1 Verics

Cytat ze strony głównej projektu Verics:

> Verics (VerICS) jest naszym oryginalnym narzędziem przeznaczonym do automatycznej weryfikacji automatów czasowych oraz protokołów opisanych za pomocą pomocą podzbioru języka Estelle, opracowywanym w ramach projektu badawczego Automatyczna weryfikacja systemów zależnych od czasu realizowanego w Instytucie Podstaw Informatyki Polskiej Akademii Nauk.
>
> Wejściem dla Vericsa może być specyfikacja w Estelle, specyfikacja zapisana w naszym oryginalnym języku bazowym (IL), a także automaty czasowe przedstawione w postaci akceptowanej przez narzędzie Kronos. Verics umożliwia wykonanie translacji specyfikacji do postaci automatów czasowych, które przekazywane są następnie do kolejnych komponentów naszego narzędzia.
>
> [...]
>
> Verics umożliwia obecnie testowanie osiągalności. W celu wykazania, że własność bezpieczeństwa (safety) nie zachodzi, stosowana jest nowa bardzo efektywna metoda bazująca na translacji problemu osiągalności dla automatów czasowych do problemu spełnialności formuł zdaniowych (problemu SAT). Dowodzenie poprawności systemu wykonywane jest za pomocą budowania abstrakcyjnego modelu systemu przy równoczesnym testowaniu osiągalności w tym modelu stanu spełniającego żądaną własność. ...

2 Estelle

Estelle jest formalnie zdefiniowanym językiem służącym opisowi rozproszonych i współbieżnych systemów przetwarzających, w szczególności protokołów znajdujących się w warstwach OSI. Język bazuje na szeroko stosowanym i akceptowanym pomyśle komunikujących się ze sobą nie deterministycznych maszyn stanowych (*automatów*). Specyfikacja Estelle definiuje system hierarchicznych maszyn stanowych. Maszyny stanowe komunikują się poprzez wymianę komunikatów przez dupleksowe kanały komunikacyjne, którymi są połączone ich porty (*punkty interakcji*). Komunikaty są kolejkowane. Akcje, które zachodzą w maszynach stanowych (w *tranzycjach*) są zapisywane w postaci standardowego Pascala, w ten sposób Estelle może być używany przez bardzo szerokie grono programistów i naukowców.

Estelle pozwala na modelowanie synchronicznej oraz asynchronicznej komunikacji międzyprocesowej. Estelle pozwala również na dynamiczną ewolucję konfiguracji systemowej. Specyfikacja w Estelle może być zapisywana na różnych poziomach szczegółowości – od całkowicie abstrakcyjnego, na implementacji skończywszy.

2.1 Intro

Jak już było wspominano wcześniej, Estelle był opracowany na potrzeby opisów protokołów znajdujących się w stosie protokołów OSI. Ogólnie jednakże, jest on ogólną techniką specyfikacji systemów rozproszonych. Bazuje on na obserwacji, że oprogramowanie komunikacyjne zazwyczaj jest reprezentowane i implementowane jako model automatu skończonego.

Główną konsekwencją tego, że Estelle jest często wybierany jako język do modelowania automatu są oczywiste: architekci mają wygodny sposób zapisu formalnej specyfikacji modelu, programiści zaś mają natychmiastowy i oczywisty „szkielet” już gotowej implementacji konkretnego protokołu – wystarczy tylko dokonać odpowiednich „translacji” do języka docelowego.

2.2 Język

2.2.1 Specification

Specyfikacja w Estelle składa się z *modułów* komunikujących się ze sobą poprzez *kanały*.

