INTRODUZIONE E DESCRIZIONE GENERALE

Un rilascio di versione software si effettua:

1. Per aggiungere una prestazione al sistema
2. Per correggere un difetto esistente.

In entrambi i casi è richiesta un'attività di test di "non regressione" sulle feature: occorre verificare che non siano presenti "deterioramenti" di tutte le prestazioni di sistema, anche quelle non direttamente coinvolte dalla modifica.
Questi test, se eseguiti manualmente, sono costosi in termini di tempo, occupazione del materiale, sono attività "ad alta intensità" e monotoni; inoltre spesso in queste attività è facile incappare in un errore umano e per questi motivi sono particolarmente "sgraditi" per il personale del test.

Invece, se automatizzati, I test di regression possono essere molto dettagliati e efficaci; l'apporto del test automatico è quello di alleggerire i tester manuali per avere più tempo per svolgere test complessi.

Uno dei limiti di questa attività di test è legata alla scarsa flessibilità nella modifica delle configurazioni: per questo motivo è "oneroso" sia modificare le configurazioni di test che utilizzare test già esistenti in altri banchi.

In generale l'ambiente di test automatici è composto da:

1. Banco di test.
2. Sistema di automazione.

Banco di test è un insieme di Apparati (Nodi) e Strumenti (che possono essere di produzione di Ericsson o di terze parti), su cui sono installate diverse versioni dei software in produzione o in manutenzione. Di solito il banco di test rappresenta una porzione minima "significativa" della rete del cliente. La versione software del nodo è la "componente" sottoposta a certificazione di non-regressione.

Ogni Nodo fisico conserva la configurazione in un file (database o “cdb”): si tratta di un insieme di file xml strutturato e compilato sulla base dell’IM, che descrive il Nodo stesso. In questo modo conserva informazioni come quali schede sono configurate, in che posizione, con quale configurazione di traffico.
I file cdb sono quindi un IM semplificato in cui compaiono solo le entità davvero presenti su quel nodo, con i valori correnti. Il Modello Informativo, sviluppato in UML (e quindi in xml), descrive ad alto livello solamente il Nodo, ma non gli eventuali strumenti ad esso collegabili.

Il cdb viene generato dal sistema attraverso comandi di configurazione che sono in grado di “capire” cosa è presente sul Nodo (e i relativi valori dei campi) e di dare indicazione sulla configurazione desiderata e attraverso questo il sistema crea i file del cdb che descrivono queste entità.

