Referat

Formell verifiering för modellbaserad utveckling
Karin Tejler

Vid all form av produktutveckling är validering och verifiering viktiga moment för att skapa en väl fungerande produkt. Utvecklingens allt högre takt sätter stor tidspress på valideringsoch verifieringsmomentet. Den mest använda metoden för validering är testning, som även om den till stor del kan automatiseras är väldigt tidskrävande. Dessutom är den inte alltid helt tillförlitlig. Formell verifiering är, inom industrin, ett nytt tillvägagångssätt för att snabbare och lättare finna fel i system. Detta examensarbete fokuserar på formell verifikation för modellbaserad utveckling i form av modellcheckning. Två verktyg, Simulink Design Verifier och Reactis Validator, har undersökts och jämförts. Det första var ett modellcheckningsverktyg och det andra ett testningsverktyg. En Simulinkmodell för en adaptiv farthållare (AiCC), utvecklad på Scania, har använts för implementationen. Även teorin bakom de båda metoderna har studerats.

Slutsatser som kan dras efter examensarbetets slutförande är att formell verifikation definitivt är en metod för framtiden som antagligen kommer att kunna underlätta verifieringssteget och minska den tid som måste läggas ner på verifieringen. Om utvecklingen fortsätter kommer sannolikt formell verifiering även leda till mer tillförlitliga produkter. Dock är det verktyg som studerats och utvärderats under detta examensarbete inte i nuläget moget för att användas. Eftersom endast ett verktyg för formell verifikation har undersökts under detta examensarbete kan inga vidare slutsatser dras om formell verifikation som metod utan ytterligare undersökning av verktyg skulle behövas för att detta. Formell verifiering är troligen inget som kan ersätta testning på Scania, i alla fall inte inom den närmaste framtiden. Det bästa vore troligen att kombinera de två olika teknikerna och låta dem komplettera varandra. Detta skulle nog leda till det absolut bästa resultatet och därigenom mer tillförlitliga system.

Nyckelord: formell verifiering, testning, modellbaserad utveckling, modellcheckning, Simulink Design Verifier och Reactis Validator.

Institutionen för Informationsteknologi, Uppsala Universitet, BOX 337, SE-751 05 Uppsala

Abstract

Formal verification for model-based development
Karin Tejler

Validation and verification is a very important process in product development. This is because it is essential to make sure a product works in a correctly manner. Due to the rapid increase in development, pressure is consequently put on the industry to speed up validation and verification in order to get a product released on the market at quickly as possible. The most frequently used method for validation is testing, which even though it is partly automatic is very time consuming. Also testing is never 100 % accurate. Within the industry formal verification is a new approach to finding bugs and making sure systems behave as it is intended. Formal verification strives to higher the accuracy of verification and validation and also lets users spend less time and effort on this process. This Master thesis will discuss formal verification for model based development with a focus on model checking. Two different tools have been studied, evaluated and compared, Simulink Design Verifier which is a model checker, and Reactis Validator which is a tool for test vector generation. A model for an adaptive cruise control (AiCC) designed at Scania in Simulink was used for the implementation. The theory behind the two different methods was also studied.

After finishing this master thesis conclusions can be drawn that model checking definitely is a method for the future. However the tool studied in this project is not good enough to be used today. The future of model checking relies on the development of more sophisticated tools that have the ability to check complex models and still be easy to use. It is impossible to draw any further conclusions about formal verification as a whole since only one model checking tool has been used and evaluated. However a large effort must be put down in order to keep developing high-quality tools that can take model checking to the next level. One thing that is for sure is that, at the time of writing, SDV is not worth spending your money on, it needs to be further developed before it can be considered a complete tool. In the near future formal verification is not something that can completely replace testing at Scania. Newer and better tools must be studied and evaluated in order to enhance the knowledge of formal verification at Scania before the method can be incorporated into the development process. The best way to start would probably be to combine testing and formal verification and let them work together. This would almost certainly lead to the best result and through that more reliable products.

Key words: formal verification, testing, model-based development, model checking, Simulink Design Verifier and Reactis Validator.

Department of Information Technology, Uppsala University, SE-151 05 Uppsala
ISSN 1401-5765

Förord

Detta examensarbete har utförts inom civilingenjörsprogrammet i miljö- och vattenteknik vid Uppsala Universitet och omfattar 30hp. Examensarbetets är utfört på uppdrag av Scania och handledare har varit Samuel Malinen. Ämnesgranskare har varit Lars-Henrik Eriksson vid institutionen för informationsteknologi vid Uppsala Universitet.

Jag vill börja med att tacka Samuel Malinen för mycket givande handledning och uppmuntran under arbetets gång och Lars-Henrik Eriksson för värdefulla kommentarer på rapporten. Vidare vill jag tacka alla på RESP, RESK OCH RESV för hjälp, sällskap och mycket trevligt bemötande.

Karin Tejler

Solna, augusti 2008

UPTEC W08 024, ISSN 1401-5765
Tryckt hos Institutionen för geovetenskaper, Geotryckeriet, Uppsala universitet, Uppsala, 2008.

Populärvetenskaplig sammanfattning

Validering och verifiering är den process som alla produkter och system som utvecklas måste genomgå för att försäkra att dessa fungerar som de ska. System- och produktutveckling har ökat markant under senare år, och ökningen fortsätter i rask takt. Utvecklingen sker inte bara snabbare än tidigare, utan systemen blir även allt mer invecklade. Den accelererande utvecklingen sätter i sin tur press på företagen att lansera produkterna på marknaden så fort som möjligt. Detta, och även systemens högre grad av komplexitet, har i flera fall lett till att produkter har marknadsförts utan att adekvat validering och verifiering. Produkter som lanseras på marknaden och som innehåller fel leder till stora kostnader för företagen. I värsta fall kan det vara samhällsfarligt. Tillräcklig verifiering och validering av säkerhetskritiska produkter är sålunda av största vikt.

Det vanligaste tillvägagångssättet för validering är testning. Testning är ett systematiskt sätt att undersöka hur korrekt en modell av ett system är. Testning utförs oftast genom att användaren genomför experiment med modellen i form av simuleringar. Detta kallas dynamisk testning. Testningen utförs i en kontrollerad miljö, där det är känt hur systemet ska reagera. Testning kan även utföras i form av kodgranskning och dylikt. Då kallas det statisk testning. Testning är bra på många sätt men är väldigt tidskrävande. Dessutom är det omöjligt att vara helt säker på att det testade systemet är felfritt. Detta beror på att det i verkligheten aldrig är möjligt att undersöka alla möjliga simuleringssekvenser, d v s alla sätt ett system kan reagera på.

Eftersom denna valideringsprocess är så tidskrävande blir det ofta en flaskhals för utvecklingen. Företagen strävar därför efter nya och snabbare metoder för att förbättra detta förlopp. Formell verifiering är en metod som har ökat inom industrin under de senaste åren. Formell verifiering är en matematisk verifiering, vilket innebär att metoden använder sig av logiska uttryck för att bevisa att en modell är helt korrekt och avsikten med metoden är således att bevisa att en modell inte innehåller några felaktigheter.

Formell verifiering är egentligen ingenting nytt, men inom industrin har det tidigare inte använts i särskilt stor utsträckning. Dock har metoden studerats och utvecklats inom universitetsvärlden under en längre tid, men har tidigare ansetts som för komplicerad för industrin. Den finns flera olika sorters formell verifiering, dock får ekvivalenskontroll och modellcheckning anses som de vanligaste. Modellcheckning är den metod av formell verifiering som har ökat mest i industrivärlden under de senaste åren. Detta trots att ekvivalenskontroll egentligen är den enklaste varianten. Således har fokus i detta examensarbete varit just modellcheckning.

Modellcheckning är en algoritmisk metod för att verifiera att en modell av ett system uppträder som avsett. Metoden är framförallt effektiv när det handlar om att verifiera simultana system, det vill säga system som är uppbyggda av flera parallella processer som kommunicerar med varandra. Modellkontrollen utförs mer eller mindre automatiskt med hjälp av olika verktyg och består i princip av tre olika steg: modellering, specificering och verifiering. Modellering är det först steget i modellcheckningsprocessen. Under modelleringen konverteras systemdesignen som ska verifieras till en modell uttryckt i ett matematiskt språk som verifieringsverktyget kan förstå. Det finns ett flertal olika modelleringsspråk och i princip använder de flest verktyg sitt eget språk. För att undersöka om en systemmodell är korrekt måste användaren beskriva ett flertal krav som modellen måst uppfylla. Detta görs i specificeringssteget. Det finns två olika sorters krav som kan ställas på

en modell, dels sådan som är generella och gäller för alla olika modeller, dels sådana som är modellspecifika och måste utformas separat för olika modeller. Generella krav kan undersökas automatiskt av en modellcheckare, medan modellspecifika krav måste specificeras manuellt. Kraven måste i sin tur uttryckas i ett specifikt språk. Även i detta fall finns en mängd olika språk, men samtliga är baserade på matematisk logik. Verifieringen är sista och lättaste steget i modellcheckningen och sker mer eller mindre automatiskt. Under verifieringen söker den använda modellcheckaren igenom hela modellen för att leta efter fel, d v s uppträdanden som går emot något av de krav som beskrivits i specificeringen. Verktyget gör en komplett sökning och kontrollerar alla möjliga sätt systemet kan uppträda på. Det är detta totala igenomsökande som är den stora skillnaden mot testning.

Som redan nämnts är modellcheckning väldigt komplext och teoretiskt. Det är bland annat detta som har lett till att den inte slagit igenom utanför universitetsvärlden. En stor svårighet är användaren måste behärska flera olika språk.

I detta examensarbete har två olika verktyg, modellcheckaren Simulink Design Verifier och testningsverktyget Reactis Validator, använts för att jämföra testning och modellcheckning, även för att utvärdera verktygen. Vid implementeringen användes en modell för en adaptiv farthållare (AiCC) utvecklad på Scania och designad i Simulink/Stateflow. Inget av de undersökta verktygen visade sig vara särskilt bra. Simulink Design Verifier var väldigt lättanvänt, men allt för fyrkantigt och dåligt utvecklat för att kunna rekommenderas. Reactis Validator var bättre utvecklat, men dock mycket minde användarvänligt. Inget av verktygen klarade av att hantera samtliga krav i specificeringen.

Modellcheckning är definitivt en metod för framtiden, men det verktyg som undersökts vid detta examensarbete är inte tillräckligt utvecklat för at kunna användas i praktiken. Dock kan inga vidare slutsatser dras vad gäller formell verifiering som metod. För detta behövs studier och utvärderingar av flertalet verktyg. Modellcheckning är ingenting som i nuläget bör ersätta sedvanlig testning på Scania, utan borde istället ses som ett komplement till metoden. Testning och modellcheckning kommer sannolikt alltid att vara bäst för olika sorters krav och bör därför också användas för olika saker.

En rekommendation är att undersöka flera olika verktyg för modellcheckning och även följa framtida utveckling. Detta är möjligen något som skulle kunna göras i ytterligare examensarbeten.

Innehållsförteckning

1 Inledning

1.1 Bakgrund

En traditionell farthållare kan kontrollera att ett fordon håller en viss konstant hastighet. Detta är ett välanvänt system, men vid tät trafik är det oanvändbart. Adaptiv farthållning (AiCC) är ett system som anpassar fordonets hastighet till framförvarande fordon. Systemet utnyttjar en radarsensor för att få information om trafiksituationen och styrs med hjälp av motor, retarder och färdbromsar.

På Scania är en nyare version av den adaptiva farthållaren under utveckling. Syftet med det nya systemet, kallat AiCC, är att förbättra det befintliga systemet, men kommer även att innehålla nya delsystem och funktioner.

Utvecklingen av AiCCn har skett i den modellbaserade utvecklingsmiljön Simulink/Stateflow. Användningen av modellbaserad utveckling har kraftigt ökat under de senaste åren, och många uppenbara fördelar finns. Dels minskar råprogrammeringen, dels blir det enklare att verifiera och validera designen under utvecklingens gång.

Den enklare verifiering som modellbaserad utveckling har lett till betyder att nya verifieringsmetoder har blivit intressanta för industrin. Formell verifiering är en metod som har ökat markant inom den modellbaserade utvecklingen hos många industrier. Framför allt har denna metod visats sig intressant när det gäller att bevisa korrekthet hos inbyggda säkerhetskritiska system (Bouali, 2005). Inom all produktutveckling är validering och verifiering en viktig och stor del av arbetet, eftersom att det är nödvändigt att veta att en produkt håller vad den lovar när den väl lanseras på marknaden. Om en bils säkerhetsbälte eller krockkudde inte fungerar som avsett vid en krock, eller aktiveras när de inte ska, kan detta få fatala konsekvenser.

1.2 Examensarbetets syfte

Syftet med detta examensarbete är att:

- Undersöka teorin bakom formell verifiering och även delvis teorin bakom testning och testtäckning.
- Undersöka befintliga kommersiella verktyg för formell verifiering av modeller implementerade i Simulink/Stateflow och för formell verifierig av C-kod.
- Utföra formell verifikation med hjälp av Simulink Design Verifier på befintlig AiCC-modell.
- Utföra testning med hjälp av Reactis Validator på befintlig AiCC-modell.
- Jämföra de två ovan nämnda verktygen.
- Jämföra formell verifiering och testning.
- Utreda hur formell verifikation skulle kunna passa in i Scanias utvecklingsprocess.

2 Formell verifiering - teori

Syftet med formell verifiering är att undvika att fel uppkommer tidigt under utvecklingen och även att hitta de fel som likväl uppkommer så tidigt som möjligt (Nilsson, 2002). En av anledningarna till att formell verifiering utnyttjas allt mer är att man med matematiska bevis kan undersöka alla möjliga kombinationer och uppträdanden hos ett system. Dessutom kan vissa former av formell verifikation helt eller delvis automatiseras, vilket sparar både tid och pengar (Ouimet, 2005). Vanligtvis används testning/simulering som valideringsform. Denna metod har visserligen också utvecklats under åren men det är inte möjligt, ur tid och kostnadsperspektiv, att simulera alla möjliga uppträdanden hos ett system. Precis som inom de flesta kunskapsområden finns olika uppfattningar om det exakta syftet med formell verifiering. Vissa anser att meningen med formell verifiering är att användaren ska kunna vara 100 procent säker på att det inte finns några fel i systemet (Bouali, 2005), medan andra anser att syftet är att finna fel, precis som vid testning. Dock kan sägas att det är omöjligt att vara helt säker på att ett system fungerar oklanderligt i verkligheten, detta eftersom att inte hela utvecklingsprocessen är formell (Eriksson, 2008), men formell verifiering tycks vara ett steg i rätt riktning mot detta mål.

