"Aggiornare il tool oppure il sorgente"

Un approccio innovativo per risolvere questo dilemma

Anders Holmberg
IAR Systems

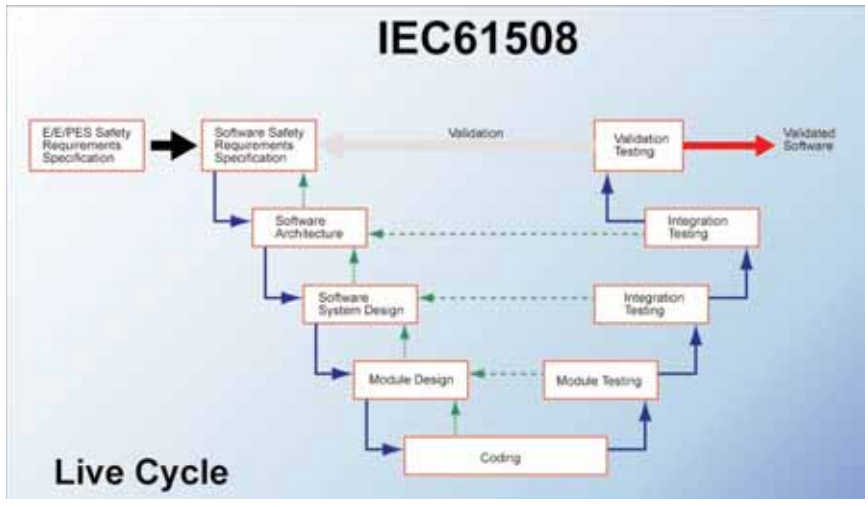

Fig. 1 – Flusso del modello a V previsto dal framework IEC 61508-3 relativo a progetti che devono garantire un'elevata integrità

Quando si utilizzano tool per la generazione del codice come ad esempio i compilatori (ovvero gli strumenti che traducono il codice sorgente scritto dal programmatore in linguaggio macchina, rendendo il programma eseguibile dall'elaboratore) può capitare che venga osservato un comportamento non corretto. Oppure, nel caso di programmi scritti in linguaggio C, può accadere di scoprire che il codice di produzione (production code) si basa su un comportamento non documentato o non definito. In una situazione di questo tipo l'utente si trova a dover scegliere tra due alternative: aggiornare il tool a una versione che corregga il problema oppure modificare il comportamento del codice sorgente per risolvere il problema legato al comportamento indesiderato. In questo articolo viene spiegato un approccio innovativo che permette di risolvere questo dilemma.

Gli effetti delle modifiche al software

L'apporto di modifiche al software nelle fasi finali di un progetto non è mai una scelta consigliabile. Sebbene la correzione dell'errore possa rivestire un'importanza cruciale per un utilizzo corretto e sicuro del prodotto finale, vi sono alcuni effetti secondari indesiderati che coinvolgono:

il processo: effettuare modifiche al codice e ricostruire l'immagine dell'applicazione costringerà di fatto a riavviare una o più attività di test o di controllo qualità (QA) del progetto. La minimizzazione di tali attività senza compromettere le caratteristiche di sicurezza e integrità in presenza di modifiche del codice può divenire un aspetto critico per il time-to-market.

In figura 1 viene riportato il flusso del modello a V come descritto dallo standard IEC 61508-3 che si occupa della sicurezza funzionale (in particolare la parte 3 prende in considerazione gli aspetti software) e la correlazione con le attività richieste dallo standard. Più tardi viene individuato un problema in un processo, più a monte si deve andare per il riesame di certe attività. Nel caso peggiore, potrebbe essere necessario prendere in considerazione una ricertificazione o una ri-validazione esterna;

il codice: la modifica del codice per correggere un comporta-

mento errato comporta sempre il rischio di introdurre un nuovo comportamento indesiderato. Per questa ragione talvolta si decide di non cercare di rimediare al problema che affligge il prodotto e di documentare con chiarezza l'impatto di questo sul comportamento del codice. I framework che definiscono le norme relative all'integrità spesso contribuiscono a rendere questo compito ancora più arduo, in quanto richiedono esaustive analisi dell'impatto delle modifiche prima dell'esecuzione delle stesse;

reputazione sul mercato: modifiche frequenti o di ampia portata negli stadi finali di un progetto può avere un impatto negativo sugli azionisti in relazione al successo commerciale del prodotto finale. Se il prodotto è già stato introdotto sul mercato la situazione può risultare ancora peggiore.