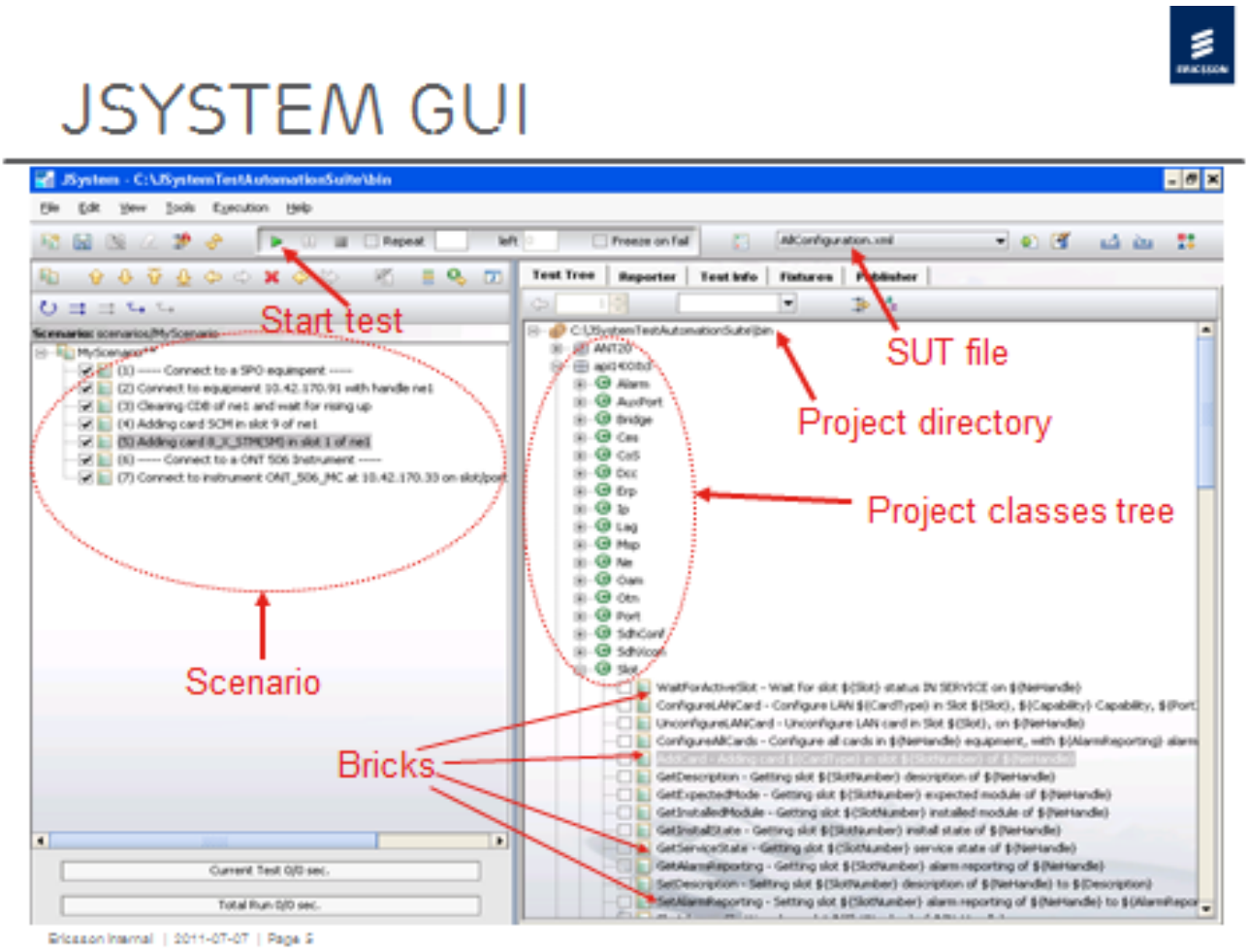
Il sistema di automazione si sostituisce a questi tools per programmare una serie di “operazioni” verso il Banco (sia apparato che tutti gli strumenti connessi) sia di configurazione che di esecuzione di test.

Struttura dei test automatici

Un test case (di seguito “test” per semplicità) consiste una sequenza di passi (operazioni), per verificare la correttezza del comportamento e/o delle funzionalità software. Di solito viene fornito un risultato o comportamento atteso. Normalmente il comportamento ad alto livello è descritto in un file word, cui sono poi associati eventuali allegati, fra cui la descrizione del banco su cui quel test deve essere eseguito.

Ericsson usa JSystem come sistema di automazione. Anche se è possibile utilizzare JSystem per la programmazione del test vero e proprio, questa modalità di utilizzo non permette di ottenere le prestazioni richieste e gestisce bene solo sequenze non molto articolate.
Per questo motivo l’attività di preparazione dei test viene fatta attraverso java mentre JSystem si occupa di eseguire correttamente la sequenza di questi programmi e della documentazione dell'esito di test (rapporto di test). L'attività di automazione in E/// è di programmazione.

In JSystem possiamo individuare 3 parti fondamentali.



- a sinistra viene riportato l'elenco dei test da eseguire (ad ogni riga corrisponde, a grandi linee, un singolo test)
- a destra (per ogni test) viene presentato l'insieme di passi da eseguire per quel test (ogni riga a destra viene definita "Brick", e corrisponde a del codice Java)
    - L'elenco di tutti i Brick disponibili è già stato definito precedentemente, quindi per ogni test si deve solo selezionare l'elenco, e l'ordine, dei Brick desiderati
- Un file detto SUT (System Under Test) che descrive il test bench su cui viene eseguito il test

Il file SUT serve per evitare di cablare nel codice di ogni test valori e parametri strettamente legati al test bench (ad esempio, il numero identificativo della scheda presente sul nodo, o le porte configurate etc), rendendo il test troppo dipendente dal test bench. Sfruttando il SUT è quindi possibile definire test e brick indipendenti dal banco di prova: nel codice del brick è possibile recuperare informazioni dal SUT per settare, ad esempio, variabili, così come è possibile da gui di JSystem specificare il nome delle stesse proprietà come argomento dei singoli passi, il cui valore verra' poi letto dal SUT.