Det finns ett flertal olika former av formell verifiering, men inom industriverksamheten dominerar två former, nämligen ekvivalenskontroll och modellcheckning. Ekvivalenskontroll är den metod som anses enklast och är därför den metod som har använts mest. Ekvivalenskontroll innebär att man med hjälp av matematiska metoder jämför om två representationer av samma konstruktion har ett identiskt beteende (Nilsson, 2002). Likväl är det inte ekvivalenskontroll, utan modellcheckning, som numer används mest inom industrin. Modellcheckning är betydligt mer avancerat än ekvivalenskontroll. Användaren kan med hjälp av dessa verktyg specificera önskade krav på sin modell och sedan bevisa att dessa krav håller. Trots olika uppfattningar verkar modellcheckning vara den metod som under de kommande åren kommer att bli mest använd (Bouali, 2005).

2.1 Modellcheckning

Modellcheckning är en algoritmisk metod för att verifiera att en modell uppträder som avsett. Den är en metod som är effektiv framförallt när det gäller simultana system, alltså system som är uppbyggda av multipla processer som kommunicerar med varandra (Huth, 2004). Verifieringen sker genom att verktyget utför en jämförelse med en av användaren skriven kravspecifikation.

Modellcheckningen består i stort sätt av tre delar: modellering, specificering och verifiering (Huth, 2004).

2.2 Modellering

Modellering är det första steget i modellcheckningsprocessen. Designen som ska verifieras konverteras under modelleringen till en modell uttryckt i ett matematiskt språk som verifieringsverktyget kan förstå. Det finns en mängd olika språk som kan användas för att beskriva en modell. I princip har de flesta verktyg sitt eget språk, men en majoritet av dem är en variant av en ändlig tillståndsmaskin, FSM (Finite state machine) (Palshikar, 2004).

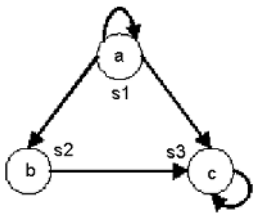
2.2.1 Ändlig tillståndsmaskin (FSM)

En FSM är en modell av beräkningar som är uppbyggd av en mängd tillstånd: ett begynnelsetillstånd, ett inmatningsspråk och en övergångsfunktion som beskriver övergångar från ett tillstånd till ett annat (Black, 1998).

2.2.2 Kripkestruktur

De flesta verktyg för modellcheckning skapar en Kripkestruktur, eller tillståndsövergångsstruktur M (S,R,L), se figur 1, av den modell som ska verifieras (Palshikar, 2004).

S = ändligt set av tillstånd $\{s_1, s_2, \ldots s_n\}$
R = övergångsrelation
L = mängd med beteckningar på tillstånd så att L (s) = g om tillståndet s har egenskapen g. I detta fall L (s) = {a, b, c} eller L (s1) = a.



Figur 1 *Kripkestruktur.*

Alla egenskaper ska utgöras av frågor och måste vara uppbyggda av atomära, eller grundegenskaper, d v s egenskaper som inte kan brytas ner till enklare frågeställningar. Dessutom ska frågorna kunna besvaras med sant eller falskt. Ett logiskt uttryck har ett eget sanningsvärde som beror av sanningsvärdena för de begynnande grundegenskaperna vid samtliga tidpunkter. En speciell uppsättning av sanningsvärden för grundegenskaperna kallas tolkningar. Kripkestrukturen är således ett sätt att representera dessa tolkningar. Modellcheckningen undersöker alltså om en viss Kripkestruktur representerar en logisk modell av specifikationen (Eriksson, 2008).

Strukturen är i stora drag uppbyggd av noder, som representerar de tillstånd som systemet kan nå, och bågar, som representerar övergångar från ett tillstånd till ett annat. En övergång i strukturen innebär en förändring av värdet på en eller flera tillståndsvariabler (Palshikar, 2004).

2.3 Specificering

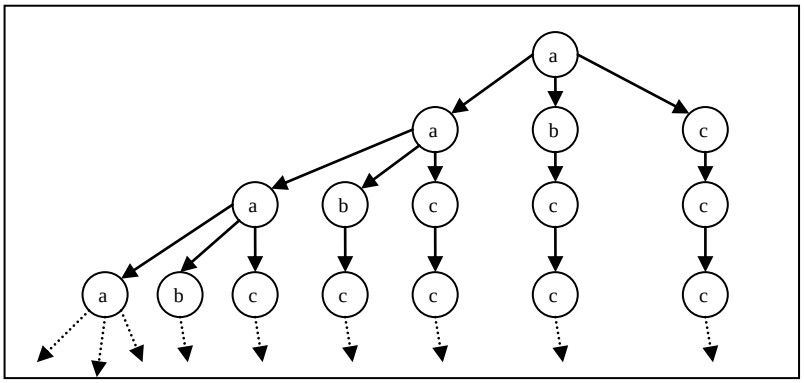
Andra steget i modellcheckningsprocessen är att skapa en kravspecifikation, som innehåller de krav som systemet måste uppfylla. Kraven eller frågorna som ställs kan delas in i två olika grupper. Den första innehåller generella frågor som kan appliceras på alla modeller, t ex om alla komponenter eller tillstånd är nåbara. Den andra gruppen är frågor som är modellspecifika och måste utformas manuellt (Ranville, 2004).

Vid specificeringen beskrivs tillståndsövergångar med utgångspunkt i Kripkestrukturen, se figur 1.

Språken som används för att formellt skriva kravspecifikationen kan variera, men oftast används ett språk uppbyggt av temporal logik. Med temporal logik menas att en egenskap, hos till exempel en modell, inte alltid är rätt eller alltid fel, utan kan variera beroende på hur systemet beter sig över tiden. Temporal logik är alltså en dynamisk notation (Huth, 2004). Det finns en mängd olika versioner av temporala språk, som används för att beskriva olika systemuppträdanden. Två av de vanligaste idag är *Computation Tree Logic* (CTL) och *Linear Time Temporal Logic (LTL)* (Zheng, 2007).

2.3.1 Computation Tree Logic (CTL)

CTL anser tiden som icke-linjär och använder sig av förgreningar. Detta betyder att vid varje gren kan ett antal olika tillstånd nås (Zheng, 2007). Vid användandet av CTL nystas tillståndstransformationsstrukturen upp och bildar en struktur i form av ett logiskt träd. Vid utveckling av strukturen i figur 1, med avseende på tillstånd s_1, fås följande träd, se figur 2.



Figur 2 *Oändligt beräkningsträd men avseende på tillstånd s_1.*

Roten på trädet representerar ett valt initialtillstånd, i det här fallet s_1, och de s k barnen till varje givet tillstånd representerar de möjliga nästkommande tillstånden. Varje systemexekvering är en väg genom detta träd (Palshikar, 2004).

För att kunna beskriva olika vägar i förgreningsstrukturen används följande vägbeskrivare, se tabell 1. (Bérard, 2001):

Tabell 1 *Notation som används för att beskriva vägar genom ett beräkningsträd i CTL.*

Notation	Betydelse
A	För alla sekvenser
E	För vissa vägar, det finns en väg

Följande temporala operatorer används för att beskriva egenskaperna av en väg genom trädet, se tabell 2.(Bérard, 2001):

Tabell 2 *Temporala operatorer som används för att beskriva vägar genom ett beräkningsträd i CTL.*

Temporal operator	Betydelse
X	Nästa steg, nästa tillstånd
F	Någon gång, tillslut
G	Alltid, globalt
U	Tills

Från det logiska trädet i figur 2 kan man t ex få följande tabell:

Tabell 3 *Exempel på logiska uttryck i CTL.*

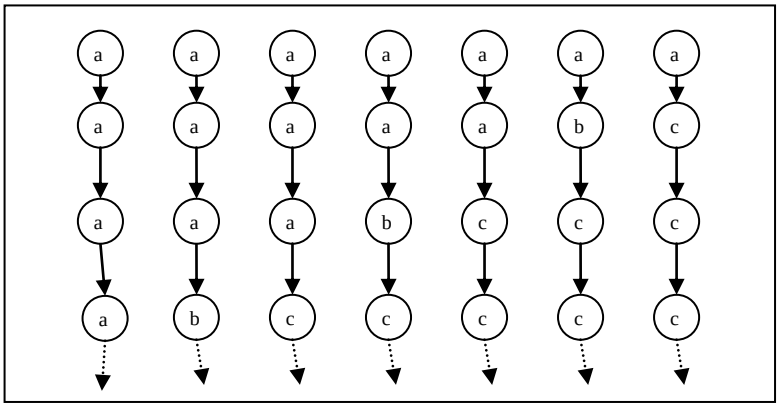
Notation	Förklaring	Sant/Falskt
EG (a)	Det finns en väg där a är sann i varje tillstånd.	sant
E (a U c)	Det finns en väg där tillstånd c någonstans är sann och där a är sann i samtliga tidigare tillstånd.	sant
AF (b)	Det finns en väg där b är sann vid varje tillstånd.	falskt

Den temporala logiken kan även kombineras med booleska operatorer som t ex:

¬ , ~, !	INTE (NOT)
∧, &&	OCH (AND)
∨, ||	ELLER (OR)
→	IMPLIKATION (IMPLIES)

2.3.2 Linear Time Temporal Logic (LTL)

LTL är en annan viktig form av temporal logik som vissa verifieringsverktyg använder. Precis som i CTL ses framtiden som en sekvens av tillstånd, eller väg genom systemet, men till skillnad från CTL undersöker inte LTL påståenden genom en förgrening utan genom individuella vägar, se figur 3. För att ett påstående ska vara sant måste varje enskilt påstående vara sant för varje väg i en mängd av vägar. LTL använder sig alltså inte av E – ’för vissa vägar’ eller ’det finns en väg’ (se tabell 1) som i CTL, utan undersöker alltid om ett påstående är sant ur ett globalt perspektiv. Ett påstående ϕ är sant om och endast om ϕ är sant för samtliga vägar i en mängd (Huth, 2004).



Figur 3 *Linjära sekvenser med avseende på* s_1.

LTL kan på samma sätt kombineras med de sedvanliga booleska operatorerna, men LTL skiljer sig lite vid användandet av temporala operatorer. LTL använder X, F, G och U som CTL, se tabell 2. Dock finns vissa skillnader i hur de används. I CTL måste X, F, G och U användas i par med A och E, medan dessa operatorer inte används alls i LTL. Utöver X, F, G och U tillkommer även R och W. Se tabell 4 (Huth, 2004).

Tabell 4 *Temporala operatorer som används i LTL, exklusive de som är samma som i CTL.*

Notation	Betydelse
R	Frigör
W	Svag tills

Till exempel:

(a W b): a är sant tills det att b blir sant. Dock måste inte b bli sant för att uttrycket ska vara sant. Detta till skillnad från (a U b) där b måste bli sant för att hela uttrycket ska vara sant.

(a R b): b måste vara sant tills det att a är sant eller a frigör b.

Dock kan sägas att även om R och W inte finns som separata operatorer i CTL kan innebörden av dem ändå uttryckas som kombinationer av andra operatorer i CTL.

LTL och CTL är båda mycket användbara temporala språk inom modellcheckning. Dock används de ofta för olika sorters system, eftersom vissa saker uttrycks lättare i det ena eller andra språket. Med LTL kan man till exempel inte uttrycka en egenskap som eventuellt ska vara sann, utan kan bara uttrycka att en egenskap alltid ska vara sann. På grund av detta har flera verktyg utvecklats för att kunna använda sig av båda språken. Dessutom finns det flera verktyg som kan hantera en kombination av både CTL och LTL, detta språk kallas CTL* (Huth, 2004).

2.3.3 Symbolisk modellcheckning

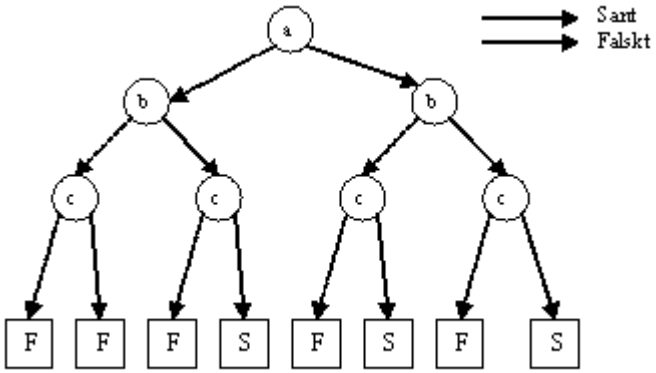
För att underlätta modellcheckning används ofta s k symbolisk modellcheckning. Metoden innebär att en mängd tillstånd undersöks samtidigt istället för varje tillstånd för sig. Detta underlättar verifieringsprocessen avsevärt och möjliggör verifiering av modeller med större antal tillstånd. Traditionellt används binära beslutsdiagram (binary decision diagrams) (BDD) för symbolisk modellcheckning, men under senare år har även verktyg baserade på satslogisk satisfierbarhets (SAT)-teknik utvecklats (Bérard, 2001), se avsnitt 2.3.5.

2.3.4 Binära beslutsdiagram (BDD)

BDD är en specifik datastruktur som kan användas för att representera en mängd tillstånd (Bérard, 2001) och som först utvecklades av Randal E. Bryant 1986. Se tabell 5 och figur 4.

Tabell 5 *Sanningstabell för (a||b)&&(a||c)*

a	b	c	S/F
0	0	0	F
0	0	1	F
0	1	1	S
0	1	0	F
1	0	0	F
1	0	1	S
1	1	1	S
1	1	0	F

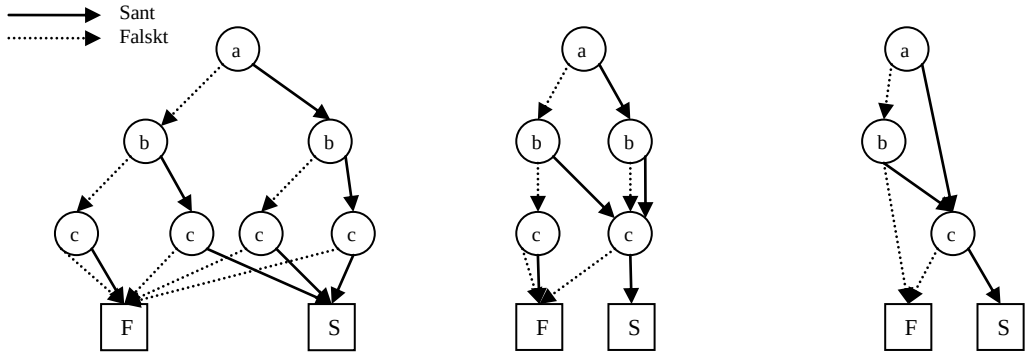


Figur 4 *Beslutsträd för (a&&b)||(b&&c).*

Med hjälp av BDD kan verktyg hantera fler antal tillstånd än om varje tillstånd skulle lagras enskilt. Detta leder till att verktygen kräver mindre minne vid verifieringen och därför kan hantera större modeller än vid icke-symbolisk modellcheckning. Dessutom är algoritmerna som används för verifieringen relativt enkla och lätta att implementera (Bérard, 2001).