2.2.2 Channel

Kanały w Estelle definiują jakiego typu *komunikaty* mogą być wysyłane przez każdą z komunikaujących się stron.
Przykład:

channel DaemonServer (User, Provider);
  by Provider:
    Bump;
channel GameServer (Player, Machine);
  by Player:
    Probe(x:integer, y:integer);
  by Machine:
    Win;
    Lose;

2.2.3 Module

Moduły w Estelle definiują interfejs modułów – sposób komunikacji z modułem. Nie definiują one sposobu obsługi komunikatu przez moduł.
Przykład:

module Game **activity**;
  ip
    P:GameServer (Machine) **individual queue**;
    D:DaemonServer (User) **common queue**;
end;

2.2.4 Body

Ciało modułu definiuje, w jaki sposób zachowuje się moduł o danym interfejsie. W tym miejscu zawarta jest logika, definicja automatu danego modułu.

2.2.5 Transition

Tranzycje w Estelle definiują przejścia w automacie pomiędzy stanami. Każda tranzycja posiada kilka dozorów:

- **when** specyfikuje punkt interakcji oraz komunikat, który jest oczekiwany,
- **from** specyfikuje stan, w jakim powinien się znaleźć automat,

- **to** specyfikuje stan, w jakim znajdzie się automat po wykonaniu tranzycji,
- **provided** specyfikuje warunki, które muszą zajść na zmiennych lokalnych modułu, aby tranzycja mogła być dozwolona,
- **delay** specyfikuje warunek czasowy w jakim tranzycja może zostać wykonana, musi to być tranzycja nieoczekująca (bez warunku **when**). **delay**(d_1, d_2) specyfikuje ile czasu musi minąć od momentu, aż wszystkie inne dozory są spełnione, aby tranzycja mogła zajść i od kiedy tranzycją musi zajść jeśli jest jedyną tranzycją dozwoloną w danym momencie.

2.3 Alternating Bit Protocol

```
specification ABtmout systemactivity;

default individual queue;
const tmout=2;

channel TR(M,A);
  by M:
    Msg(data: integer;
        bit: integer);
  by A:
    Ack(ack: integer);

module Sender activity;
  ip comm: TR(M);  {-- size=3 --}
end;
module Loosy activity;
  ip commT: TR(A); {-- size=2 --}
     commR: TR(M); {-- size=3 --}
end;
module Rec activity;
  ip comm: TR(A); {-- size=2 --}
end;

body SenderB for Sender;

  state S1, S2, S3;
  var m, s: integer;

  initialize to S2
      begin
        s := 0
      end;
  trans
    from S1 to S2
      begin
        { next message generating }
      end;
  trans
    from S2 to S3
       begin
          output comm.Msg(m,s);
       end;
  trans
```

```
    from S3 to S1 when comm.Ack
      provided (ack = s)
        begin
          s:=1-s;
        end;
  trans
    from S3 to S2 when comm.Ack
      provided ( ack <> s )
        begin end;
  trans
    from S3 to S2 delay(tmout)
        begin end;
 {++ prop
      ack: in S1;
      bit0: s==0;
  ++}

end; { SenderB }

body ReceiverB for Rec;

  state R1, R2, R3;
  var i,a,e: integer;

  initialize to R2
        begin
          e:=0
        end;

  trans
    from R1 to R2
        begin
          { message accepting}
        end;
  trans
    from R2 to R3 when comm.Msg
        begin
          i:=data;
          a:=bit;
          output comm.Ack(a);
        end;

  trans
    from R3 to R1 provided (a = e)
        begin
          e:=1-e;
        end;

  trans
    from R3 to R2 provided (a <> e)
        begin end;

  {++ prop
       ack: in R1;
       bit0: e==0;
  ++}
```

```
end; { ReceiverB }

body LoosyChanB for Loosy;

  state C1;

  initialize to C1
         begin end;
  trans
    from C1 to same when commT.Msg
         begin output commR.Msg(data,bit) end;
  trans
    from C1 to same when commT.Msg
         begin end;
  trans
    from C1 to same when commR.Ack
         begin output commT.Ack(ack) end;
  trans
    from C1 to same when commR.Ack
          begin end;

end; { LoosyChanB }

modvar
  S: Sender;
  C: Loosy;
  R: Rec;

    initialize
      begin
        init S with SenderB;
        init C with LoosyChanB;
        init R with ReceiverB;
        connect S.comm to C.commT;
        connect C.commR to R.comm;
       end;
end.
```

