Modellera det perifera på Intels språk

Av Jakob Engblom, Intel

Jakob Engblom jobbar med att främja ekosystemet för Simics-simulatorn hos Intel i Stockholm. Han har jobbat med Simics och andra simulatorlösningar de senaste tjugo åren, först på startupbolaget Virtutech, sedan på Wind River, och nu på Intel.

Med Intels Device Modeling Language – som nu är öppen källkod – blir det enklare att skapa kortfattad och tydlig programkod för modeller av periferienheter till virtuella plattformar.

Intel har släppt modelleringsspråket DML (Device Modeling Language) med tillhörande kompilator som öppen källkod. DML används för att skapa periferienhetsmodeller till virtuella plattformar och har varit i bruk tillsammans med simulatorn Simics sedan 2005.

Vad är DML ämnat att göra?

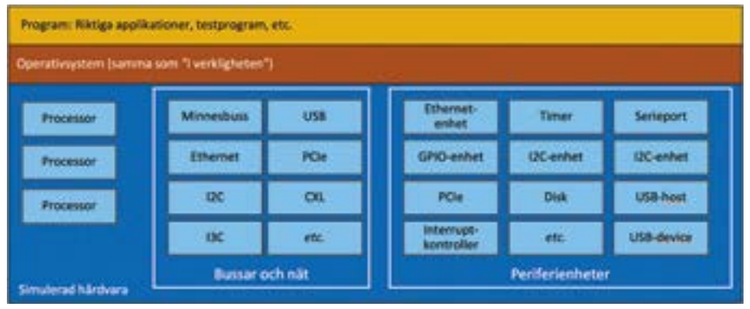
En virtuell plattform är en mjukvara som simulerar en hårdvara, så att man kan köra samma mjukvara som kör på hårdvaran. Den är väldigt användbar för att utveckla och testa mjukvara för hårdvara som är under utveckling. Den används också när hårdvaran är krånglig att sätta upp, dyr att köpa, eller allmänt besvärlig att jobba med.

På Intel använder vi virtuella plattformar för att kunna tidigarelägga mjukvaruutveckling för kommande hårdvaruprodukter (”shift left”). Dessutom används modeller för att designa hårdvara – simulatormodeller är de viktigaste verktygen för datorarkitekter, och har varit det sedan 1960-talet.

En virtuell plattform (se figur 1) består av ett antal modeller av hårdvaruenheter, vilka man grovt kan dela in i tre sorter: bussar eller nätverk som kopplar ihop saker, processorer som kör maskinkod, och periferienheter. De sistnämnda är väldigt varierande och innefattar allt från enkla lysdioder och räknare till diskar och nätverksenheter.

Det går typiskt flera hundra sorters periferienheter på varje processorvariant, så det mesta av arbetet med att ta fram virtuella plattformar går åt till modellering av periferienheter. Effektiva verktyg för modellering sparar väldigt mycket tid och gör det möjligt att tillhandahålla fungerande plattformar tidigare i en produkts livscykel.

Figur 2 visar de viktigaste koncepten hos en modell i en virtuell plattform.



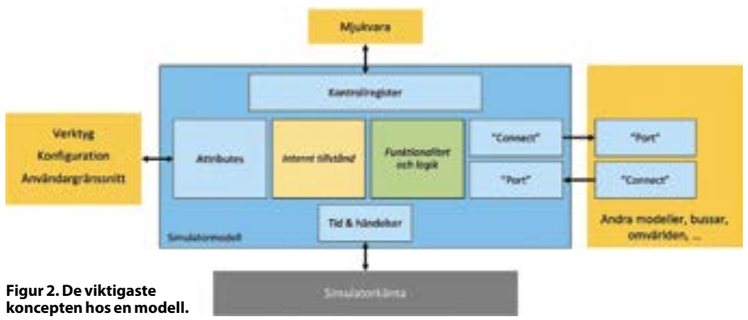
Figur 1. En virtuell plattform.

Traditionellt har modellering av perienheter gjorts genom att använda bibliotek i vanliga programmeringsspråk som C, C++, eller Python. Men det blir lätt rätt klumpigt. Strukturen hos ett generellt programmeringsspråk passar inte strukturen hos en hårdvarumodell. DML är istället ett så kallat domänspecifikt språk som låter programmeraren använda begrepp från hårdvarumodellering: registerbanker, kopplingar till andra modeller, hantering av tid och bakdörrar för användare och verktyg.

Figur 3 visar ett exempel på hur DML-kod ser ut. Notera att DML inte beskriver mikroarkitekturen hos hårdvaran, utan en modell av designen. En modell av själva mikroarkitekturen blir alldeles för detaljerad och därmed långsam att simulera.

DML har använts för modellering med simulatorn Simics sedan 2005. En första version kom 2005, sedan en mindre uppdatering 2007, och nu senast en rejäl omarbetning 2019.

Det är denna senaste version av DML, version 1.4, som nu utvecklas vidare inom ett open-source-projekt. Samma kod, kompilator och språkversion används av Simics-användare inom och utanför Intel.



Figur 2. De viktigaste koncepten hos en modell.

Hur förhåller sig DML till SystemC?

Man kan tycka att båda är ”språk” som används för att skapa virtuella plattformar, men i praktiken löser de olika problem. Grunden i SystemC är en simulatorkärna som sköter schemaläggning av simuleringstrådar, händelser, signaler, med mera. För att bygga modeller i SystemC behöver man någon slags modelleringsbibliotek som tillhandahåller register och liknande koncept. I princip kunde DML användas för att generera modeller som kan köras ovanpå en SystemC-kärna.