Il SUT descrive non solo il Nodo ma anche gli strumenti ad esso collegati (quali strumenti sono collegati a quali schede/porte, come etc): include quindi informazioni che sono deducibili/mappabili nei file cdb e altre info che ad oggi non hanno una rappresentazione formale (ma che seguono comunque naturalmente una sintassi e struttura definita informalmente nel gruppo di test). Il SUT (che è un file xml) non usa informazioni/termini presi dall'IM: all'interno di uno specifico tag del SUT sono presenti stringhe con il formato "campo=valore", indentate a formare una struttura, che vengono poi ricercate dal codice JAVA (quindi le stringhe "campo" seguono un lessico ad hoc, che è tuttavia in stretta relazione con l'IM e standardizzato per quanto concerne gli strumenti).

A grandi linee, le fasi per la creazione (ed esecuzione) del test sono:

1. In base alla descrizione del test case, il tester seleziona i bricks che servono per ottenere quanto richiesto
2. Il tester crea il SUT file che descrive il test bench (o seleziona un SUT file già eventualmente esistente)
3. Il tester crea e associa eventuali Proprietà di JSystem
4. Il test viene eseguito: allo start JSystem, grazie ad una serie di API o comandi TCL, configura il test bench seguendo le indicazioni trovate nel SUT (quindi riconfigura come indicato il Nodo/Nodi ed esegue una serie di comandi per configurare gli strumenti e le connessioni), ed esegue il test
5. A fine esecuzione JSystem crea automaticamente un report testuale in cui riporta i risultati dell'esecuzione dei vari brick (in formato testuale), e un risultato globale del test (Passed, Fail, Fail Exception)

In generale vengono eseguiti un insieme di test (Test suite), uno dopo l'altro automaticamente (creando quindi un elenco di test nella parte sinistra di JSystem), che vengono poi lanciati anche in notturna, visto che le Test suite possono anche impiegare ore (se non giorni).

Nelle righe a sinistra, così come nei brick, compaiono anche dei test/operazioni “di controllo d flusso” che servono al tester per gestire il controllo della test suite: ad esempio esistono “Test” il cui compito è resettare la configurazione di un Nodo (per riportarlo ad esempio ad uno stato iniziale dopo/prima di un test e far pulizia), oppure selezionare un diverso SUT (quindi cambiando la configurazione del Test bench per il test successivo). Di conseguenza, è importante notare che il risultato di un test può, a volte, dipendere in qualche modo dai test eseguiti in precedenza poiché questi potrebbero, ad esempio, aver creato uno stato sul Nodo che non era quello previsto dal test in esecuzione, o aver lasciato “problemi”.

Infine, è oggi in fase di sviluppo (ed inserimento dati) un database relazione che mantenga lo storico di tutte le esecuzioni di test automatici eseguite. Per ogni test viene quindi creato un record in cui vengono salvati i dati identificativi del test (che ha un nome e appartiene ad una macro area relativa ad una specifica tecnologia e ad una specifica area), l’esito del test, la durata, il test bench associato etc. Si sta ipotizzando di espandere il db includendo ad esempio non solo il risultato globale del test, ma anche i risultati parziali (quindi il risultato dell’esecuzione dei singoli brick): questi dati/scelte sono legati anche alle soluzioni e risultati delle tesi proposte, soprattutto 1 e 2.

1) DYNAMIC/ADAPTIVE TEST CASES (3 TESI TRIENNALI O 1 SPECIALISTICA)

I test automatici sono il supporto per permettere allo sviluppatore di conoscere il risultato del suo lavoro in tempi rapidi, avere una confidenza sulla bontà della modifica e un'idea anticipata sui possibili side-effects introdotti nel sistema.