3 Translacja

W większości wypadków translacja nie sprawia problemów – Intermediate Language i Estelle mają bardzo dużo cech wspólnych. Model tranzycji i dozorów jest niezwykle zbliżony. Problemy powstają z zaawansowanym systemem typów Estelle, komunikatami Estelle, zagnieżdżeniem modułów, dynamicznymi instrukcjami oraz instrukcjami warunkowego wykonania i nawrotów.

3.1 Translacja typów

Każda zmienna w Estelle jest „rozpłaszczana” na klika zmiennych w IL'u.
Przykład (część definiująca ciało modułu):

```
body SenderB for Sender;

  state S1;
```

```
  type
    rec1=record
        a, b: integer;
    end;
    rec2=record
        field1: rec1;
        field2: integer
    end;


  var i: rec2;

  initialize to S1
      begin
      end;
  trans
    from S1 to SAME
      begin
         i.field1.a := i.field1.b + 1;
      end
end; { SenderB }
```

Jest tłumaczone na:

```
ex1{
  process SX_ {
  /* state declarations */
    stateset
      SX_X_S1;
  /* variable declarations */
    var int SX_iX_field2;
    var int SX_iX_field1X_b;
    var int SX_iX_field1X_a;
  /* transition section */
    to SX_X_S1: {
    }
    from SX_X_S1 to SX_X_S1: {
      SX_iX_field1X_a = SX_iX_field1X_b + 1;
    }
  prop {
  }  }/* for SX_ */

/* main section */
  main {
    SX_
  }
}
```

3.2 Translacja instrukcji warunkowych

Estelle zawiera większość elementów sterujących zapożyczonych z Pascal'a. Itermediate language w odróżnieniu nie ma żadnych instrukcji warunkowych ani warunkowego powtarzania. Jedyny mechanizm dostępny – to warunki dozorów oraz stany. Instrukcje warunkowe oraz pętle warunkowe muszą więc być „emulowane” za pomocą mechaniznów dostępnych w IL'u. Przykład (część definiująca ciało modułu):

```
body SenderB for Sender;
```

```
  state S1, S2;
  var i: integer;

  initialize to S1
      begin
        i := 1;
      end;
  trans
    from S1 to S2
      begin
        i := 2;
        if i = 1 then
                i := 0
        else
                i := 1;
        i := 3;
      end;
    from S2 to S1
        begin
        end
end; { SenderB }
```

Jest tłumaczone na:

```
ex2{
  var bool threading_mutex = false;

  process SX_ {
    stateset
      IFSTATE_finish_0,
      IFSTATE_start_0,
      SX_X_S2,
      SX_X_S1;

    var int SX_i;
  /* transition section */
    to SX_X_S1: {
      SX_i = 1;
    }
    from IFSTATE_finish_0 to SX_X_S2: {
      SX_i = 3;
      threading_mutex = false;
    }
    from IFSTATE_start_0 to IFSTATE_finish_0:
      when (
        !(SX_i == 1)) {
      SX_i = 1;
    }
    from IFSTATE_start_0 to IFSTATE_finish_0:
      when (
        SX_i == 1) {
      SX_i = 0;
    }
    from SX_X_S2 to SX_X_S1:
      when (
        (threading_mutex == false)) {
      threading_mutex = true;
```

```
      threading_mutex = false;
    }
    from SX_X_S1 to IFSTATE_start_0:
      when (
        (threading_mutex == false)) {
      threading_mutex = true;
      SX_i = 2;
    }
  prop {
  }  }

  main {
    SX_
  }
}
```

3.3 Translacja komunikatów zawierających typ Estelle

Mechanizm komunikacji w Estelle jest niezwykle rozbudowany – komunikaty mogą mieć różne nagłówki w tym samym kanale (*typ komunikatu*).