In definitiva, anche se non è sempre possibile evitare modifiche al codice, l'adozione di metodologie e tool per eliminare o quanto meno limitare l'impatto di queste modifiche può risultare estremamente utile. L'apporto di modifiche nelle ultime fasi del processo di sviluppo è imputabile essenzialmente alle ragioni qui di seguito esposte:

errore (bug) nel codice sorgente: esso è dovuto a errori o a interpretazioni errate da parte del programmatore nella fase di implementazione oppure a specifiche funzionali ambigue o incomplete che lasciano troppo spazio alla libera interpretazione. Sebbene questo sia un evento abbastanza comune, non verrà preso in considerazione nel resto dell'articolo;

errore latente nel codice non-ANSI/ C/C++: lo standard ANSI C presenta alcuni "lati oscuri" tali per cui il comportamento è definito dall'implementazione oppure risulta indefinito. Se alcune parti del codice sorgente sono implementate in modo da dipendere dal comportamento di un particolare compilatore in presenza di questi casi "patologici" (corner case) dello standard, in futuro è verosimile aspettarsi l'insorgere di un problema. Poiché questo genere di errore latente è un problema essenzialmente legato a problematiche di processo o di conoscenza, non sarà ulteriormente discusso.

A questo punto è possibile concentrare l'attenzione sull'oggetto di questo articolo, ovvero l'errore determinato dal tool per la generazione del codice oggetto. Quindi la trattazione sarà limitata agli errori nella toolchain formata da compilatore, assembler e linker.

Un esempio concreto

Si consideri la situazione che segue. L'errore individuato è frutto di un'ipotesi errata fatta dal compilatore circa l'allocazione nei registri e nello stack di variabili locali di una certa funzione. L'errore viene rivelato quando parecchie variabili competono per i registri della CPU disponibili e alcune variabili devono essere momentaneamente spostate nello stack. L'errore è stato riscontrato in una funzione corposa con un gran numero di calcoli aritmetici, ma non esiste alcuna garanzia che l'errore si manifesterà solamente in funzioni di notevoli dimensioni con un numero elevati di calcoli. A questo punto resta il dilemma se convincere il fornitore del compilatore a correggere l'errore (fix) o applicare soluzioni temporanee - che però non riparano l'errore stesso (workaround) – a tutta la base del codice (ovvero a tutto il codice sorgente usato per realizzare una determinata applicazione) con le implicazioni a livello progettuale poco sopra delineate.

Nel caso di progetti che richiedano un'elevata integrità la toolchain e il fornitore della stessa dovrebbero essere sottoposti a verifiche minuziose prima della scelta finale.

Solitamente, una volta scelti un particolare compilatore e una data versione, il loro utilizzo si protrae per tutta la durata del progetto. Alcuni framework per processi a elevata integrità richiedono che la scelta dei tool

sia soggetta a un processo formale che prevede la qualifica preventiva o la validazione in conformità a determinati criteri.

Un approccio innovativo

A questo punto ci si soffermerà su una tecnica particolare che può essere utilizzata nel caso esista una stretta cooperazione con il fornitore del compilatore. Si prenda in esame il caso di un compilatore per un'architettura a 32 bit. Parecchi core di CPU a 32 bit prevedono l'esecuzione delle istruzioni in modalità pipeline per incrementare le prestazioni: in pratica si tratta di dividere l'esecuzione di istruzioni complesse in sottoparti ciascuna delle quali richiede un ciclo di clock eseguito per essere eseguita all'interno del relativo stadio di pipeline (pipeline stage). In condizioni ideali, quindi, è possibile ottenere un throughput pari a una istruzione per ogni ciclo di clock. È molto frequente che questo processo ideale sia interrotto nel caso di istruzioni successive che competano per la medesima risorsa. Un classico esempio è rappresentato da due istruzioni successive la prima delle quali preveda la scrittura in un registro e quella successiva contempli la lettura del medesimo registro. In molte architetture pipeline una situazione di questo tipo provocherà il cosiddetto stallo della pipeline (pipeline stall): ciò significa che non è più possibile l'esecuzione di un'istruzione per ciclo in quanto la seconda istruzione resta in attesa che la prima istruzione finisca l'operazione di scrittura del registro. Un compilatore ideale per un'architettura di CPU di questo tipo cercherà di riordinare o pianificare le istruzioni in modo da ottimizzare la distanza tra le istruzioni che utilizzano le medesime risorse della CPU bloccando di fatto la corretta esecuzione delle pipeline.
Nell'effettuare tale riordinamento il compilatore deve sviluppare uno o più grafici di dipendenza (dependency graphs) per il blocco di istruzioni da pianificare al fine di valutare l'opportunità di spostare un'istruzione più avanti o indietro nel flusso di istruzioni. Il compilatore utilizza un insieme di funzioni per determinare se due istruzioni sono indipendenti, ovvero non utilizzano risorse in maniera conflittuale per cui il loro ordine può essere scambiato.
A questo punto si prende in considerazione la funzione che segue, che il compilatore potrebbe utilizzare per determinare l'indipendenza di due istruzioni MOV.

```
Bool AreIndependent(Instr inst1, Instr inst2)
{
  if (IndependentSourceAndDest(Inst1, Inst2))
  {
    return true;
  }
  else
  {
    return false;
  }
}
```