BDD är egentligen reducerade beslutsträd. Denna reduktion sker på följande sätt:

Steg 1. Först exkluderas terminaldubbletter.
Steg 2. Sedan slås ekvivalenta noder samman.
Steg 3. Slutligen raderas de noder som inte ger upphov till något riktigt val, d v s båda barnen är identiska, se figur 5.



Figur 5 *Konstruktion av reducerat beslutsträd utifrån beslutsträdet i figur 4.*

Detta reducerade träd är enklare än vanlig tillståndsrepresentation, och samtliga booleska variabler kan användas i vid denna representation (Bérard, 2001).

2.3.5 Satslogisk satisfierbarhet (SAT)

Dessa typer av algoritmer för modellcheckning har utvecklats under senare år för att komplettera den traditionella symboliska modellcheckningen baserad på BDD. Det finns ett antal olika varianter, men den vanligaste är s k begränsad modellcheckning (Bounded Model Checking, BMC). Ett uttryck uppbyggt av ett antal variabler undersöks för att se om det finns en tilldelning av sant eller falskt till dessa variabler som gör så att hela uttrycket klassificeras som sant. Detta görs generellt genom att formeln konverteras till Konjunktiv Normal Form (CNF). Ett booleskt uttryck är skrivet på konjunktiv normal form om det består av konjunktioner av klausuler där klausulerna är uppbyggda av disjunktioner av booleska operatorer, t ex. (A||B) && (C||D). (Huth, 2004)

I SAT-tekniken bildas en struktur för att symbolisera övergångar hos ett system. Syftet med BMC är att utveckla denna struktur k gånger med start i valt initialtillstånd och slut i ett tillstånd där specificerat krav inte uppfylls. Varje sekvens är alltså lika lång eller kortare än k. En SAT-lösare söker igenom alla möjliga sekvenser för att undersöka om det finns något tillstånd där ett krav inte håller (Clark, 2006). Kraven specificeras ofta i CTL, men andra temporala språk kan också användas beroende på vilket verktyg som används.

BMC är inte helt optimal för modellcheckning av mjukvara, eftersom dessa system är för stora och kräver att värdet på k är orimligt stort. Nyare metoder som löser detta problem är under utveckling, och många verktyg för modellcheckning kombinerar SAT-teknik och symbolisk modellcheckning baserad på BDD (Clark, 2006).

2.3.6 Kraven

Med hjälp av CTL, LTL eller annat specificeringsspråk bildas olika krav som användaren vill att systemet ska uppfylla, t ex att A och B aldrig får ske samtidigt o s v. När samtliga krav är specificerade kan verifieringen genomföras. Det finns två grundläggande kravställningar.

1. Säkerhetskrav: Krav på systemet att något (dåligt) inte kommer att förekomma.
2. Livenesskrav: Krav på systemet att något (bra) måste förekomma (Huth, 2004).

Kraven är alltid utformade som påståenden eller frågor som kan besvaras med sant eller falskt. Vid varje tillstånd är alltså en egenskap antingen sann eller falsk.

2.3.7 Inbyggda kravställningar

Många verktyg har ett flertal inbyggda frågeställningar/krav som kan exekveras under verifieringen. Syftet med dessa är att kontrollera att specificeringen är stabil. Ofta förekommande sådana är t ex kontroller som undersöker om det finns onödiga tillstånd i modellen. Onödiga tillstånd kan leda både till att en modell blir enklare att förstå men också att en modell verkar tillkrånglad. Extra tillstånd påverkar egentligen inte en modells funktion, men det är likväl önskvärt att en modell är så lätt som möjligt att förstå, eftersom det oftast är många personer som arbetar med den. Viktigast att betänka är dock att extra tillstånd kräver extra minne vilket nästan alltid är en begränsande faktor (Ranville, 2004). Elimineringen av överflödiga tillstånd kan alltså vara av stor betydelse för om verifikationen ska fungera eller inte. Andra inbyggda kontroller kan inkludera undersökningar om huruvida några tillstånd är återvändsgränder (deadlocks) för systemet eller om någon överflödig kod (redundant code) förekommer. Med återvändsgränder menas tillstånd där två processer väntar på att den andra först ska avslutas, vilket leder till att simuleringen inte kan fortskrida. Med överflödig kod

menas kod som exekveras vid en simulering men som inte har någon inverkan på systemet (Ranville, 2004).

2.3.8 Manuella kravställningar

Det finns egentligen inga begränsningar av vilka krav som kan ställas på en modell. Tanken är dock att det ska vara frågor som kan besvaras med sant eller falskt. Bland de vanligaste frågorna finns frågor om t ex två eller fler villkor uppfylls samtidigt eller om ett visst villkor uppfylls efter ett annat och inom en viss tid (Ranville, 2004).

2.3.9 Utveckling av enklare verktyg för modellbaserad formell verifikation

Under senare år har ett antal enklare, mer användarvänliga verktyg för modellbaserad modellcheckning utvecklats. De bygger på en annan teknik än traditionell temporal modellcheckning (Rao, 2008). Många är baserade på SAT-teknik eller använder temporala språk, men matematiken är i dessa fall helt gömd. I dessa nya verktyg konfronteras användaren inte med komplicerad matematik; inte heller krävs det extra arbetet att lära sig ett nytt, formellt språk. De nyutvecklade verktygen är kompatibla med de vanliga modellutvecklingsspråk som används inom industrin, t ex Simulink/Stateflow (Rao, 2008).

Vid nyttjandet av dessa nya verktyg behöver användaren inte använda sig av kravspecifikationer i temporalt språk. Kraven beskrivs istället direkt på det språk som användaren har valt att modellera i. Även om kravspecificeringssteget kvarstår blir det betydligt lättare att utföra. Vid användandet av vissa nya verktyg behövs ingen konvertering av modellen till ett annat modelleringsspråk utan de fungerar direkt på det språk som användaren valt. Hos andra verktyg sker denna konverterig automatiskt, vilket underlättar modellcheckningen avsevärt.

2.3.10 Simulink

Simulink är en miljö för modellering, simulering och analysering av dynamiska system utvecklat av The Mathworks. Modelleringen sker i huvudsak i form av konstruktion av blockscheman. Programmet innehåller ett bibliotek med ett stort basutbud av block för en mängd funktioner som t ex derivata och matematiska operationer mm. Det finns även möjlighet för användaren att konstruera egna bibliotek med egendesignade block med önskade funktioner (The Mathworks, 2008).

2.4 Verifiering

Sista steget i modellcheckningen är det lättaste och sker mer eller mindre automatiskt. Under verifieringen utförs en sökprocedur för att undersöka om transformationssystemet uppfyller specifikationen. Verifieringen avslutas alltid med att ge ett booleskt svar; d v s sant eller falskt. Dessutom genererar verifieringen ett motexempel för att visa var felet ligger (Zheng, 2007). På så sätt är det lättare att åtgärda de fel som upptäcks. Motexemplet består av en sekvens av tillstånd som leder tillbaka användaren till det tillstånd där felet uppstod (Huth, 2004).

Det finns två grundläggande former av verifieringsalgoritmer: dels nåbarhetsanalys, vilket kontrollerar säkerhetskraven, dels loopanalys som kontrollerar livenesskraven. Verktyget som används för verifieringen söker igenom alla möjliga tillstånd som systemet kan befinna sig i

för att undersöka om modellen uppfyller alla de krav som ställts. Det är detta totala genomsökande som är den stora skillnaden jämfört med testning där det endast är möjligt att undersöka en bråkdel av de befintliga tillstånden hos en modell.

2.4.1 Svårigheter med modellcheckning

Även om modellcheckning verkar vara en bra metod är den naturligtvis inte perfekt. Det finns ett flertal negativa sidor med metoden:

- Metoden är effektiv för ett givet krav i specifikationen, men i likhet med andra verifieringsmetoder kan en kravspecifikation inte rimligtvis innehålla alla krav som ett system borde uppfylla.

- Användaren måste kunna översätta modellen till ett specifikt modelleringsspråk som verktyget förstår. Detta moment har dock underlättats då vissa nya verktyg fungerar direkt på t ex Simulink/Stateflow eller C-kod eller konverterar modellen automatiskt.

- Användaren behöver lära sig ett specifikt språk för att kunna beskriva kraven på systemet. Många krav som behöver verifieras kan vara svåra eller rent av omöjliga att beskriva i något av dessa språk.

- Det finns ingenting som kontrollerar själva kraven i specifikationen och felaktigt ställda krav kan leda till en felaktig modell. Detta är dock ett problem som finns inom alla valideringss- och verifieringsmetoder.

- Verktyg för modellcheckning kan inte hantera hur stora modeller som helst. Om systemet innehåller för många tillstånd blir modellcheckningen omöjlig.

- Inom modellutveckling är det vanligt att det finns applikationer som innehåller data i form av flyttal. De flesta modellcheckningsverktygen kan inte hantera sådana data eftersom att de innehåller allt för många tillstånd. Detta kan vara ett betydande problem som förhindrar formell verifiering av modeller av detta slag (Ranville, 2004). Dock finns det ett fåtal nyare verktyg som innehåller diverse nyskapande tekniker för att ta hand om detta problem.

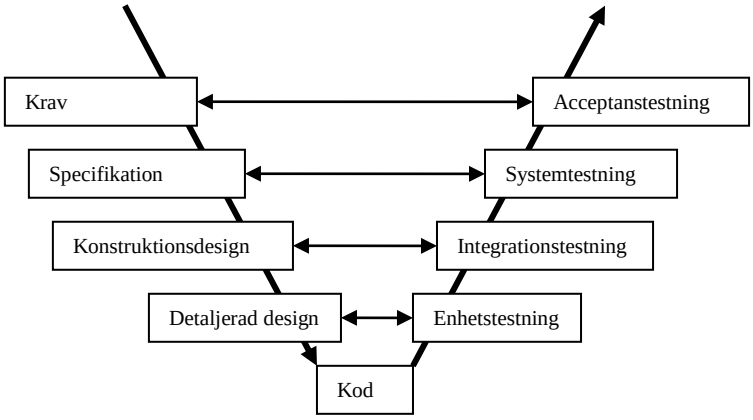
3 Testning - teori

Testning är den form av validering som används mest. Syftet med testning är, i likhet med modellcheckning, att undersöka kvalitén på ett system och att upptäcka fel som medför att ett system inte uppträder som förväntat.

Testning är vidare ett systematiskt sätt att undersöka hur korrekt en modell är. Testningen utförs genom att testaren genomför experiment med modellen i form av körningar. Testningen utförs i en kontrollerad miljö där det är känt hur systemet ska reagera. Efter exekveringen utvärderas hur korrekt systemet är. Testning är en väldigt tidskrävande del av systemutvecklingen. Det är aldrig möjligt att vara 100 % säker på att ett system är felfritt eftersom att det i princip är omöjligt att testa alla olika simuleringssekvenser. På grund av detta är det av stor vikt hur olika testfall designas (Andersson, 2005).

3.1 Typer av test

Testningen utförs vid flera olika stadier i utvecklingsprocessen och genomförs ofta enligt den sk V-modellen, se figur 6, (Andersson, 2005). Fyra olika typer av test brukar användas: enhetstestning (unit testing), integrationstestning (integration testing), systemtestning (system testing) och acceptanstestning (acceptance testing).



Figur 6 *V-modellen för testning av mjukvara.*

3.2.1 Enhetstestning

Enhetstestning är en metod för att testa enskilda program eller funktioner och är den första testningen som utförs. Poängen är att verifiera att individuella enheter av källkoden fungerar som de ska. Testtäckning är extra intressant inom enhetstestning, detta för att undersöka att så stora delar av modellen som möjligt har undersökts (Andersson, 2005).

3.2.2 Integrationstestning

Vid integrationstestning undersöks konstruktionen hos ett system genom att flera enheter som ska interagera testas samtidigt (Andersson, 2005).

3.2.3 Systemtestning

Systemtestning delas ibland upp i funktionell systemtestning och icke-funktionell systemtestning. Vid funktionell systemtestning kan hela system testas mot en viss specifikation så att kraven kan verifieras och på så sätt kontrollera funktionaliteten hos ett system. Vid icke-funktionell systemtestning valideras systemegenskaper såsom flexibilitet och prestanda (Andersson, 2005).

3.2.4 Acceptanstestning

Acceptanstestning är den tredje formen av testing som utförs på ett system. Acceptanstestningen används egentligen inte för att finna fel i ett system, utan för att förtroende ska skapas för systemet och utförs ofta av eller tillsammans med köparen (Andersson, 2005).

3.3 Testtekniker

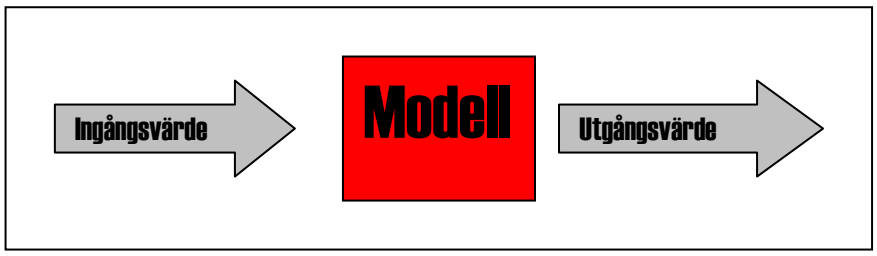
Det finns generellt två tekniker som används vid testning: statisk och dynamisk testning.

3.3.1 Statisk testning

Statisk testning är en form av testing där själva programmet eller systemet inte körs. Tekniker för statisk testning kan bland bestå av genomläsning av kod eller jämförelse mellan kod och någon form av kodstandard eller dylikt. Statisk testning kan utföras antingen manuellt eller automatiskt med hjälp av en kompilator eller ett separat program. Kompilering är det vanligaste tillvägagångssättet (Andersson, 2005).

