2	Collaborator event	{edit().moveMouse(), edit().moveMouseOff(), accept(), reject(), pend(), negotiate()}⊆ E
3	Collaboration Client (CC)	{start(), stop()}⊆O;{ push(), push().failed(), pull(), pull().failed(), check(), check().failed(), display(), display().failed(), displayWarn(), displayWarn().failed, refresh(), refresh().failed()} ⊆Π
4	Network (NW)	{connected(), down()} ⊆O
5	Collaboration Server (CS)	{start(), stop()}⊆O; {update(), update().failed(), check(), check().failed()}(⊆Π)
6	Pending list on CS	{CS.pendList, empty(), !empty()}⊆D
7	Pending list on CC	{CC.pendList, empty(), !empty()}⊆D
8	Timing Patterns	{during(startT,endT), at(time), before(time)} ⊆ T

List 7: Algorithm to automatically generate test scenarios

```
//Configuration: to get parameters and conflict resolution policy
configure(){
  getParameter(pull time, pull rate, push time, transfer time, process time);
  getPolicy(resolution policy);
  getScenarioType(scenario type: γ∈Γ);}
// Generation of atomic test scenario
gen_atomic_testScenario(Ci){
  gen_pre_condition({o}) ; // {o}⊆O;
  if editting, gen_edit();
  gen_collab_stimuli e∈E on Ci.sm ∈ S;
  gen_verify(){
    gen_CCi.push(sm){
      if γ = = withoutConflict{
        CCi.push(sm)_timing (t ∈ T) not in (∀CCj.sm.δ∈Δ), CCj.sm ∈{s}
          &&∀CCj.push(sm)_timing (t' ∈ T) not in (CCi.sm.δ∈Δ),CCi.sm ∈{s}
      } else if γ = = withConflict{
        CCi.push(sm)_timing (t ∈ T) in (∃CCj.sm.δ∈Δ), CCj.sm ∈{s}
          || ∃CCj.push(sm)_timing (t' ∈ T) in (CCj.sm.δ∈Δ) of CCi.sm∈ {s};}}
    gen_CS_res(CS.{π}){ //CS.{π}⊆Π;
      if γ = = withConflict{
        gen_CS_checkConflict()#(t2 ∈ T); //timing pattern t2
        gen_ CS_update_pendList()#(t3 ∈ T); //timing pattern t3
      }else gen_CS_update()#(t4 ∈ T);} //timing pattern t4
    for each CCj ({CC}-{CCi}){ // i≠j
      gen_CCj_res({πj}){ //CCj.{πj}⊆Π,
        if γ = = withConflict{
          gen_CCj_checkConflict()#(t5 ∈ T); //timing pattern t5
          gen_ CCj_update_pendList()#(t6 ∈ T); //timing pattern t6
        }else gen_CCj_update()#(t7 ∈ T); //timing pattern t7; }}}
  rec_post_condition ({o'}) ;} // {o'}⊆O
// Generation of complex test scenario
gen_complex_testScenario(){
  gen_numOf_atomic_test_cases(num);
  call configure(); select Ci ∈{C};
  label L: call gen_atomic_testScenario(Ci);
  select Ci ∈{C} with earliest endTime of E;
  num--; if num != 0, goto: label L;}
```

Analysis on the Test Case Generation

The proposed approach to test cases generation can be characterized as follows:

- Configurable: With the support of automatically generating test cases, the proposed approach takes the changing factors as configurable parameters to the algorithm, thus saves much of time and effort in generation of test cases.
- Completeness and Consistency: Test scenarios are

generated by properly combining atomic scenario elements and itself. There are two types $\gamma \in \Gamma$ of atomic scenarios. *Condition* includes conditions $A.\{o\} \subseteq O$ of CS, CC and NW and data conditions $\{d\} \in D_1 \cup D_2$. *Event* is chosen from C.*E*, and has dependency with the conditions, e.g., if the CC_1 condition is off, no event in $C_1.E$ can be chosen. *Actions/responses* can be chosen from $CS.\Pi \cup CC.\Pi$. Depending on types of event and responses, *Timing* is suitably chosen from T.

5. Tool Support

The support tool consists of one sharing database and three components for configuration, test case generation and timeline diagram generation:

- Configuration utility: Tester can use GUI or edit XML file to define parameters and conflict resolution policy. The specified parameters are passed to test cases generator or store into database for later use.
- Test case generator: Automatically generate a variety of test scenarios taking the passed parameters. The output of test case generator is plain text version, as shown in Lists 1 through 6, and taken as input to time-line diagram generator or stored into database for test execution.
- Time-line diagram generator: It is implemented using SVG technique, i.e., the output is XML format and follows SVG specification rendering on SVG browsers or viewers.