Każdy typ komunikatu może mieć własne parametry przekazywane. Intermediate Language jest niezwykle ubogi pod tym względem – jedyna możliwość komunikacji jest jednokierunkowy bufor, który przetrzymuje wartości konkretnego typu prymitywnego. Ze względu na to potrzebna jest metoda „serializacji" komunikatu Estelle do buforu Intermediate Language oraz sensowne zarządzanie buforami.

W tym miejscu nie upominamy tablic interakcji, które są jeszcze potężniejszym narzędziem dostępnym w Estelle. Tablicowe punkty interakcji są również tłumaczone, jednakże ze względu na statyczną strukturę specyfikacji w IL'u, nie jest możliwe tłumaczenie podłączeń punktu interakcji tablicowego do punktu interakcji tablicowego – w translowanej specyfikacji do IL nie jest wtedy możliwe statyczne dowiązanie buforów do siebie, co jest warunkiem niezbędnym. Przykład:

```
specification ex3 systemactivity;

channel TR(M,A);
  by M:
    Msg(data: rec);
  by A:
    Ack(i: integer);

type
  rec = record
      a, b: integer
  end;

module Sender activity;
  ip comm: TR(M);
end;

module Receiver activity;
  ip rec: TR(A);
end;

body SenderB for Sender;
```

```
  state S1;
  var m: rec;

  initialize to S1
      begin
        m.a := 0;
        m.b := 1;
      end;
  trans
    from S1 to SAME
      begin
        output comm.Msg(m);
      end;
end; { SenderB }

body ReceiverB for Receiver;

  state S1;
  initialize to S1
       begin
       end;
  trans
       from S1 to SAME
       when rec.Msg(msg)
       begin
         output rec.Ack(msg.a);
       end;
end; { ReceiverB }

modvar
  S: Sender;
  R: Receiver;

  initialize
      begin
        init S with SenderB;
        init R with ReceiverB;
        connect S.comm to R.rec;
       end;
end.
```

Jest tłumaczone na:

```
ex3{
/* Constant definitions */
  define TRX_MsgX_ 1;
  define TRX_AckX_ 0;

/* Buffer definitions */
  buffer int SX_commX_in;/* for RX_recX_out */
  buffer int SX_commX_out;/* for RX_recX_in */

/* Processes definitions */
  process RX_ {
    stateset
      RX_X_S1;
  /* variable declarations */
    var int TRX_MsgX_dataX_b;
```

```
    var int TRX_MsgX_dataX_a;
    var int X_X_aux;
  /* transition section */
    to RX_X_S1: {
    }
    from RX_X_S1 to RX_X_S1:
      when (
        get(SX_commX_out, X_X_aux) &&
        get(SX_commX_out, TRX_MsgX_dataX_a) &&
        get(SX_commX_out, TRX_MsgX_dataX_b) &&
        (X_X_aux == TRX_MsgX_)) {
      put(SX_commX_in, TRX_AckX_);
      put(SX_commX_in, TRX_MsgX_dataX_a);
    }
  prop {
  }  }/* for RX_ */
  process SX_ {
  /* state declarations */
    stateset
      SX_X_S1/* for SX_X_S1 */;
  /* variable declarations */
    var int SX_mX_b;
    var int SX_mX_a;
  /* transition section */
    to SX_X_S1: {
      SX_mX_a = 0;
      SX_mX_b = 1;
    }
    from SX_X_S1 to SX_X_S1: {
      put(SX_commX_out, TRX_MsgX_);
      put(SX_commX_out, SX_mX_a);
      put(SX_commX_out, SX_mX_b);
    }
  prop {
  }  }/* for SX_ */

/* main section */
  main {
    RX_ || SX_
  }
}
```

4 Sieć

http://www.ipipan.waw.pl/ penczek/abmpw