3.3.2 Dynamisk testning

Dynamisk testning innebär att programmet som ska testas körs. Efter detta görs en analys av resultatet. Dynamisk testning kan i princip delas upp i två olika kategorier: black-box testning och white-box testning. Black-box testning innebär att den som genomför testningen endast vet vilket utgångsvärde som ska genereras av ett specifikt ingångsvärde, se figur 7, men inte hur eller varför resultatet blir som det blir.



Figur 7 *Black-box testning av mjukvara.*

Vid white-box testning finns alla de processer som utförs i modellen tillgängliga för användaren. Användaren kan på detta sätt använda programkodens struktur för att utföra testningen (Mattson och Niska, 2003). Vid white-box testning finns det möjlighet för användaren att undersöka varje villkor och varje beslut i systemet. Vid denna slags testning måste användaren veta precis vad som ska hända i systemet och varför, för att denna ska kunna förstå om systemet avviker från det normala i något hänseende (Andersson, 2005).

Sällan används endast en av de ovan nämnda testningsvarianterna utan de ska ses som komplement till varandra och utnyttjas för att hitta olika sorters fel (Mattson och Niska, 2003).

Det finns även ett mellanting mellan white-box och black-box testning som kallas grey-box testning. I dessa fall är vissa delar av ett system kända, t ex en större struktur av moduler, men inte modulerna i sig (Andersson, 2005).

3.4 Automation

Manuell testning är tidskrävande, och för att öka testningens effektivitet automatiseras stora delar av testningsprocessen. Automatisering av bl a regressionstest som måste köras varje gång ett system uppdaterats för att försäkra att ändringarna fungerar som de ska, ökar effektiviteten på ett påtagligt sätt. Dessutom leder automatisering till att alltfler test kan utföras på mindre tid och därför också oftare än vid manuell testning. Därtill kan testfallstillverkarna i stället ägna mer tid till åt att designa bättre testfall (Andersson, 2005).

Det finns dock några nackdelar med testautomatisering, och det kommer alltid att finnas test som enklast utförs manuellt. Testning som innefattar stora delar fysisk interaktion, som att t ex stänga av och på ett system eller koppla ifrån något delsystem, kommer troligen alltid att vara lättast att utföra manuellt (Andersson, 2005).

Det finns ett antal olika tillvägagångssätt när det gäller att automatisera testning. Några av dessa är slumpad testgenerering (random test generation), kombinatorisk testgenerering (combinatorial test generation) och modellbaserad testgenerering (model-based test generation) (Mattson och Niska, 2003).

3.4.1 Slumpad testgenerering

Slumpad testgenerering är en enkel black-box metod och går ut på att användaren specificerar datatyper och databegränsningar för alla insignaler. Därefter genererar verktyget helt slumpartade test utifrån denna specificering. Denna typ av test måste ofta vara omfattande för att uppnå önskad testtäckning (Andersson, 2005).

3.4.2 Kombinatorisk testgenerering

Kombinatorisk testgenerering används till stor del för att minska storleken på testvektorerna utan att för den delen minska testtäckningen. Antalet test som behövs för att täcka alla möjliga parvis eller n-vägskombinationer av insignaler växer logaritmiskt med antalet insignaler och linjärt med antalet olika värden en insignal kan anta, medan antalet test som behövs för att testa alla möjliga kombinationer av insignaler växer exponentiellt. Ett verktyg som skapar antingen parvis eller n-vägskombinationer kan på detta sätt minska antalet test och ändå åstadkomma en högre testtäckning än vid slumpad testning (Andersson, 2005).

3.4.3 Modellbaserad testgenerering

Modellbaserad testgenerering är en metod för att generera test utan omfattande manuellt arbete och används av flertalet modellbaserad verktyg, bl.a. Reactis, se avsnitt. 4.3.1. Verktygen konverterar de designade modellerna till en form ur vilka testvektorer sedan kan bildas. Algoritmerna som används för testvektorgenereringen försöker öka modelltäckningen

dynamiskt under testgenereringen. Hur algoritmerna ser ut varierar från verktyg till verktyg, men är ofta komplexa och därför heller inte offentliga för allmänheten att ta del av (Andersson, 2005).

3.4.4 Svårigheter

Även vid användning av ovan nämnda verktyg för testgenerering är det fortfarande komplicerat att designa och skapa bra testfall. Även om mycket att automatiserat kräva ändå att vissa specifika parametrar så som testsekvenser, hantering av flyttal, datatyper och databegränsningar sätts. Användaren behöver sålunda stor betydande kunskap om hur modellen fungerar för att kunna skapa korrekta och användbara testfall.

3.5 Testtäckning

Testtäckning är ett sätt att undersöka kvaliteten för ett visst testfall och används ofta i traditionell testning. Även hos vissa verktyg för modellcheckning används denna kvalitetsgranskning. Vid mätning av testtäckning undersöks hur stora delar av modellen som verkligen används vid valideringen. Det finns ett antal olika varianter av testtäckning. De oftast förekommande är villkorstäckning (condition coverage), beslutstäckning (decision coverage) och modifierad villkors/beslutstäckning (MC/DC)(modified condition/decision coverage) (Andersson, 2005).

3.5.1 Beslut

Ett beslut kan ses som en nod i ett program. Besluten kan bestå av ett antal komplexa booleska strukturer, men är alla en punkt där en förgrening sker och informationsflödet kan välja en av dessa förgreningar (Andersson, 2005).

3.5.2 Villkor

Ett villkor är ett uttryck med värden av boolesk karaktär, d v s sant eller falskt, som inte kan brytas ner till enklare form. Ett beslut består ofta av ett antal booleska villkor (Andersson, 2008).

3.5.3 Beslutstäckning

För varje beslut ett verktyg måste ta finns två möjliga svar, sant eller falskt. En komplett täckning innebär att varje frågeställning har evaluerats till både sant och falskt (Reactive Systems Inc., 2008).

3.5.4 Villkorstäckning

I likhet med varje beslut ger varje villkor också upphov till två valmöjligheter, sant eller falskt. En täckning är i detta fall komplett om varje villkor har evaluerats till både sant och falskt. Beslutstäckning och villkorstäckning är mycket lika, men man måste komma ihåg att fullständig villkorstäckning inte nödvändigtvis betyder fullständigt beslutstäckning (Reactive Systems Inc., 2008).

3.5.5 Modifierad villkors/beslutstäckning (MC/DC)

MC/DC introducerades av John J. Chilenski vid Boeing under det tidiga 90-talet. MC/DC är en form av täckning som är svårare att uppnå än andra och anses som ett av de bästa måtten på hur bra en testkörning är. För att nå full MC/DC måste följande punkter alla vara uppfyllda.

- Varje beslut måste testa alla möjliga utfall.
- Varje villkor som finns i ett beslut måste testa alla möjliga utfall.
- Samtliga in- och utgångar måste användas.
- Varje villkor i ett beslut måste visas enskilt kunna påverka beslutets utfall.

(Reactive Systems Inc., 2008).

För säkerhetskritiska system är det av största vikt att modellen uppfyller MC/DC för att systemet ska kunna anses som tillräckligt säkert för att lanseras.

4 Verktyg

4.1 Verktyg för modellcheckning av modeller i Simulink

Det finns ett antal verifieringsverktyg på marknaden som tillåter användaren att utföra modellcheckning direkt på modeller designade i Simulink/Stateflow.

4.1.1 Simulink Design Verifier (SDV)

Simulink Design Verifier är ett verktyg från Mathworks som genererar kontroller för modeller i Mathworks eget språk. Verktyget integreras lätt i modelleringsmiljön och använder sig av formella analystekniker tillhandahållna av Prover Plug-In från Prover Technology (The Mathworks, 2008). Verktyget brukar automatisk testfallsgenerering, se avsnitt 3.1, abstraktion, se avsnitt 4.2.3, och SAT-teknik, se avsnitt 2.3.5, för att verifiera modellkrav.

Verktyget detekterar tillstånd som är icke-nåbara och bevisar matematiskt om ett givet villkor uppfylls eller ej. Verktyget producerar vid verifieringen även analysrapporter innehållande bl a villkorstäckning, beslutstäckning och MC/DC. Användaren kan designa egna kontroller direkt i Simulinkmodellen genom att skapa egna verifieringsblock.

SDV kan antingen användas som egenskapsbevisare (property prover) eller som testgenerator. Vid bevisning av egenskaper finns tre sätt att gå till väga för att undersöka krav på en modell. SDV använder sig av antaganden (assumptions), bevisåtaganden (proof objectives) och verifiering av delsystem (subsystem verifications) vid egenskapsbevisning.

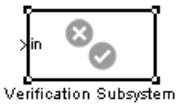
Antaganden, se figur 8, används för att begränsa en viss signal vid en kravställning, t ex att en signal ska vara 1 eller 0 eller ligga inom ett visst intervall. Inställningar kan också användas för att specificera om signalbegränsningen ska gälla under hela bevisningen eller bara initialt.

Figur 8 *Antagandeblock.*

Bevisåtaganden, se figur 9, uttrycker egenskaper som en signal måste uppfylla vid bevisningen. Egenskaperna kan uttryckas genom siffror, intervall av siffror samt sant eller falskt. Ett exempel på detta är att ett bevisåtagande hävdar att utsignalen ska ha egenskapen 1 när ett antagande på insignalen säger att den ska vara 0.

Figur 9 *Bevisåtagandeblock.*

Verifiering av delsystem representerar bevis eller testobjekt utan att påverka simuleringen eller den genererade koden, se figur 10.



Figur 10 *Block för verifiering av delsystem.*

I ett delsystem tillåts användaren tillfoga komponenter från SDVs blockbibliotek utan att modellen berörs. Användaren kan antingen undersöka ett krav i taget eller flera på en gång.

Med hjälp av ovanstående block kan SDV formellt bevisa, eller förkasta, egenskaper eller krav hos en modell. Vid förkastandet av ett krav skapar SDV ett motexempel för att visa var felet ligger.

Vid användning av SDV som testgenerator kan användaren använda testobjekt, testvillkor eller verifiering av subsystem.

Testobjekt används för att beskriva specifika egenskaper som ett testfall måste uppfylla. Testvillkor begränsar värden av signaler för olika testfall vid testsimulering. På samma sätt som vid antaganden kan användaren specificera om signalbegränsningen ska gälla under hela simuleringen eller bara initialt.

Verifiering av delsystem kan även användas vid testning och fungerar då på samma sätt som vid egenskapsbevisningen, se ovan. Med hjälp av ovanstående block producerar SDV testfall som uppfyller specifika kriterier (SDV User's Guide, 2008)

En fördel med SDV är att modelleringsmiljön är väldigt lik verifieringsmiljön, vilket borde underlättar verifieringen för användaren.

4.1.2 Embedded Validator

För att utföra verifieringen med Embedded Validator krävs att modellen i Simulink konverteras till ett annat verifieringsspråk. Med hjälp av Target Link, ett verktyg som automatiskt genererar C-kod från Simulinkmodeller, kan denna konvertering ske. C-koden används sedan för själva verifieringen, vilket gör detta verktyg till ett mellanting mellan modellbaserad modellcheckning och traditionell modellcheckning på befintlig kod (OSC Embedded Systems, 2008).

Embedded Validator tillhandahåller två olika valmöjligheter för verifieringen, en komplett och en begränsad analysmetod.

Den kompletta metoden benämns *Verification Interacting with Synthesis* (VIS) och utnyttjas för verifiering av system med begränsat antal tillstånd. Denna analysmetod går igenom alla möjliga tillståndssekvenser som finns i systemet (OSC Embedded Systems, 2008).

ProverCL Plug-In benämns den begränsade metod som utnyttjas för modeller med oändligt, eller väldigt stort antal, tillstånd. ProverCL begränsar antalet tillstånd i varje sekvens som kontrolleras. Denna metod verifierar också alla möjliga kombinationer av tillstånd, men

begränsar sekvensernas längd. Längden på sekvenserna begränsas antingen av antal steg i sekvenserna eller hur lång tid det tar att verifiera dem. Detta är således en form av begränsad modellcheckning (BMC) (OSC Embedded Systems, 2008), se avsnitt 2.3.5.

EmbeddedValidator innehåller ett bibliotek med fördefinierade block med funktionella krav. Kravspecificeringen bygger på en temporal logik som dock är osynliggjord för att verifieringen ska vara så enkel som möjligt. I blocken fyller användaren i simpla booleska uttryck för att formulera olika krav. Om ett krav inte uppfylls vid en körning produceras en felrapport. Verktyget genererar då automatiskt en M-fil, d v s en editorfil i Matlabs eget språk, som kan användas för felsökning i Matlab (OSC Embedded Systems, 2008).

4.2 Verktyg för modellcheckning av modeller i C-kod

Under senare år har verktyg utvecklats för att utföra formell verifiering direkt på modeller som är skrivna i C-kod. Sådana verktyg underlättar i betydande utsträckning arbetet för programmerare, eftersom man inte behöver manuellt omformulera hela modellen för att matematiskt undersöka om den är korrekt. Ett av de största problemen med verifieringsverktygen för C-kod är att de flesta verktygen är designade för att verifiera ANSI C-kod. C-kod är en standard som utvecklats vid American National Standard Institute och som inte alla andra C-kodstandarder är kompatibla med (Schlich, 2006). Detta innebär att modellen till slut ändå måste modifieras, något som i många fall är komplext, tar tid och kostar pengar. Detta är emellertid ett problem som är lätt att lösa om endast en och samma standard skulle användas. Om det dessutom är känt från början att modellcheckning ska utföras på koden som skrivs bör detta inte vara ett bestående problem.

Det finns ett par verktyg för modellcheckning direkt på C-kod på marknaden, se nedan.

4.2.1 Spin Model Checker

Spin Model Checker är ett populärt verktyg för formell verifiering av mjukvarusystem. Verktyget började utvecklas av Unix Group of the Computing Science Research Center vid Bell Laboratories redan på 80-talet och har funnits på marknaden sedan 1991. Spin är populärt, eftersom det, till skillnad från andra formella verifieringsverktyg, framför allt verifierar mjukvara och inte hårdvara (Spinroot, 2007).

Senare versioner av verktyget (Spin 4.0 och senare) gör det möjligt att utföra formell verifiering direkt på C-kod. Spin använder sig av en metod som kallas on-the-fly, vilket innebär att den inte behöver bilda en fullständig tillståndsgraf, eller Kripkestruktur, som de flesta andra verktyg. Kravspecificeringen kan anges med hjälp av villkor, uttryckta i det temporala språket LTL eller av mindre formella uttryck bestående av s k aldrig-krav i Promela, se nedan. I syfte att minska modellens storlek använder sig Spin av BDD för att beskriva mängder av tillstånd om så behövs och även av effektiva tillståndsreduktionstekniker (Spinroot, 2007).