L'obiettivo finale di questo lavoro di tesi è rendere l'esecuzione dei test automatici (test singoli e possibilmente test suite) non legata ad uno specifico test bench: ad oggi dato un test il tester crea manualmente il SUT che descrive il test bench su cui quel test verrà eseguito, ma se si volesse eseguire lo stesso test su un altro test bench si dovrebbe nuovamente creare manualmente il SUT, operazione che richiede tempo e conoscenza approfondita del test bench.

L'obiettivo principale della tesi è rendere questa operazione di "esecuzione test su altro test bench" praticamente automatica, evitando così che l'operatore umano perda tempo a ricreare il SUT e a modificare se necessario la parte sinistra.

La tesi si suddivide in vari passi:

1. Studio dell'IM, Cdb, SUT e API per capire la relazione fra i file e come le informazioni sono trasformate da un formato all'altro (estrapolando quindi ad esempio il mapping implicito fra stringhe nel SUT e gli elementi nel CDB).

2. Formalizzazione del test bench: si vuole dare una formalizzazione di quanto rappresentato nel SUT (creando quindi un SUT nuovo e più strutturato, sulla base di un modello formale), così da modellare anche i possibili strumenti e le relazioni/connessioni con i Nodi

3. Sviluppo di un tool in grado ricreare automaticamente il SUT a partire da un dato test bench (sfruttando quindi i tools già a disposizione che automaticamente sono in grado di creare il cdb di un Nodo e di capire come/a quali strumenti sia legato)

4. Integrazione e sviluppo del tool in grado di "spostare" effettivamente un test previsto per un test bench A su un altro test bench B (aggiornando/creando quindi il SUT file e modificando, se necessario, le parti di sinistra).
   In questo passaggio sarebbe utile inserire una sorta di "controllo di fattibilità", poiché l'operazione è possibile se e solo se i 2 SUT sono coerenti (ad esempio, se nel primo esistono 2 schede di un certo tipo, devono esistere anche nel secondo, altrimenti i test non possono essere effettivamente eseguiti). Estensione di questo controllo potrebbe

anche essere la ricerca, dato un insieme di SUT già esistenti, di quelli che potrebbero essere adatti alla sostituzione, magari con poche modifiche.

Questa tesi potrebbe essere svolta o così intera (quindi come tesi specialistica) o divisa in 3 tesi triennali (punti 1-2, 3, 4).

2) ADVANCED AUTOMATIC FIRST ANALISYS (SPECIALISTICA O VARIE TRIENNALI)

Al termine di una sessione di test è necessario ottenere un responso rapido a fronte di una grande mole di dati ricevuti: i risultati di una test suite eseguita ad esempio in notturna sono centinaia di record in cui possono comparire svariati problemi (fail). A seguito di questa mole di informazione è richiesta una funzione di filtro rispetto a chi si occupa del debug. Il compito del tester è individuare i test falliti di maggior importanza (ordinandoli in ordine di gravità) e decidere come operare di conseguenza, ad esempio rieseguendo il test per vedere se l'errore si riverifica, per poi segnalare agli sviluppatori eventuali problemi individuati dall'analisi dei test. Tuttavia la scelta di quale test fallito rieseguire/riverificare è molto difficile e time consuming, così come serve molta esperienza per capire quale possa essere la root cause di un errore, mentre sarebbe ad esempio molto utile aver già rieseguito i test falliti al primo tentativo durante la notte, recuperando così tempo utile.

Inoltre il problema riscontrato in un test può causare il fail di molti test successivi, che si potrebbero ad esempio direttamente non eseguire: un utente umano esperto è in grado di rendersi conto di questo tipo di problemi. JSystem non può/sa applicare ragionamenti di questo tipo, quindi l'esecuzione in notturna può generare un alto numero di fallimenti riconducibili ad una singola causa.