Figure 8 reveals the practice using proper design patterns to achieve the design goals. Strategy design pattern allows testers to define different conflict resolution policies for different applications or for same application but in different situations. Template method *createTestSceanrio* defines the skeleton of test steps, receiving stimuli from one collaborator, verifying responses from client, server and other collaborator(s), correspondingly; deferring some steps to subclasses. Composite pattern facilitates creating complex test scenarios based on atomic test scenarios or existing complex test scenarios, letting treat atomic test scenarios and complex test scenarios uniformly.

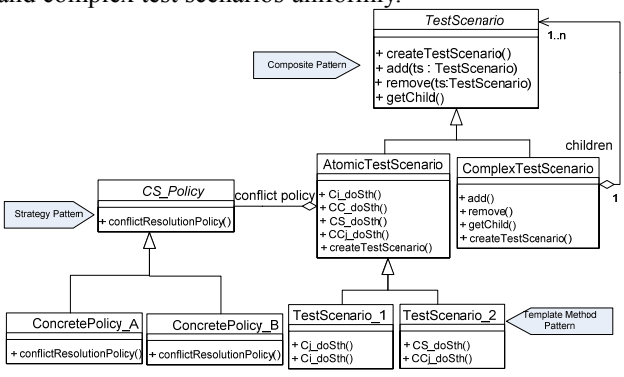

Figure 7: Design of Generating Test Cases

Figure 9 exemplifies the screenshots of the three components: 1) a configuration GUI for a tester to specify the parameters; 2) generated testing definition and testing logic for Scenario *C*; 3) SVG-based time-line diagram for that scenario. In contrast to the hand-written List 5 and List 6 and Figure 8 drawn in Visio, the generated version is more efficient with fewer errors.

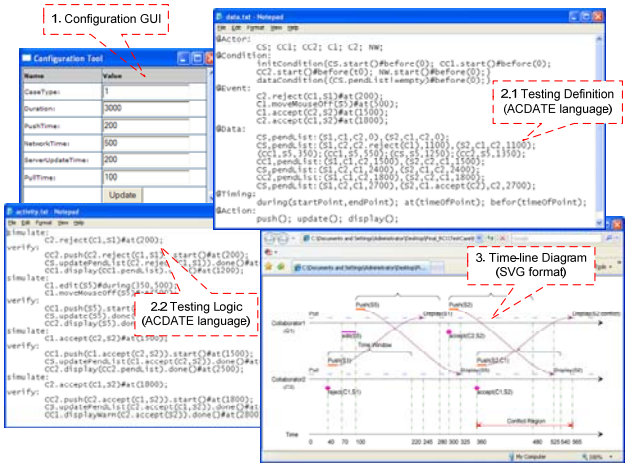

Figure 8: Screenshots of tool support prototype

6. Related Work

As CRETs gain more and more popular, their quality will draw much more attention. Although little literature was published that directly addressing issues of the quality assurance, we find the survey in this section is supportive.

State-based approach is often used to generate test cases following the process: build state model based on requirements; flat the model if the former is hierarchical; traverse the model to get test paths; and instantiate each path with data to get test cases[8][9][10]. The process can be performed automatically by writing a program to do so. In a collaborative situation, the mechanical combination and enumeration can not mirror timing constraints on the responses and conflict regions of responses as Statechart has no constructs to model two aspects, The proposed time-line diagram can be used to model along time dimension the motion and its consequences of every collaborator simultaneously; identify conflict region associated with a collaborator on some sentence, then check if the timing of other's push events falls into the conflict region to assert whether there exists conflicts among the collaborators as shown in Figures 5 and 6.

The Tree and Tabular Combined Notation (TTCN) is defined and standardized in Part 3 of the international standard 9646 OSI Conformance Testing Methodology and Framework (CTMF) [11]. OSI conformance testing is understood as functional black-box testing, i.e., an implementation under testing (IUT) is given as a black-box and its functional behavior is defined in terms of inputs to and corresponding outputs from the IUT.

Subsequently, TTCN test cases describe sequences of stimuli to and expected responses from the IUT. The third edition, CCTN-3 [12], comprises more extensions including the handling of test verdicts, matching mechanisms to compare the reactions of the IUT with expected range of reactions, timer handling, distribution of tester processes and the ability to specify encoding information. CCTN embraces the approach of functional black-box testing, and addresses the specification of test cases for communication among distributed systems. It does not provide information on execution path through the system. This paper adopts grey-box testing technique for CRETs, allowing test program to interact with APIs of CRETs' components on execution path of interest. The specification of test cases for CRETs covers collaborators involved, collaboration clients, collaboration server and network communication.