För att verifiera en modell bildas en formell version av denna i Spins modelleringsspråk Promela. Promela är ett uttrycksfullt modelleringsspråk som består av processer, variabler och meddelandekanaler och som är väldigt likt vanlig programmeringskod som t ex C-kod. Verktyget transformerar automatiskt C-kod till promela, något som avsevärt underlättar processen. För att transformationen ska fungera korrekt måste dock användaren definiera

vissa regler för kodabstraktion. Detta kräver manuellt arbete och ställer dessutom kunskapskrav på användaren (Hotzmann, 2000).

Antal tillstånd som Spin kan hantera avgörs till stor del av hur mycket minne som finns tillgängligt på den dator som används. Med en kraftfull dator finns möjlighet för användaren att verifiera modeller på upp emot en trillion tillstånd (Spinroot, 2007). Dessa siffror är hämtade från Spins egen användarmanual. Dock har inga tekniska rapporter påträffats där dessa uppgifter bekräftats, något som tyder på att denna stora tillståndshantering inte förekommit.

Liksom de flesta andra verifieringsverktyg kan Spin automatiskt undersöka den inmatade modellen för bl.a. återvändsgränder, icke-använd kod och icke-nåbara tillstånd. För andra specifika kravkontroller får användaren definiera krav i LTL (Holtzmann, 2000).

Spin kan användas på tre sätt:

1. Som simulator för att utföra olika former av simuleringstester.
2. Som verifierare för att matematiskt bevisa användardefinierade kravställningar.
3. Som approximationssystem för att validera även väldigt stora modeller med full täckning.

I Promela finns inga flyttal, men användaren kan lösa detta problem genom att lagra flyttalen som symboliska konstanter. Symbolerna används sedan i stället för de värden de representerar vid verifieringen (Spinroot, 2007).

Ett krav för att Spin ska kunna användas som verifieringsverktyg är att modellen som ska granskas är skriven i ANSI C-kod.

4.2.2 VeriSoft

VeriSoft är ett verktyg från Lucent Technologies vid Bell Laboratories. VeriSoft är lätt att använda eftersom det fungerar automatiskt och helt kan integreras i befintlig modelleringsmiljö. Dessutom kan verifieringen appliceras direkt på modeller i programmeringsspråken C och C++. För att verifieringen ska kunna fungera som avsett måste några kriterier vara uppfyllda. Först måste alla processer i systemet kommunicera via verktygets egna interna processobjekt (IPC). Sedan måste systemet vara slutet för att kontrollen ska fungera. Om systemet inte är slutet från början måste det slutas, vilket innebär att systemmiljön måste simuleras genom en icke-deterministisk process (Bell Labs, 2008).

VeriSoft genomför sina undersökningar automatiskt och numrerar alla de sekvenser som ett system kan ge upphov till. Längden på dessa sekvenser bestäms av användaren och kallas djup.

Användaren kan använda sig av krav i form av påståenden som systemet måste uppfylla för att det ska anses korrekt. För att utforma påståenden används en operation, ”VS_assert” som kan tillfogas vilken process som helst i koden. I likhet med många program ska detta vara ett påstående som är antingen sant eller falskt. Om ett påstående efter evalueringen är falskt anses det överskridet och därmed finns något i modellen som inte är korrekt.

VeriSoft kan hantera fyra typer av fel: återvändsgränder (deadlocks), divergeringar, låsningar (livelocks), påståendeöverskridningar (assertion violation).

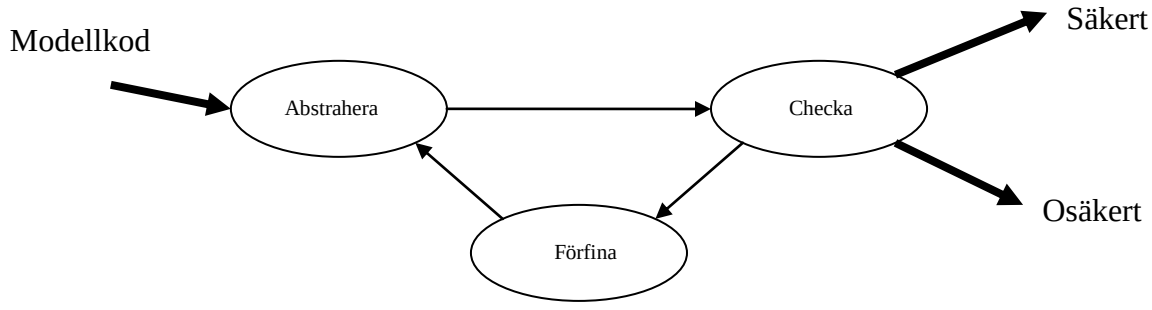
- Återvändsgränder är ett tillstånd där två processer väntar på att den andra ska slutföras och därför sker ingenting, d v s alla processer är blockerade.
- Divergering uppstår då en process inte klarar av att kommunicera med resten av systemet längre än en given tidslängd.
- Låsningar uppträder då en process är blockerad vid en sekvens som har ett större antal tillstånd än vad användaren angivit.
- Påståendeöverskridningar sker då ett av användaren angivet påstående inte uppfylls.

4.2.3 BLAST – Berkeley Lazy Abstraction Software Verification Tool

BLAST är ett verifieringsverktyg för modellcheckning av C-kod som utvecklats vid Berkeley, University of California. Syftet med BLAST är att undersöka om ett system uppfyller vissa angivna krav och verktygets konstruktion är baserat på konceptet abstrahera – checka – förfina. Se figur 9.

Verifieringen med BLAST utförs genom att verktyget resonerar om ett tillstånd rubricerat ’error’ är nåbart. Verifieringen avslutas med att verktyget antingen berättar för användaren att systemet är säkert, d v s givet tillstånd är inte nåbart, eller att systemet inte är säkert och ger då ett motexempel för att visa var felet ligger. Dessa tillstånd beskrivs, i likhet med andra verktyg, av påståenden som ska vara antingen sanna eller falska.

Vid abstraheringen används ett set av predikat för att abstrahera programmet på så sätt att varje abstraherat tillstånd representeras av predikat med tilldelningen sant.
Vid checkningen används den abstraherade modellen för att kontrollera om ett visst krav uppfylls, det vill säga om det är säkert eller inte. Om den abstraherade modellen är säker är den verkliga modellen det också. Om den däremot inte är säker genereras ett motexempel. Verktyget undersöker först om det abstraherade motexemplet motsvarar ett verkligt motexempel. Om så är fallet producerar BLAST detta motexempel för att visa att det finns fel i systemet. Om det abstraherade motexemplet inte motsvarar ett verkligt motexempel behöver abstraktionen förfinas. Förfiningen genererar nya predikat för att bilda en ny abstraktion av systemet i abstraheringssteget, se figur 11 (BLAST User’s Manual, 2005). Loopen fortsätter till ett krav antingen fastställs som säkert eller osäkert.



Figur 11 *Abstrahera – Checka - Förfina loopen hos BLAST.*

BLAST använder sig av s k on-the-fly abstraktion, vilket innebär att abstraheringen av en region endast sker om denna region behövs för nästa steg i kontrollen. Abstraktionen drivs

alltså av en kontrollprocess vilket leder till att ingen onödig abstraktion sker. Detta leder i sin tur till att mindre minne krävs. Även s k förfining på begäran används av BLAST. Detta innebär att verktyget återanvänder partiella svar som har framkommit vid tidigare iterationer. Detta betyder att verktyget inte behöver förfina områden som redan i tidigare steg förfinats och bevisats säkra (BLAST User's Manual, 2005).

Första steget i verifiering med BLAST utgörs av att verktyget konverterar C-koden till en dataflödesgraf (CFG), vilket är en graf som representerar alla sekvenser som är möjliga vid en exekvering.

För att minska antalet tillstånd använder sig BLAST av BDD och utför alltså symbolisk modellcheckning, se 2.4.4.

För att kunna fungera felfritt är BLAST beroende av några andra verktyg. Bland annat tar verktyget Simplify hand om abstraheringen och verktyget Vampyre används för att utforma predikaten (BLAST User's Manual, 2005).

4.3 Verktyg för testning av modeller i Simulink/Stateflow

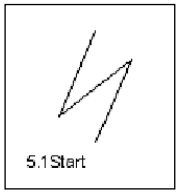
Det finns även verktyg som använder sig av testvektorgenerering som utnyttjas för validering av system utvecklade i en modellbaserad miljö. Ett vanligt förkommande verktyg för testning av modeller i Simulink/Stateflow är Reactis Validator.

4.3.1 Reactis Validator (Reactive Systems)

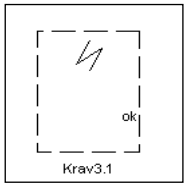
Validator är egentligen bara en mindre del av ett större verktyg för undersökning av modelltäckning. Verktyget används genom att användaren laddar upp Simulink/Stateflow modeller för att sedan validera dem. En stor skillnad mot formell verifiering är att även om krav inte falsifieras kan användaren aldrig vara helt säker på att det inte finns ett testfall där fel skulle kunna uppstå. Som ovan nämnt finns ingen möjlighet för ett testningsverktyg att undersöka alla möjliga simuleringssekvenser. Ju sämre testtäckning verktyget genererar desto mindre tillförlitligt är kravvalideringen (Reactive Systems, 2008).

Validator är ett modellbaserat verktyg och använder sig således av modellbaserad testgenerering. Verktyget använder sig även av slumpad testgenerering som samtidigt också är modellbaserad. Verktyget är helt automatiserat och testgenereringen kräver inget manuellt arbete. Användaren kan dock manuellt lägga in krav, specificerade enligt nedan, för att vidare undersöka modellens uppträdande.

I Validator kan användaren välja mellan att specificera kraven antingen som uttryckskrav (expression assertions), se figur 12, eller som diagramkrav (diagram assertions), se figur 13.

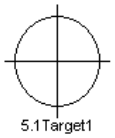


Figur 12 *Uttryckskrav i Validator.*

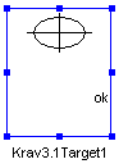


Figur 13 *Diagramkrav i Validator.*

Uttryckskrav består av booleska uttryck likt dem i C-kod och diagramkrav består av block från Simulinks eller Stateflows bibliotek. Verktyget använder sig av även av mål definierade av användaren (user-defined targets) för att genomföra valideringen. Målen uttrycks på samma sätt som kraven, d v s antingen som uttrycksmål, se figur 14, i form av booleska uttryck eller som diagrammål, se figur 15, designade i Simulink/Stateflow.



Figur 14 *Uttrycksmål i Validator.*



Figur 15 *Diagrammål i Validator.*

Kraven används för att uttrycka egenskaper eller villkor som alltid måste vara sanna för en modell. För varje krav söker Validator efter en simulering där det definierade kravet är falskt. Ett krav är falskt om utfallet är ’falskt’, vid uttryckskrav, eller ’falskt’ eller noll, vid diagramkrav. Ett påstående anses som täckt om ett fel hittas, det vill säga ett täckt krav är dåligt ur valideringssynpunkt (Reactive Systems, 2008).

Mål definierade av användaren nyttjas för att bestämma specifika systemexekveringar som användaren vill att simulatorn ska utföra. Till skillnad från krav används målen för att undersöka ett visst test som konstruerats så att det sedan kan användas inom en testsekvens. Målen är särskilt användbara när man ska undersöka hur en modell uppträder under vissa specifika förhållanden (Andersson, 2005). Till skillnad från krav är alltså ett täckt mål positivt ur verifieringssynpunkt. Både krav och mål är betydelsefulla för att modellvalidering med Validator ska ge ett tillfredsställande resultat (Reactive Systems, 2008).

I verktyget ingår en rapport om grad av täckning, inklusive MC/DC, som visar hur tillförlitlig valideringen är. Dessutom finns en simulator i Validator som visar vilka delar av modellen som exekverats, vilka som inte har det och vilka delar som inte är nåbara. Verktyget kan hantera påtagligt stora modeller (Ranville, 2004).

En nackdel med Validator är att verktyget inte stödjer hela Simulinks bibliotek och att det använder sig av vissa andra modelleringsregler. Detta betyder att modellen måste utvecklas från början med åtanke att Reactis Validator ska användas för verifieringen. I många fall går det att modifiera modellen i efterhand, men det kräver stora kraftansträngningar och mycket tid.

5 Modellcheckning med SDV

Simulink Design Verifier är ett verktyg som är integrerat i Simulink och används för att utföra både testgenerering och s k egenskapsbevisning, d v s formell modellcheckning av Simulinkmodeller, se avsnitt 4.1.1. Den uppenbara fördelen med att använda ett verktyg som är integrerad i modelleringsmiljön är att nödvändiga språkkonverteringar är helt automatiserade.

5.1 Grundläggande arbetsgång vid användning av SDV

- Försäkra att modellen är kompatibel med SDV, om så inte är fallet måste modellen modifieras för att uppnå kompabilitet.
- Lägga till block från SDVs bibliotek för att definiera antaganden och bevisobjekt som verktyget sedan ska använda sig av vid bevisningen.
- Specificera de valmöjligheter som finns när SDV ska utföra bevisningen, t ex måste användaren konfigurera verktyget till att använda fixa steg vid verifieringen.
- Utföra verifiering och därefter undersöka resultaten som producerats (SDV User's Guide, 2008).

5.2 Användarvänlighet

Syftet med SDV är att underlätta för användaren vid verifiering av en modell. Verktyget är utformat så att formell verifiering ska vara möjlig utan att användaren behöver komma i kontakt med komplicerad matematik eller okända specificeringsspråk.

Enkelheten i själva verifieringssteget är ett stort plus för SDV. Utöver användarvänlighet innehåller dock verktyget ett antal komplikationer. Det första som måste göras är att kontrollera så att modellen är kompatibel med SDV. Med tanke på att SDV är ett verktyg tillhörande Simulink kan tyckas att en modell skapad i Simulink automatiskt borde vara kompatibel med verktyget, men så är inte fallet. Andra svårigheter för SDV kommer att beskrivas nedan.

5.2.1 Kompatibilitetsundersökning

Det första användaren måste göra vid användning av SDV är att undersöka om modellen som ska verifieras är kompatibel med verktyget. Om modellen inte är designad för att använda SDV som verifieringsverktyg är risken stor att det kommer att uppstå problem. Det finns ett stort antal Simulinkblock och även funktioner som inte kan hanteras av SDV. Hur användaren kommer runt detta beskrivs i avsnitt 5.1.3 nedan.