L'obiettivo è avere un tool di supporto che tramite ragionamento complesso (intelligenza artificiale) identifichi le root-cause di fail di test o relazioni complesse fra test, e che sia in grado a run time di modificare l'esecuzione di un insieme di test in base ai risultati, oppure di segnalare nel log riassuntivo questo tipo di problematiche, così da velocizzare notevolmente l'analisi dei risultati finali e migliorare allo stesso tempo, se possibile, i risultati stessi.
Ad esempio, come spiegato in precedenza, il risultato di un test potrebbe dipendere dallo stato del nodo su cui viene eseguito, o dai test eseguiti in precedenza: questo tool dovrebbe essere in grado, a fronte di un fail di un test, di capire se esistono relazioni con i test precedenti che potrebbero averlo influenzato, e ad esempio potrebbe rischedulare il test in coda agli altri resettando prima il Nodo. Un altro comportamento intelligente potrebbe essere quello di dedurre un problema ampio su un Nodo ed evitare di continuare ad eseguire test successivi che

quasi sicuramente darebbero risultato negativo: a questo modo si risparmierebbe tempo di esecuzione che potrebbe essere sfruttato per eseguire altri test, verificare altri errori, etc., migliorando la copertura dei test e la stessa analisi del problema.

La tesi si articola in varie parti:

1. Analisi e formalizzazione regole. Alcune azioni intraprese dall'utente umano a seguito di un fail su un test possono essere formalizzate e quindi applicate via software.
   In prima fase è necessario interagire con gli specialisti per capire queste regole e formalizzarle (ad esempio in un linguaggio logico, e con una sintassi coerente col modello informativo).
   Inoltre, le relazioni fra test dovute ad esempio a parti comuni sono nuovamente da evincere e formalizzare, così come l'azione da adottare per verificare l'errore.

2. Analizzando il log di JSystem e degli stessi brick, si potrebbe ricercare la root cause di un problema ampliando la ricerca nei log del Nodo, delle sue componenti e degli strumenti associati, così da migliorare la reportistica (includendo più informazioni reperite automaticamente e logicamente collegate, operazione che viene quindi evitata all'utente umano)e specializzando le informazioni di debug a corredo.

Ad oggi i log sono testuali, quindi sebbene la formalizzazione sopra citata usi possibilmente una sintassi associata all'IM, l'analisi e l'estrapolazione delle informazioni dai test e dai loro risultati deve essere fatta parsando i log. Ad esempio, sebbene si possa esplicitare una regola del tipo:

Se 5 test sugli allarmi sono falliti, allora salta i successivi test sugli allarmi

Per capire che un test è relativo ad un allarme, e che il fallimento riguarda proprio un allarme, bisogna guardare i log e la descrizione testuale del test case (procedura che naturalmente non è detto che funzioni perfettamente). La fase sotto riguarda la formalizzazione dei test case e quindi l'aggiunta di informazione semanticamente rilevante ai test case (e ai log), così da permettere di sviluppare regole più complesse e comportamenti più intelligenti nel software.

3. Per capire se i test sono in qualche modo simili/collegati è possibile guardare il SUT su cui operano ma sarebbe ancora più utile sapere a priori su che parti del modello informativo quel test andrà ad operare, o su che strumento.
   Per fare ciò bisognerebbe modificare/aggiungere la definizione del test case, ad esempio esplicitare in maniera formale le entità (non le istanze!) su cui opera il test: di conseguenza sarebbe possibile inferire relazioni fra test che ad oggi sono difficilmente deducibili automaticamente. Questa fase prevede quindi di creare un formalismo per

descrivere i test case, mettendoli al modello informativo o ancor meglio al modello della configurazione (se creato nella tesi 1), così da facilitare un ragionamento logico automatico migliore.
Inoltre, questa formalizzazione dei test case potrebbe apportare altri benefici generali, ad esempio permettendo di salvare nel database citato in precedenza informazioni più precise sui vari test case, che potrebbero poi facilitare una successiva analisi dei dati (tesi 6)

4. realizzato il punto 3, si dovrebbe naturalmente realizzare un tool che permetta di creare il test case il più semplicemente possibile, selezionando le entità coinvolte dai modelli a disposizione

5. Ipotizzando di aver formalizzato il test case e il suo contenuto, si potrebbe generare automaticamente il SUT: oggi il SUT, o meglio il test bench, viene creato manualmente dal tester che, conoscendo il test case (o la test suite), sa che servono tot Nodi, con tot schede e porte configurate in un certo modo etc, e di conseguenza crea fisicamente il test bench adatto all'esecuzione di quella test suite. Se i test case fossero formalizzati, e quindi fosse possibile specificare/dedurre di che cosa fisicamente c'è bisogno (tot Nodi, schede, apparati), sarebbe anche possibile dedurre (almeno parzialmente) il test bench minimo necessario (e quindi il relativo SUT).