A prima vista sembrerebbe una funzione che non dia luogo ad alcun problema.
Essa fondamentalmente trasferisce la richiesta relativa all'indipendenza a una funzione di aiuto (helper) che determina se la sorgente e la destinazione delle istruzioni MOV siano usate in maniera indipendente.
Da parte del compilatore è corretto lasciare le istruzioni insieme nello stesso ordine.
Per cercare di risolvere il problema e ottenere le migliori prestazioni a volte si potrebbe creare un nuovo stallo della pipeline mediante lo spostamento di due istruzioni per evitare un altro stallo.
Si riprenda adesso in considerazione la funzione sopra riportata e il compilatore che utilizza tale funzione. Quando un utente effettua la compilazione di un determinato programma con questo compilatore il lavoro procede senza intoppi, a eccezione del fatto che rileva due operazioni di scrittura in memoria eseguite nell'ordine sbagliato, ovvero opposto a quanto specificato nel programma.
A questo punto dovrebbe intervenire lo scheduler che in questo risulta inadatto perché l'utente ha specificato che entrambe le variabili interessate dalle istruzioni MOV sono dichiarate come volatili, il che implica che l'ordine delle scritture non può cambiare – questo fatto assume un'importanza particolare nel caso in cui per esempio le operazioni di scrittura nella memoria siano destinate all'inizializzazione di qualche componente hardware esterno.
La funzione AreIndependent() ignora l'attributo di volatilità di entrambe le funzioni e comunica che è corretto procedere al riordinamento delle funzioni.
Come segnalato sopra, lo schedulatore può naturalmente scegliere di lasciare due istruzioni indipendenti al loro posto, ma per l'utente è facile osservare che esiste almeno una locazione influenzata da questo errore: la domanda da porsi è se esistano altre locazioni interessate.
La risposta a questa domanda equivale a controllare l'intera base del codice, ricercando gli accessi alle variabili volatili ed esaminando il codice generato: in definitiva si ritorna al nucleo centrale di questo articolo – ovvero come semplificare la gestione delle modifiche da parte dell'utente.
Un possibile rimedio per una situazione di questo tipo è il seguente: disporre di una versione specifica del compilatore originale che può cercare di identificare tutto il codice presente nella base del codice dell'utente che è effettivamente influenzata dall'errore.
Di seguito viene spiegato come un compilatore può essere trasformato in un rilevatore di errori.
La funzione prima riportata è utilizzata in un'altra funzione che modifica l'ordine di due istruzioni quando la medesima funzione ha stabilito che un'operazione di questo tipo ha effetti positivi.

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
  …
  if (AreIndependent(inst1, inst2))
  {
    ChangeOrderHelper(inst1, inst2);
  }
}
```

Questa funzione può essere modificata per rilevare il caso di errore, cioè quando la funzione ChangeMOVOrder() utilizza l'informazione errata per prendere una decisione. Il codice aggiunto riportato sotto (in rosso) va alla ricerca della situazione che genera la violazione e, nel momento in cui questa situazione si presenta, comunica le locazioni della sorgente interessata.

```
void ChangeMOVOrder(Instr inst1, Instr inst2)
{
  // Do other processing first
  …
  // Bug detection code
  if (IsVolatile(inst1) || IsVolatile(inst2)
  {
    ReportSourceStatement(inst1);
    ReportSourceStatement(inst2);
    return;
  }
  if (AreIndependent(inst1, inst2))
  {
    ChangeOrderHelper(inst1, inst2);
  }
}
```

Si faccia attenzione al fatto che il codice scritto in rosso è anche in grado di porre rimedio all'errore in quanto classifica come dipendenti tutte le istruzioni MOV con l'attributo di volatilità. Ma è di fondamentale importanza accertarsi che il codice di rilevamento non sia stato posizionato nella funzione affetta dall'errore. Nel caso si verifichi tale situazione, esso comunica ogni occorrenza di un'istruzione MOV potenzialmente errata. Anche questo esempio molto semplificato mette in evidenza uno dei possibili trabocchetti in cui si può incorrere nella creazione di un rilevatore di errori di qualità in produzione. Se può essere abbastanza semplice isolare la causa alla radice dell'errore, risulta molto più complicato determinare quando questo errore darà effettivamente luogo alla generazione di codice errato. Si potrebbe per esempio avere un certo numero di funzioni differenti di complessità variabile che dipendono dalla funzione AreIndependent(). Ma se risulta praticamente possibile creare un rivelatore di errori, esso può essere usato per determinare con precisione le esatte locazioni di ogni altro codice che è influenzato dall'errore originale. In tal modo non è più necessario esaminare tutto il codice oggetto in maniera manuale alla ricerca di possibili manifestazioni di questo problema.

IAR Systems (Microtask Embedded)
readerservice.it n.14