Om modellen inte är kompatibel med SDV produceras ett felmeddelande där det framgår vilken del av modellen och vilket block som inte stöds av SDV. Detta underlättar delvis för användaren, men är likväl inte helt problemfritt, eftersom att detta felmeddelande inte alltid fungerar perfekt. Dels kan undersökningen förbise vissa block, eller funktioner, och påstå att dessa är kompatibla fast de senare visar sig inte vara det, och dels kan felmeddelandet ibland visas utan att det specificeras var i modellen felet ligger. Även vid en relativt liten modell blir det väldigt svårt att finna fel när detta händer.

SDV verkar ha stora problem med att undersöka kompatibilitet hos Stateflowdiagram. Diagrammen kan passera kompatibilitetstestet utan anmärkning: När verktyget sedan ska användas produceras dock felmeddelanden utan att de specificerar var felen ligger. Dessutom kan dessa funktioner i Stateflow medföra att själva egenskapsbevisningen falsifierar en egenskap och producerar motexempel, fast denna egenskap egentligen är helt korrekt.

Dessa kompatibilitetssvårigheter medför stora komplikationer för användaren och verifikationen blir tidskrävande. När fullständig kompatibilitet föreligger är verktyget dock i grunden lätt att använda. Att lägga in bevisobjekt, antaganden och analysera resultaten skapar inga stora svårigheter.

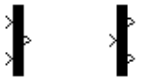
5.2.2 Hantering av icke-stödda block

Det finns ett stort antal block från Simulinks bibliotek som inte stöds av SDV. Detta leder till att det blir krävande för den som ska verifiera modellen om denna omständighet inte har beaktats då modellen designades. Några exempel på detta beskrivs nedan. För fullständig lista på vilka block och funktioner som SDV inte kan hantera, se appendix A1.

- Bussblock är block som antingen samlar flera signaler till en signal, busskapare (bus creators) eller separerar en signal till flera olika busselektorer (bus selectors), se figur 16.

Figur 16 *Busskapare respektive busselektor.*

  Ur dessa block kan användaren antingen plocka ut den eller de signaler som är intressanta. Bussblocken fungerar ungefär som vanliga vektorer, men en funktion som är särskilt fördelaktig är att användaren inte behöver använda alla signaler i en buss. Om t ex en buss skapas med fem insignaler lagras dessa i vektorformat med givna namn. Om användaren sedan behöver två av dessa signaler, sätter han denna i en busselektor och plockar ut de två intressanta signalerna. Signalerna in och ut ur bussar får dessutom vara av olika datatyper. För att komma runt hanteringsproblemet måste alla bussar i modellen ersättas med t ex muxar och demuxar, se figur 17.



Figur 17 *Mux respektive demuxblock.*

  Muxar och demuxar är block som också samlar, respektive delar upp, signaler. Problemet är att muxar fungerar precis som vanliga vektorer, vilket innebär att alla signaler som går in i en mux måste komma ut på andra sidan vid separation. Dessutom måste alla signaler som går in i en mux vara av samma datatyp, och användaren måste själv komma ihåg på vilken plats i vektorn en viss signal befinner sig. Vid utbytet av bussar mot muxar blir modellen betydligt svårare att följa.
- S-funktioner är funktionsblock som är egendesignade av användaren. Dessa block innehåller Mex-filer med C-kod eller vanliga M-filer med Matlabkod och kan heller

inte hanteras av SDV. Det finns inget generellt sätt att hantera problemet med S-funktioner, utan varje enskilt fall måste behandlas separat. Användaren får helt enkelt designa ett delsystem med block som utför samma sak som S-funktionen i fråga. Detta försvårar avsevärt verifieringen av modellen och är tidskrävande. Detta gäller särskilt om användaren som ska utföra verifikationen själv inte har programmerat S-funktionen utan först grundligt måste sätta sig in i vad som händer i funktionen.

- I SDV finns en funktion som benämns blockersättare (block-replacements). Det är en funktion genom vilken användaren kan definiera regler för att SDV ska byta ut vissa block under verifikationen. På så sätt kan man komma runt problemet med icke-stödda block. Detta är emellertid inte en funktion som kan utnyttjas för alla block utan bara om det finns något liknande block som utför samma sak och som inte kräver några andra modifieringar i modellen. Sålunda kan användaren inte använda denna funktion för att lösa de problem som beskrivits ovan, eftersom att det för att lösa sådana problem inte räcker med att byta ut ett block, utan användaren måste lägga om stora delar av modellen. Blockersättningsfunktionen kan också användas för att sätta krav eller begränsningar på olika blocks in- och utsignaler, vilket kan vara önskvärt under verifikationen.
- Problem uppstår även för vissa Stateflowdiagram. Om ett diagram innehåller ett tillstånd som bara ska vara aktivt under en begränsad tid kan SDV inte utnyttjas. Ett exempel på detta är om det finns en signal, t ex en varning, som ska slås av efter en viss tid. Verktyget kan då inte bevisa att en signal alltid är sann i detta fall. Verktyget kan inte heller producera ett motexempel utan körningen avslutas med ett meddelande som säger att egenskapsbevisningen är obestämbar (undecided). Det finns ingen universallösning på detta problem. Användaren får sålunda handskas med varje problem separat. Ett tillvägagångssätt är att verifiera eller validera Stateflowdiagrammet enskilt med hjälp av simulering eller något annat verktyg som kan hantera dessa problem. Ett annat är att användaren i stället försöker få modellen att motbevisa ett specificerat krav. Variationerna i tillvägagångssätt är betydande beroende på vilket problem som uppstått. Naturligtvis kan användaren även göra om diagrammen så problemen försvinner. Beroende på modellens storlek kan detta kräva omotiverat mycket tid.
- Förutom nämnda problem kan tillfogas att väldigt vanliga block, som till exempel integratorer och derivata, inte stöds av SDV. Dessutom hanteras endast diskret tid, inga kontinuerliga system kan behandlas.

5.2.3 Hantering av flyttal

Flyttal i Simulink kan hanteras genom exponentiell representation: $f*2^e$, där f benämns mantissa, 2 är bas och e är exponent. Detta tillåter användaren att använda flyttal. Problem kan ändå uppstå om värdena på en insignal är flytande. Då SDV utför en total tillståndsgenomgång skulle alla möjliga värden för en signal med ingångsvärde varierande mellan 0 och 100 vara omöjligt att testa. För att lösa detta problem kan användaren göra på följande sätt.

- De insignaler som består av flyttal begränsas till ett set av representativa värden, eller om det är möjligt, till en konstant. För en signal som kan variera mellan 1 och 100 kan t ex representeras av [0, 12, 21, 39, 54, 76, 99, 100]. Denna metod kallas diskretisering och är det enklaste sättet att undkomma flyttalsproblemet.
- Flyttalssignaler kan hanteras om de representeras genom en linjär olikhet, t ex $x < y$ eller $a > 0$. Däremot kan SDV inte hantera icke-linjära beräkningar.

- Om användaren inte specificerar hur verktyget ska hantera flyttal så använder sig SDV av en approximation som kallas approximering med hjälp av rationella tal (rational number approximation).

6 Testning med Reactis Validator

För att vidare undersöka potentialen för modellcheckning studerades ytterligare ett verktyg. Verktyget i fråga var ett testtäcknings- och valideringsverktyg, som inte använde sig av formell verifiering utan av simulering och testvektorgenerering. Testningsverktyget undersöktes så att en jämförelse mellan formell verifikation och sedvanlig testning kunde göras. På så sätt kunde fördelar respektive nackdelar uppfattas, dels med de två olika metoderna, men också med de undersökta verktygen. Verktyget som användes var Reactis Validator och har utvecklats av Reactive Systems Inc. Validator är designat så att användaren kan ladda upp Simulink/Stateflow-modeller för sedan applicera verktyget direkt på modellen utan vidare omarbetning eller translation, se avsnitt 4.1.3.

6.1 Grundläggande arbetsgång

- Importera önskad Simulinkmodell till Validator.
- Undersöka modellens kompatibilitet och utföra nödvändiga modellmodifieringar i Simulink. Dessa modifieringar överförs automatiskt och laddas in i Validator.
- Lägga till önskade krav och användardefinierade mål.
- Ändra eventuella inställningar för valideringen.
- Utföra valideringen och analysera resultatet som produceras.

6.2 Användarvänlighet

Verktyget är något mindre användarvänligt än SDV, och tidsåtgången för att förstå dess fulla potential är betydligt längre. Dessutom finns några element som försvårar kravvalideringen i Validator. Inget motexempel produceras vid falsifieringen av ett krav. Detta leder till att det är svårt att förstå varför ett krav falsifieras. Det är därför också svårt att förstå om det är användaren själv som har gjort något fel i utformningen av kravet eller om det är en korrekt falsifiering. Kraven uttrycks antingen som mer eller mindre enkla booleska uttryck, eller som diagramkrav designade i Simulink/Stateflow. För en förstagångsanvändare dröjer det innan uttryckssätten fulländas. Diagramkraven är dock en positiv funktion hos Validator, eftersom att de tillåter stor flexibilitet vid utformningen av krav. I princip finns det inga krav som Validator inte kan hantera.

6.2.1 Kompatibilitet

Validator hanterar inte många fler block än SDV, se appendix A2 för fullständig blockdokumentation. Däremot hanterar det ett antal mycket väsentliga block. Validator kan hantera både bussblock, S-funktioner, diskret derivata och matematiska funktioner såsom kvadratrot. Att verktyget hanterar både bussblock och S-funktioner är en betydande fördel.

6.2.2 Hantering av icke-stödda block

I likhet med SDV finns ett antal block från Simulinks bibliotek som inte stöds av Validator. Detta skapar problem för avvändaren, men samtidigt finns vissa ofta återkommande block

som Validator hanterar som SDV inte klarar av. De i detta sammanhang är bussar och S-funktioner de viktigaste. Precis som vid användning av SDV försvåras verifieringen avsevärt om modelldesignarna inte har beaktat att just detta verktyg ska användas. Nedan följer några exempel på block och funktioner som inte kan hanteras av Validator. För fullständig dokumentation av block och funktionshantering, se appendix A2. Block som inte stöds av Validator måste tas om hand på samma sätt som i SDV, d v s omkonstrueras med andra block.

- S-funktioner. Validator hanterar S-funktioner innehållande både M-filer och C-kodsfiler. Dock finns det vissa begränsningar. S-funktioner uppbyggda av M-filer får inte innehålla komplexa tal, multipla sampeltider eller Level 2 S-funktioner. S-funktioner uppbyggda av C-kodsfiler får inte innehålla komplexa tal, portbaserade eller multipla sampeltider, Level 1 S-funktioner eller anrop till någon av MATLABs Mex-filer.
- Validator hanterar inte Simulinks egna verifieringsblock såsom assertions, check dynamic eller static gain.
- Liksom i SDV är bara validering av modeller i diskret tid möjlig. Samtliga block som förutsätter kontinuitet, t ex derivata och integrator, hanteras inte av Validator.
- Validator förutsätter endast reella tal. Ingen form av komplexa tal får förekomma.

6.2.3 Hantering av flyttal

För att representera flyttal använder sig Validator av exponentiell approximation på samma sätt som i Simulink. Men eftersom approximationen är just en approximation och små avrundningsfel alltid förekommer kan värdena vid simulering skilja sig en aning mellan programmen. För att hantera insignaler bestående av flyttal använder sig Validator av samma teknik som SDV. Också sätten för att komma runt flyttalsproblemet är de samma som för SDV, se avsnitt 5.1.2 ovan.

7 Verifiering med SDV på AiCC-modell

7.1 Helhetsintryck

Generellt kan sägas att SDV verkar representera ett intresseväckande nytänkande för verifiering av modeller av detta slag. Verktyget behöver dock vidareutvecklas innan det kan användas fullt ut och kommer nog snarare att utnyttjas som ett komplement till sedvanlig testning.

En något negativ överraskning var att SDV hade svårigheter att hantera krav som var tidsberoende. Krav på att signaler ska anta ett visst värde efter en viss tid måste verifieras med hjälp av separata Stateflowdiagram, något som ökar tidsåtgången en hel del. Dessutom har SDV stora problem att verifiera bevisåtaganden som bildas på detta sätt. Av någon oförklarad anledning fungerade det ibland, men ibland inte. Detta är ett uppenbart minus, eftersom sådana krav är vanligt förekommande.

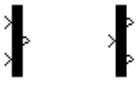
En annan uppenbar nackdel är den tid användaren måste lägga ner på att modifiera modellen för att få den kompatibel med SDV. Detta problem beror naturligtvis på hur modellen är designad och vilka block som använts. Vid beskrivet implementeringsförsök krävdes över en veckas arbete för att få modellen kompatibel.

7.2 Modellmodifiering

7.2.1 Bussar (Buses)

Bussblock är block som antingen samlar flera signaler till en enda signal eller separerar en signal till flera olika, se figur 18 nedan och avsnitt 5.1.3 ovan. Samtliga bussblock byttes ut mot mux- eller demuxblock, se figur 19, för att skapa vektorer av samtliga signaler. Detta var mycket tidskrävande, framför allt eftersom det fordras stor noggrannhet för att undvika fel. Fel uppstår lätt då man vid användning av vektorer måste hålla reda på exakt var alla signaler ligger och i vilken ordning de ligger.

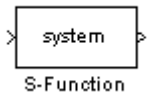
Figur 18 *Busskapare respektive busselektorblock.*



Figur 19 *Mux respektive demuxblock.*

7.2.2 S-funktioner

S-funktioner är block som innehåller en fil bestående av C-kod och är helt konstruerade av användaren, se figur 20 nedan och avsnitt 5.1.3 ovan.

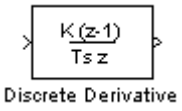


Figur 20 *S-funktionsblock.*

Omkonstruerandet av dessa block var tidskrävande, eftersom att alla olika block måste hanteras på enskilda sätt. Vissa block var svårare att göra om än andra och krävde ganska stor Simulinkvana. Vissa block kunde dock användas på flera ställen vilket minskade modifieringstiden något.

7.2.3 Diskret derivata

Diskret derivata är ett block som utför en derivering av en signal i diskret tid, se avsnitt 5.5.5 ovan och figur 21.

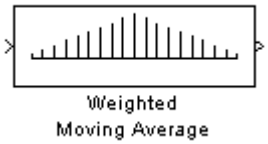


Figur 21 *Diskretderivatablock.*

Orsaken till att detta block inte stöds av SDV är att det innehåller ett mindre block, Sample *time math* som är en S-funktion. Blocket används, i detta fall, för att dividera insignalen med ett visst tidssampel och på så sätt skapa en derivata. Problem med diskret derivata uppträdde vid två tillfällen under implemenationen. Båda gångerna var problemet lätt löst, då blocket kunde bytas ut mot en enkel multiplikation inuti derivatablocket.