ACDATE model and language was developed to denote scenario-based test specification, and has been successfully applied to a variety of systems, e.g., command and control systems, communication processor systems, UDDI-based applications, and manufacturing control systems [13][14][15]. This paper has three extensions to ACDATE language:

- Add constructs for specifying timing and conflict to mirror the collaboration requirements;
- Provide templates for specifying testing CRETs logic to enhance the reusability and ease of understanding;
- Refactor the language structure and realize separation of test data and test logic in order to increase the reusability and maintainability of test code.

7. Conclusion and Future Work

The paper makes clear that collaboration is the requirement goal of CRETs, from which derives features of awareness, reflection, and action, while real-time, concurrency, synchronization/asynchronization are design tactics to achieve the goal. Based the requirement, Statechart is used to describe the behavior, and a time-line diagram is proposed to visually model timing and collaborative conflicts. ADACATE language is introduced formally specify test scenarios each of which consist of testing definition and testing logic. The testing definition part creates instances or test data, while testing logic provides testing algorithms and templates. Drawing time-line diagram and writing ACDATE program are time-consuming and error-prone; an algorithm is proposed to automatically generating ACDATE programs and time-line diagram. A prototype of the tool support is implemented to validate the feasibility of the algorithm. Two tasks are undergoing, establishing testing execution framework to automatically execute the testing using the generated test scenarios described in this paper; formally proving the correctness of the algorithm. The results will be published in our next paper.

Reference

[1]. Google Docs & Spreadsheets: http://www.google.com
[2]. SynchroEdit: http://www.synchroedit.com
[3]. MobWrite: http://neil.fraser.name/software/mobwrite
[4]. S. Greenberg, D. Marwood, "Real-time groupware as a distributed system: Concurrency control and its effect on the interface". Proc. Of The ACM Conference on Computer Supported Cooperative Work, North Carolina, October 1994, pp. 207-218.
[5]. C. Ignat, and M. Norrie, "Tree-based Model Algorithm for Maintaining Consistency in Real-Time Collaborative Editing Systems", Proceedings of the Fourth International Workshop on Collaborative Editing Systems, CSCW 2002, New Orleans, USA, November 2002.
[6]. P. Dewan, R. Choudhary, and H. Shen, "An Editing-based Characterization of the Design Space of Collaborative Applications." Journal of Organizational Computing, Vol.4, Ablex, Norwood, NJ, USA, 1994, pp. 219-240.
[7]. "Model for Build Collaboration", Prepared for: Social Planning Council for the North Okanagan, Prepared by Tina-Marie Christian, MA.OM, B.Ed Armstrong, BC, April 2003 URL: http://www.socialplanning.ca/health/building_collaboration_report.pdf
[8]. S. Gnesi, D. Latella, and M. Massink, "Formal test-case generation for UML statecharts", Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS'04) , IEEE Computer Society, Washington, DC, USA, 14-16 April 2004, pp. 75-84.
[9]. M. Chen, X. Qiu, and X. Li, "Automatic test case generation for UML activity diagrams", Proceedings of the 2006 international workshop on Automation of software test , ACM Press, New York, NY, USA, 2006, pp. 2-8.
[10]. K. Chang, W.H. Carlisle, J.H. Cross. II, and D.B. Brown, "A heuristic approach for test case generation", Proceedings of the 19th annual conference on Computer Science, ACM Press, New York, NY, USA, 1991, pp. 174-180.
[11]. T. Walter, and J. Grabowski. "Real-time TTCN for Testing Real-time and Multimedia Systems", Testing of Communicating Systems, volume 10, Chapman & Hall, September 1997, pp. 37-54.
[12]. J. Grabowski, "TTCN-3 - A new Test Specification Language for Black-Box Testing of Distributed Systems", Proceedings of the 17th International Conference and Exposition on Testing Computer Software (TCS'2000), Washington D.C., June 2000.
[13]. W. T. Tsai, Lian Yu, Zhu P, Paul R., "Rapid embedded system testing using verification patterns", IEEE SOFTWARE 22 (4): 68-75, JUL-AUG 2005.
[14]. W. T. Tsai, Lian Yu, Ray Paul,; Chun Fan,; Xinxin Liu,; Zhibin Cao, "Rapid scenario-based simulation and model checking for embedded systems", Proceedings of the Seventh IASTED International Conference on Software Engineering and Applications, 2003, p 568-573.
[15]. W. T. Tsai, Lian Yu, A. Saimi, R. Paul, "Scenario-based Object-Oriented Test Frameworks for Testing Distributed Systems", Proceedings of IEEE Future Trends of Distributed Computing Systems, 2003. pp. 288-294.