Vad är det som är så bra med DML?

Vår erfarenhet är att modeller går lättare att skriva. Koden blir kortare och tydligare och därmed enklare att underhålla. Modellerna blir generellt snabbare eftersom det är svårare att göra enkla misstag och koda på dåliga sätt.

DML-kompilatorn kan också automatiskt generera stöd för simulatorfinesser som checkpointing, ”attribut” för inspektion och ändring av tillståndet, metadata för register, loggning, brytpunkter och olika former av instrumentering.

En liten men trevlig fördel är att DML-koden är mestadels oberoende av simulatorkärnan, vilket gör det enklare att uppgradera modeller till nya versioner av simulatorn. Det räcker med att kompilera om DML-koden för att dra nytta av nya simulator-anrop och sluta använda sådana som tagits bort.

DML utvecklas kontinuerligt. Ändringar sker i själva språket, i kompilatorn, och i standardbiblioteket. Språket utökas för att göra det enklare att modellera, för att göra det enklare att skriva effektiva modeller, eller för att göra beteendet mer konsekvent. I ett språk under utveckling kommer det hela tiden upp nya små och stora saker man kan göra.

Ett exempel på en kraftfull men på ytan enkel utökning var införandet av nyckelordet `saved` för att markera variabler i en modell som ska sparas vid checkpointing. Nu kan man skriva så här:

```
saved uint32 i;
```

Tidigare var programmeraren tvungen att deklarera ett attribut för detta, vilket kunde kräva ganska många rader tråkig kod. Speciellt för strukturerade data.

Ett annat exempel på en bekvämlighet var införandet av en genväg för ett vanligt kodmönster där man skriver ut ett loggmeddelande en gång på en låg loggnivå så att det syns. Men framtida utskrifter sker bara om man höjer loggnivån, för att undvika att upprepa samma information gång på gång.

```
register example @ 0x10 is write
”Demonstrate then syntax” {
  method write(uint64 v) {
          log info  , 1:
”Logging every time”;
          log unimpl, 1 then 2:
”I will log this at level 1 only once”;
  }
}
```

Mycket av det som ser ut att vara inbyggt i språket är i själva verket delar av standardbiblioteket, vilket gör det betydligt enklare att ändra. Det finns många mallar (templates) som definierar beteendet för register, attribut, et cetera, och dessa kan justeras utan att språket eller kompilatorn behöver ändras. Nya mallar har införts för att tillhandahålla nya funktioner ”i språket” utan att egentligen ändra på det. ■

HISTORIK

Simulatorn Simics, som DML används tillsammans med, är sprungen ur forskningsinstitutet SiCS, numera RiSE. Den första koden skrevs år 1991. År 1998 grundades företaget Virtutech för att kommersialisera tekniken. Intel köpte Virtutech år 2010, och sedan dess har simulatorn utvecklats vidare inom Intel. Stockholmskontoret är fortfarande centrum för utveckling av tekniken. Idag sker all utveckling av DML på github, och en kompilerad version ingår i Intels Simics-simulatorpaket.

DML **Simics**

Figur 3. Här är ett utdrag ur en modell skriven i DML. Den kompletta koden finns i exemplet workshop-02 i den publika utgåvan av Simics-simulatorn.

```
dml 1.4;
device m_control;
param desc = "mandelbrot control unit";
param documentation = "Control unit to sync multiple compute units.";
import "utility.dml";
import "simics/devs/signal.dml";
import "simics/devs/memory-space.dml";
import "simics/simulator-api.dml";
[...]
param max_compute_units = 8;
param stall_on_status_read = false;
#if (stall_on_status_read) {
   // attribute uses two standard templates to build the behavior
   attribute status_reg_stall_time is (double_attr, init) {
      param documentation = "Time to stall on status register read(s)";
      param configuration = "optional";
      param init_val = 50.0e-6; // 50 ms or about 5000 pixels
   }
}
[...]
connect register_memory { // Connections to other objects
   param desc = "Memory space for the on-accelerator registers";
   param configuration = "required";
   interface memory_space;
}
[...]
// Templates break out common declarations or functionality
template compute_unit_bitmask_register {
   field reserved @ [63:max_compute_units] is (zeros);
   field unit[i<max_compute_units] @ [i] is (read_only);
}
[...]
bank ctrl {   // Register bank: Declaring the layout
   param register_size = 8;
   register compute_units @ 0x00 is (read_only) "Available units";
   register start      @ 0x08 is (write)      "Start operation";
   register status     @ 0x10 "Operation status" {
      field done       @ [63] is (write)      "Operation completed";
      field processing @ [62] is (read_only)  "Op in progress";
      field unused     @ [61:0] is (reserved) "unused";
   }
   register _reserved  @ 0x18 is (reserved) "Reserved";
   // Managing the compute units as bit masks in registers
   register present @ 0x20 is (compute_unit_bitmask_register)
                      "compute units present bitmask";
   register used    @ 0x28 is (compute_unit_bitmask_register)
                      "units used in current operation";
   register done    @ 0x30 is (compute_unit_bitmask_register)
                      "units used & completed operation";
}
[...]
bank ctrl {  // Adding in some behavior to a register
   [...]
   register start is write {
      method write(uint64 value) {
         start_compute(value);   // Call method defined elsewhere
         this.val = value;
      }
   }
   [...]
}
```