7.2.4 Rörligt medelvärde (Weighted moving average)

Rörligt medelvärde, se figur 22, är ett block som samplar och håller de sista n-ingångsvärdena, multiplicerar dem med ett visst värde (weight) och lägger dem i en vektor som sedan blir en utsignal. Blocket byttes ut mot ett antal fördröjningar och multiplikationer som lades ihop för att bilda den önskade utsignalen.



Figur 22 *Rörligt medelvärdesblock.*

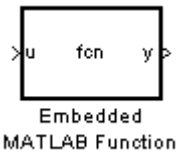
7.2.5 Matematiskfunktion: kvadratrot

SDV klarar inte av olinjäriteter, och kan således heller inte använda sig av kvadratrötter. Detta är ett problem som kan skapa betydande svårigheter, då det är en vanligt förekommande operator. Tanken fanns på att approximera kvadratroten med hjälp av Taylorutveckling, Newton-Raphsons metod eller dylikt. Detta avfärdades dock, eftersom approximeringen vid samtliga metoder måste utgå ifrån en startpunkt, som varierar avsevärt från gång till annan

beroende på en mängd olika faktorer. I slutändan skapade detta block inte några problem, eftersom blocket just i detta fall kunde beräknas manuellt och sättas till en konstant. Blocket ledde dock till problem senare i processen då en inbäddad Matlabfunktion skulle användas, se avsnitt 7.2.6 nedan.

7.2.6 Inbäddad Matlabfunktion (Embedded Matlab Function)

Dessa block, se figur 23, är uppbyggda av funktioner skrivna i Matlab och kan vara speciellt användbara vid långa och komplicerade beräkningar som kan vara tidskrävande och svåra att konstruera i blockdiagramform. Dessvärre stöds inte dessa block av SDV. Vid implementeringen fick sålunda ett sådant block tas bort ut verifieringen och ersattas av vanliga insignaler. Detta inbyggda beräkningsblock verifierades alltså överhuvudtaget inte.



Figur 23 *Inbäddad Matlabfunktion.*

Ett försök att bygga om hela beräkningsblocket m h a vanliga Simulinkblock utfördes under implementeringen, men misslyckades då funktionerna i några fall innehöll kvadratrotsberäkningar, se 7.2.5 ovan, och därför inte kunde användas.

7.3 Verifiering

Sedan lång tid avsatts åt att få AiCC-modellen kompatibel med SDV kunde verifieringen slutligen genomföras. Totalt fem krav falsifierades under verifieringen.

Nedan följer en redovisning av de problem som uppkom under verifieringen. Vidare uppmärksammas vissa aspekter som är relevanta för framtida användare.

7.3.1 Stateflow

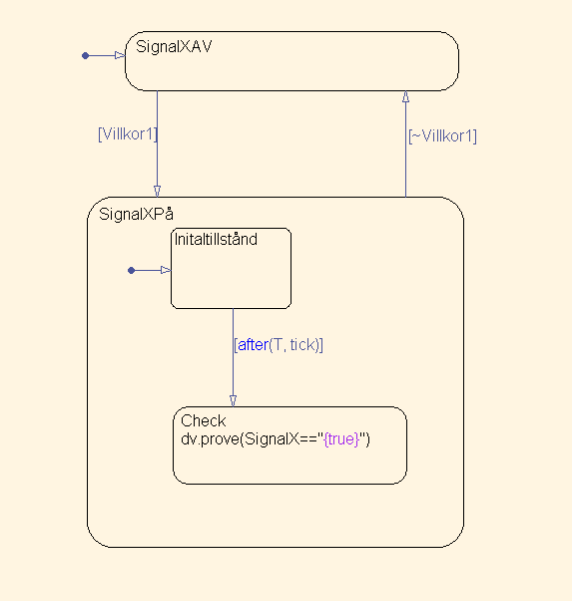
Problem uppstod vid verifiering av signaler som passerade genom Stateflowdiagram. Detta berodde på att Stateflowdiagrammen var uppbyggda så, att inkommande signal alltid gick in ett tillstånd där signalen var lika med noll. Det innebär att ett steg alltid behövs innan en signal som t ex ska vara ett verkligen blir ett. Vid användandet av SDV sätts krav för signaler som innebär att de alltid ska anta ett visst konstant värde. Om en signal således alltid ska anta ett värde skilt från än noll blir detta ett problem om Stateflowdiagrammet alltid antar att signalen är noll från början.

För att lösa detta gäller att den som designar Stateflowdiagrammen måste vara medveten om problemet och designar sina diagram så att de inte utgår ifrån att en signal alltid har startvärde noll.

I den utsträckning det var nödvändigt för att verifieringen skulle fungera modifierades Stateflowdiagrammen enligt ovan nämnda kriterier och sedan designades även egna Stateflowdiagram för hantering av temporala krav, se avsnitt 7.3.2 nedan.

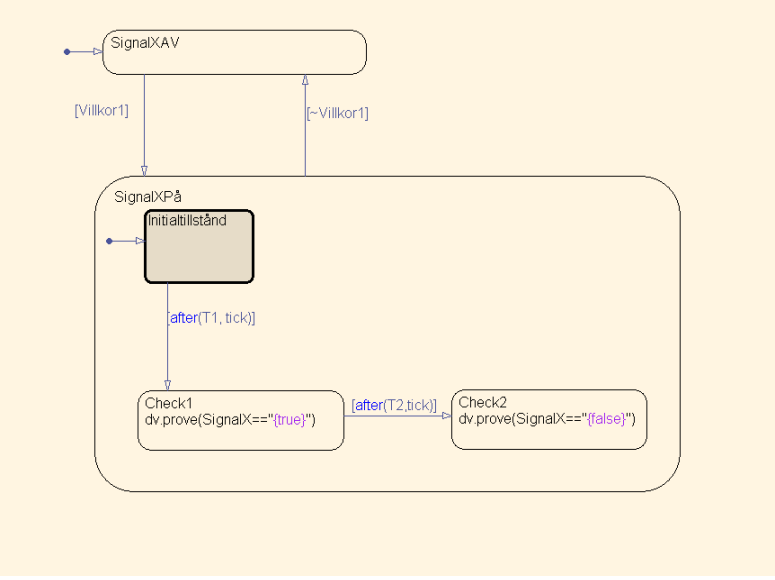
7.3.2 Uttrycka temporala krav i Stateflow

För att kunna ta hand om problemen med signaler som tar en viss tid att anta ett visst värde designades separata Stateflow modeller. Se figur 24 nedan.



Figur 24 *Stateflowdiagram för uttryckandet av temporala krav.*

Diagrammet i figur 24 ovan visar att när Villkor1 är uppfyllt intas tillståndet SignalXPå, och efter T sampel kommer SDV att utföra bevisåtagandet: SignalX == sant. På detta sätt kan användaren hjälpa SDV att hantera signaler som ska ändra värde efter en viss tid, eller ett visst antal sampel. Diagrammet kan även byggas ut för signaler som ska byta värden mer än en gång, se figur 25.



Figur 25 *Stateflowdiagram för verifikation av en signal som ska sättas på efter T1 sampel och som sedan ska stängas av och förbli avstängd efter T2 sampel.*

Detta tillvägagångssätt är särskilt användbart vid verifiering av temporala krav och är mer tillförlitligt än att använda simulering som hjälpmedel, se avsnitt 7.3.3 nedan. Det enda användaren måste tänka på är att om detta Stateflowdiagram finns i ett biblioteksblock och sedan läggs in i modellen måste länken mellan modellen och biblioteket brytas (disable link) för att verifieringen ska fungera. Vad detta beror på är ännu oklart, men resultatrapporten produceras inte som den ska och verifieringen blir därför ofullständig.

Ett problem uppstod också vid verifiering på ovan nämnda vis. Stateflow tycks inte alltid kunna hantera signaler som ska anta ett visst värde efter en längre tid, d v s efter ett större antal sampel. Ett exempel på detta uppstod när en signal skulle anta värdet -2.5 efter 1 sekund, d v s efter 100 sampel (varje sampel i modellen var lika med 10ms). Vid verifieringen klarade inte SDV att komma fram till någon slutsatts inom angiven tid (20 min) och vid ett senare försök producerade SDV odefinierade felmeddelanden och avbröts. Detta problem uppkom flera gånger under verifieringen men det var likväl inget konsekvent fel, eftersom de fungerade utmärkt för vissa krav. Vad detta berodde på kunde aldrig förklaras. Undersökningar gjordes för att se om problemet utgjordes av att modellen var för stor. Verifieringen genomfördes i stället på mindre delar av modellen. Det visade sig då att modellstorleken inte hade någon betydelse, utan verifieringen misslyckades även vid ett sådant tillvägagångssätt.

7.3.3 Simulering som hjälpmedel

Vid verifiering av signaler som varierar inom ett intervall är det av nytta att kombinera verifieringen med simulering. När krav falsifieras av SDV producerar verktyget en s k harnessmodell, som är en kopia av modellen. Denna kopia kan användaren utnyttja för simuleringar i syfte att t ex se hur signaler varierar och dylikt. Vid ett krav på att en signal ska anta ett visst värde efter en viss tid kan simulering vara ett användbart alternativ om användaren inte har möjlighet att verifiera m h a Stateflow eller om detta av någon anledning inte fungerar.

7.3.4 Delsystem

För att underlätta verifieringen finns det möjlighet att bygga egna delsystem och verifiera dessa separat, se avsnitt 4.1.1 ovan. Vid implementeringen uppvisade detta tillvägagångssätt både fördelar och nackdelar.

Vid designandet av verifieringsfall har användaren möjlighet att sätta antaganden på signaler för att visa att ett visst bevisåtagande håller. Dessa antaganden kan endast sättas på modellens insignaler, alltså inte på signaler som uppträder mitt i modellen. För att sätta antaganden på signaler mitt i modellen måste användaren sätta antaganden redan på de insignaler som gör att en signal senare i modellen antar ett visst värde eller intervall av värden. Detta gör att användaren med säkerhet måste veta hur olika signaler påverkar varandra.

Det positiva med delblocksverifiering är att processen kan förenklas genom att användaren kopierar den del av modellen som ska verifieras och lägger in den i ett verifieringsdelblock. På så sätt erhåller användaren de signaler som behövs för verifieringen som insignaler och antaganden kan sättas som önskat.

Det negativa med verifiering av delblock är att krav som bevisats sanna inom ett delblock inte alltid är sanna vid verifiering av hela modellen. Det finns betydande risker med att verifiera

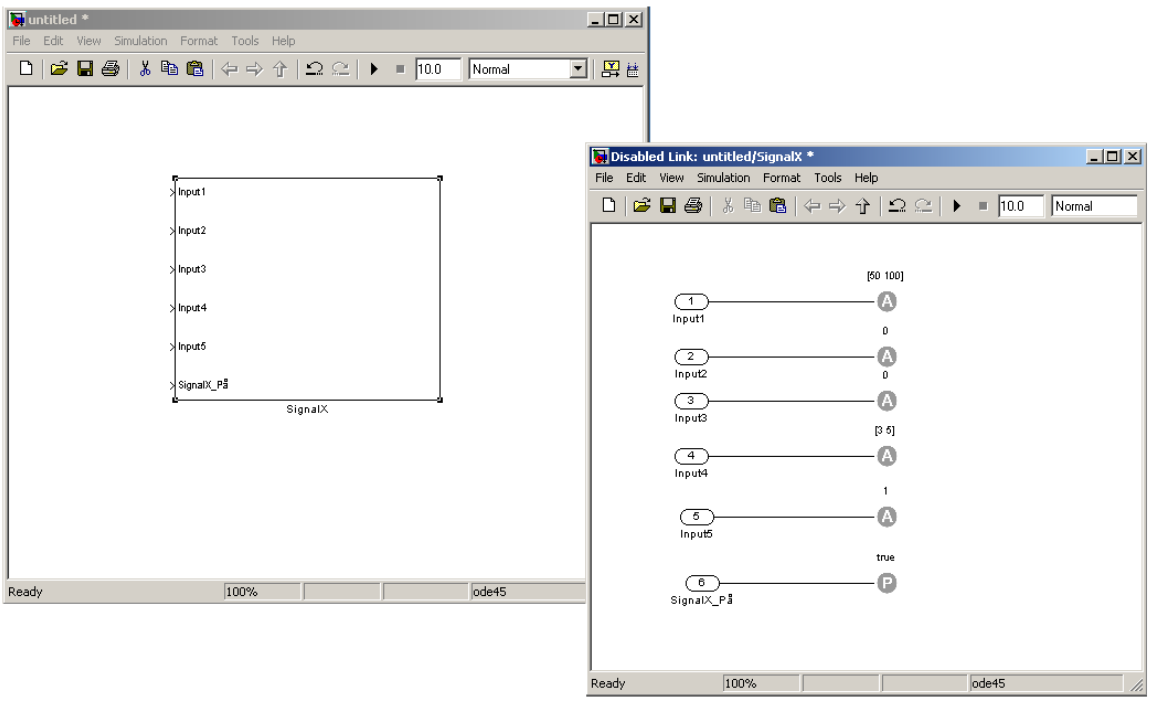
egenskaper på detta sätt, då fel kan uppstå på vägen genom modellen vilket SDV inte har möjlighet att upptäcka.

7.3.5 Lagring av verifieringsfall

Till skillnad mot testning, där olika testfall skrivs separat och sedan sparas utanför den modell som ska valideras, sker modellcheckningen med SDV direkt i modellen. Detta medför att vid varje verifieringsfall måste användaren manuellt ändra villkor och bevisåtaganden i modellen, vilket leder till att det blir svårt att spara och dokumentera varje enskilt verifieringsfall.

För att kunna spara varje verifieringsfall på ett effektivt sätt kan användaren skapa ett bibliotek med egna delblock innehållande villkor och bevisåtagande för varje enskilt krav, se figur 26 nedan. På så sätt blir det enkelt för användaren att återgå till krav som verifierats vid tidigare tillfällen.

I figur 26 nedan visas ett delblock som ligger i ett skapat bibliotek. Detta block innehåller villkor på de fem översta insignalerna, och ett bevisåtagande på insignalen är placerad längst ner. För att sedan använda blocket för verifiering av specificerade krav placeras biblioteksblocket på valfri plats i modellen där det finns möjlighet att koppla samman blocket med nödvändiga insignaler.