Tutti i punti sopra indicati possono essere svolti singolarmente come tesi triennali (naturalmente però seguendo un ordine logico laddove esistono delle relazioni fra i punti).

3) AUTOMATIC TEST GENERATION (TRIENNALE?)

La continuous integration deve essere applicata alle features legacy così come alle nuove features in fase di sviluppo.

L'obiettivo finale è quello di rendere rapido lo sviluppo di test automatici delle nuove features, ad esempio generando automaticamente i test e i bricks utilizzando come input il modello informativo. Se si fosse già sviluppato il modello formale anche del test bench, come descritto nella tesi 1, si potrebbe usare anche quello come punto di partenza per lo sviluppo dei test automatici, creando quindi test automatici non solo a livello di Nodo ma anche a livello di configurazione.

1. Modellizzazione componenti del sut senza modello informativo (e.g. strumenti) (se non fatto precedentemente)
2. Generazione sistema di scrittura test case dal confronto dei modelli: sfruttando le modelizzazioni del test bench e i documenti con i requisiti richiesti, sviluppare un tool di supporto per la stesura dei test automatici, generando «scheletri» di test automatici da implementare.
3. Realizzare un tool che permetta di creare il test in modalita' automatica, selezionando le entità coinvolte dai modelli a disposizione.

3 A) NEGATIVE TEST (SE ASSOCIATA A TESI 3, DOPO, TRIENNALE)

Obiettivo di questo lavoro è valutare l'efficacia di una sessione di test negativi eseguita tramite test automatici. Il test "negativo" ("Negative Test") viene eseguito per determinare e verificare la risposta del sistema quando l'input non è quello atteso. Si possono valutare sia le risposte di errore già previste dal sistema sia cercare di generare input non previsti verificando così il comportamento dell'apparato in casi particolari. Partendo dalla tesi 3, si vuole creare automaticamente test negativi.

3 B) ANALISI E VALUTAZIONE DEI TOOL DI DOCUMENTAZIONE AUTOMATICA PER JAVA (TRIENNALE)

Obiettivo di questo lavoro è valutare l'efficacia dei vari tool di generazione automatica di documentazione, nel contesto delle suites di test automatico, in considerazione sia della documentazione di supporto per che si occupa dello sviluppo, sia della documentazione di supporto per chi si occupa del debug e della stesura di test suites.

4) TOOL PIANIFICAZIONE TEST BENCHES E MATERIALE (TRIENNALE)

Realizzare un tool che permetta, data la formalizzazione del test case e del SUT, di specificare/dedurre di che cosa fisicamente c'è bisogno (Nodi, schede, apparati).

Sarebbe quindi anche possibile dedurre (almeno parzialmente) il test bench minimo necessario (e quindi il relativo SUT) a partire da un test case o da una test suite.

6) ADVANCED TEST RESULTS ANALISYS

L'obiettivo è sfruttare l'intelligenza artificiale per analizzare la copertura di test effettuata e identificare le aree su cui concentrare le attività di regressione da effettuare fino a fine release. Lo scopo è identificare le aree su cui concentrare le attività di regressione considerando lo storico dei test effettuati finora e gli esiti, inseriti nel db sopra citato.

Si vorrebbero applicare algoritmi di analisi dei dati (ad esempio data Mining o ragionamenti logici) così da:

1. verificare ad esempio la copertura di tutte le features/aree nelle varie release
2. individuare aree che presentano problematiche ricorrenti
3. individuare eventuali relazioni fra errori
4. etc

Così da migliorare i test futuri/correnti, ad esempio focalizzandosi di più su una certa area, verificando alcune features e il loro comportamento a seconda della configurazione etc.