Figur 26 *Biblioteksblock för sparande av villkor och bevisåtagande för ett visst verifieringsfall. Bilden till höger är innehållet i blocket till vänster.*

Verifieringen av AiCC-modellen upprepades, och för varje krav sparades ett eller flera block i ett bibliotek för att sedan kunna användas på nytt om nödvändigt. Denna process skulle kunna underlättas m h a ett Matlab-skript med förmågan att kopiera in biblioteksblocken i modellen och att köras automatiskt. Detta undersöktes dock inte närmare i detta examensarbete.

7.3.6 Odefinierade fel

Vid implementeringen uppkom vid flera tillfällen felmeddelanden som inte specificerade själva felet, vilket var otillfredsställande. Efter diverse undersökningar framkom att felmeddelandena kan uppkomma bland annat av följande anledningar.

- Vid felaktigt specificerade antaganden.
- Vid felaktig kompilering eller översättning av modellen. I detta fall rekommenderas ny körning. I sådana fall har inte användaren gjort något fel, utan bristerna finns i verktyget. Sådana fel uppstår oftast när Stateflowdiagram är involverade.
- Vid sättning av antaganden på signaler som inte är insignaler till modellen, eller delmodellen om denna sorts verifiering tillämpas.

7.4 Simulink Design Verifier version 2.1 för Matlab 2008a

Verifikationen utfördes med hjälp av Simulink Design Verifier version 1.1 för Matlab 2007b. Det har sedermera lanserats en ny version, 1.2 för Matlab 2008a, som innehåller en del uppdateringar bland annat i form av fler stödda block och funktioner.

7.4.1 Förbättringar

- Rörligt medelvärde: detta block har tagits bort ut Simulinks blockbibliotek och har ersatts av ett diskret FIR Filterblock. Detta block stöds av SDV.
- Kvadratrot: Ingår som parameter i blocket matematisk funktion. För detta block är parametrarna kvadratrot och komplex konjugat kompatibla med SDV, men endast för signaler med hel- eller flyttal som in- och utsignal.
- Inbäddad matlabfunktion: detta block fungerar utmärkt med SDV för Matlab 2008a, dock med vissa begränsningar. T ex kan SDV fortfarande inte hantera beräkning av kvadratrötter eller andra exponentialfunktioner.
- Stateflow: förbättringar har gjorts för att SDV ska kunna hantera Stateflowdiagram i större utsträckning. Framför allt har framsteg gjorts vad gäller hantering av sanningstabeller (truth tables) i Stateflow.

För komplett dokumentation av förbättringar och blockstöd se www.mathworks.com/releasenotes.

8 Testning med Validator på AiCC-modell

8.1 Helhetsintryck

Redan från första användandet av Reactis Validator fås intrycket att verktyget är något mer gediget än SDV. Verktygen skiljer sig åt på många sätt, men först märks hur mycket mindre tid det krävs för att få den aktuella modellen kompatibel med Validator jämfört med SDV. Det skilde flera dagar i nerlagt arbete för modellmodifieringen.

Validator är mer komplicerat och mindre användarvänligt än SDV. Det krävs lite mer tankeverksamhet vid verifieringen med Validator än med SDV. I stället för att bara lägga in block på önskade ställen måste användaren antingen skriva mer eller mindre komplicerade booleska uttryck för kraven eller designa kravdiagram i Simulink/Stateflow. Trots att det kan vara komplicerat att konstruera ett önskat krav är likväl flexibiliteten i kravhanteringen ett stort plus för Validator: Det finns i det närmaste inget krav som verktyget inte kan hantera. Möjligheten att specificera ett visst krav beror snarare på användarens kompetens än på verktyget.

Validator är sålunda inget verktyg för formell verifiering. Detta leder till att resultaten av kravvalideringen ibland kan tyckas ganska godtyckliga beroende på graden av testtäckning. Låg testtäckning leder till att användaren inte kan anse kravresultaten vara särskilt tillförlitliga.

8.2 Modellmodifiering

I motsatts till SDV krävdes ytterst marginella modifieringar för att få AiCC-modellen kompatibel med Validator.

8.2.1 Rörligt Medelvärde (Weighted moving average)

Blocket för rörligt medelvärde var ett av två block som behövdes bytas ut för att modellen skulle fungera med Validator. Detta block byttes ut på samma sätt som vid användningen av SDV, se avsnitt 7.2.4 och figur 22 ovan.

8.2.2 Inbäddad Matlabfunktion (Embedded Matlab function)

Blocket för Inbäddad Matlabfunktion var det andra av totalt två block som behövdes ersättas för att verifieringen med Validator skulle fungera. Blocket, som innehåller en M-fil, se avsitt 7.2.6 och figur 23 ovan, byttes ut till ett egendesignat delblock innehållande block från Simulinks bibliotek vilket utförde samma beräkningar som originalblocket.

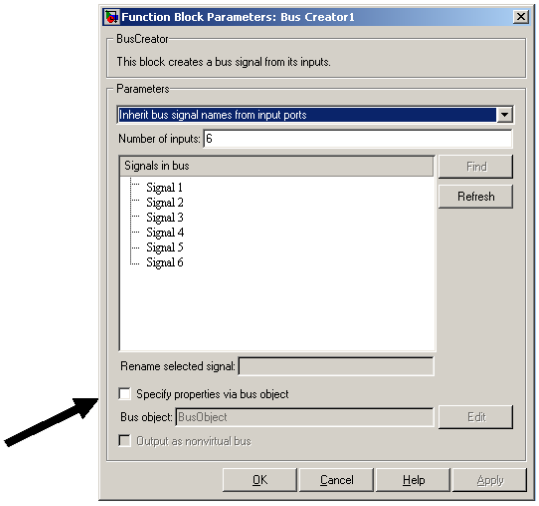
8.2.3 Övriga modifieringar

Förutom ovan nämnda block behövdes smärre justeringar för att Validator skulle fungera. Validator klarar inte av att hantera sant och falskt utan ser dessa som vilka variabler som helst. För att lösa detta deklarerades sant och falskt till 1 respektive noll i en M-fil som sedan tidigare innehöll alla konstanter tillhörande modellen, se figur 27.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%
% true och false
%%%%%%%%%%%%%%%%%%%%%%%%%%
true = boolean( 1);
false = boolean( 0);
```

Figur 27 *Deklaration av sant och falskt i M-fil.*

För att busskaparna skulle fungera i Validator behövdes en liten modifiering. Utsignalen från bussbyggaren fick inte ha rutan *Specify properties via bus object* ikryssad, se figur 28. Detta problem åtgärdades enkelt.



Figur 28 *Parametrar hos en busskapare.*

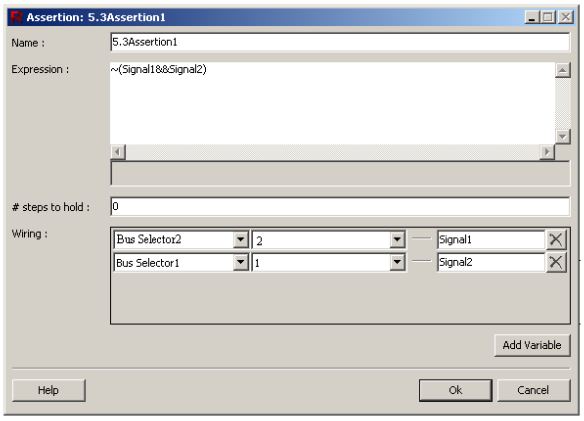
För övrigt måste användaren explicit lägga till kompletta sökvägar till alla filer som används av den validerade modellen. Det räcker alltså inte att filerna ligger i samma mapp i Matlab. Sökvägarna till filer innehållande bibliotek och konstanter mm läggs till under *search paths* i Validators programfönster.

8.3 Valideringen

Efter diverse små modifieringar av AiCC-modellen kunde valideringen genomföras. Valideringen utfördes först för alla enskilda krav, d v s ett krav för varje körning. Sedan utfördes även körningar på flera krav samtidigt, slutligen på samtliga krav samtidigt. Under dessa första körningar behölls modellen helt intakt. Körningar utfördes sedan på modellen uppdelad i tre delmodeller för att undersöka om testtäckningen kunde förbättras.

8.3.1 Uttryckskrav/uttrycksmål

De flesta kraven som fanns på AiCC-modellen var krav som gick att konstruera med hjälp av uttryckskrav respektive uttrycksmål. Dessa var lätta att konstruera och lades direkt in i den importerade modellen. En svårighet med dessa krav är att försäkra att de uttrycker precis det önskade. Detta förutsätter viss erfarenhet av arbete med logiska uttryck och vissa svårigheter uppkom pga detta under arbetets gång. Se figur 29 för dialogrutan där uttryckskraven skrevs in och sedan lades in i modellen.



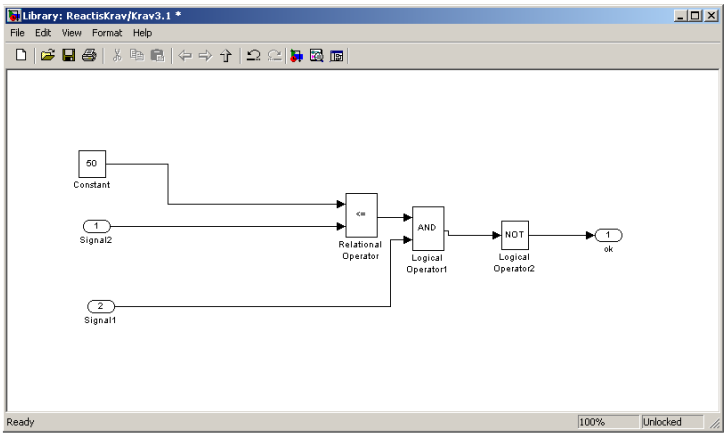
Figur 29 *Dialogruta för uttryckskrav.*

En fördel med uttryckskraven i Validator är att det finns möjlighet för användaren att bestämma att ett krav måste vara falskt i ett vist antal sampel innan kravet anses som falsifierat. Detta underlättar då det kan finnas signaler som kräver ett visst antal sampel innan de aktiveras. Som framgår av tidigare avsnitt fanns i undersökt modell Stateflowdiagram för vissa varningar som hade begynnelsevärde lika med noll. Dessa ställde till stora problem för SDV, eftersom att det tog ett sampel för varningen att aktiveras, se avsnitt 7.3 ovan. Validator hanterar sådana krav utan svårigheter.

En nackdel med uttryckskraven är att de är designade så, att de bara hanterar krav som alltid är sanna för modellen. Det går till exempel inte att bilda uttryckskrav som bara ska stämma under vissa bestämda omständigheter.

8.3.2 Diagramkrav/diagrammål

Diagramkraven och diagrammålen ger Validator en betydande flexibilitet, eftersom de tillåter användaren att tillverka diagram för i princip vilket krav som helst. Diagrammen tillverkas som ett vanligt Simulink eller Stateflowdiagram och sparas i en separat fil som sedan inkorporeras i den importerade Simulinkmodellen, se figur 30.



Figur 30 *Exempel på krav designat i Simulink.*

Figuren ovan beskriver ett krav om Signal 1 <= 50 ska Signal 2 vara = 0. Utsignalen kommer att visa sant om detta alltid stämmer och falskt i annat fall. Diagramkraven/målen ska alltid vara så uppbyggda att utsignalen visar sant då ett krav är uppfyllt och falskt annars.

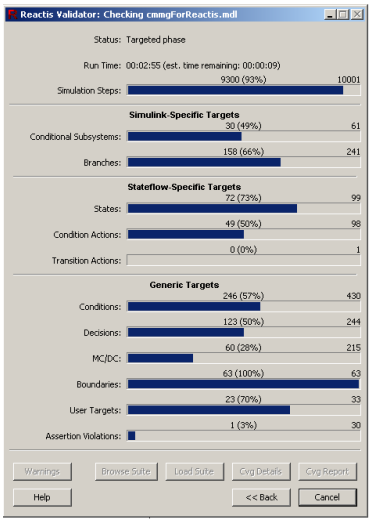
8.3.3 Simulering som hjälpmedel

Validator är ett simuleringsprogram och detta är något som användaren kan utnyttja som hjälpmedel vid valideringen. Om ett krav falsifieras finns det möjlighet att köra simuleringar på de testfall som Validator genererat under körningen. Vid simuleringen av givet testfall kan användaren undersöka vilka delar av modellen som exekverats och därigenom försöka förstå varför ett krav täckts eller ett mål inte täckts o s v. Då ett enskilt motexempel inte produceras som förklaring, som i SDV, var det mycket svårt att faktiskt förstå varför ett krav täckts. Simuleringen var under detta arbete inte till någon hjälp, utan ofta letades felaktigheter i total blindo.

8.3.4 Testning på komplett modell

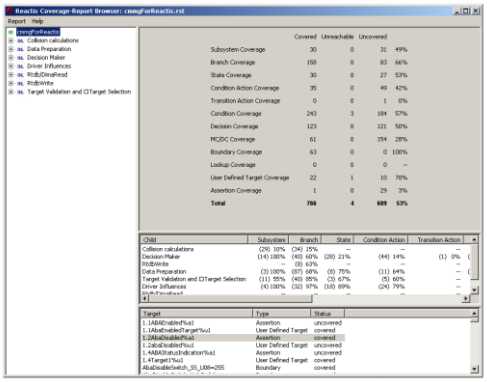
Först utfördes körningar på alla kraven, ett åt gången och till slut på hela modellen. Vid dessa körningar blev resultaten identiska med dem som uppkom vid verifieringen med SDV. Testtäckningen var dock överraskande låg, MC/DC låg runt 30 % eller strax därunder för samtliga körningar. Täckningsgraden för villkor respektive beslut låg på ca 50-60 %. Detta tydde på att valideringen av kraven inte var särskilt tillförlitliga eftersom att så stora delar av modellen inte undersökts. Hela modellen var dessutom kravställd före testningen, vilket innebär att högre testäckning borde vara möjlig. Förutom testtäckningen var också antal täckta mål väldigt lågt vilket också antydde att valideringsresultaten inte var tillförlitliga. Om ett krav och ett mål skapas innehållande samma uttryck, t ex ~(Signal1 && Signal2), och varken kravet eller målet täcks betyder detta att Validator inte har hittat ett testfall där kravet falsifierats. Att kravet inte täckts betyder visserligen att inget fel har hittats, men eftersom målet heller inte täckts har ett testfall innehållande befintligt krav heller inte funnits. Målen är alltså avgörande för att användaren ska kunna försäkra sig om att rätt testfall har skapats och därigenom en tillförlitlig kravtäckning uppnåtts.

Genom en dialogruta kan körningens förlopp följas. Här visas återstående tid, olika testtäckningar inklusive MC/DC, besluts- och villkorstäckning samt täckningen för krav och uppsatta mål, se figur 31.



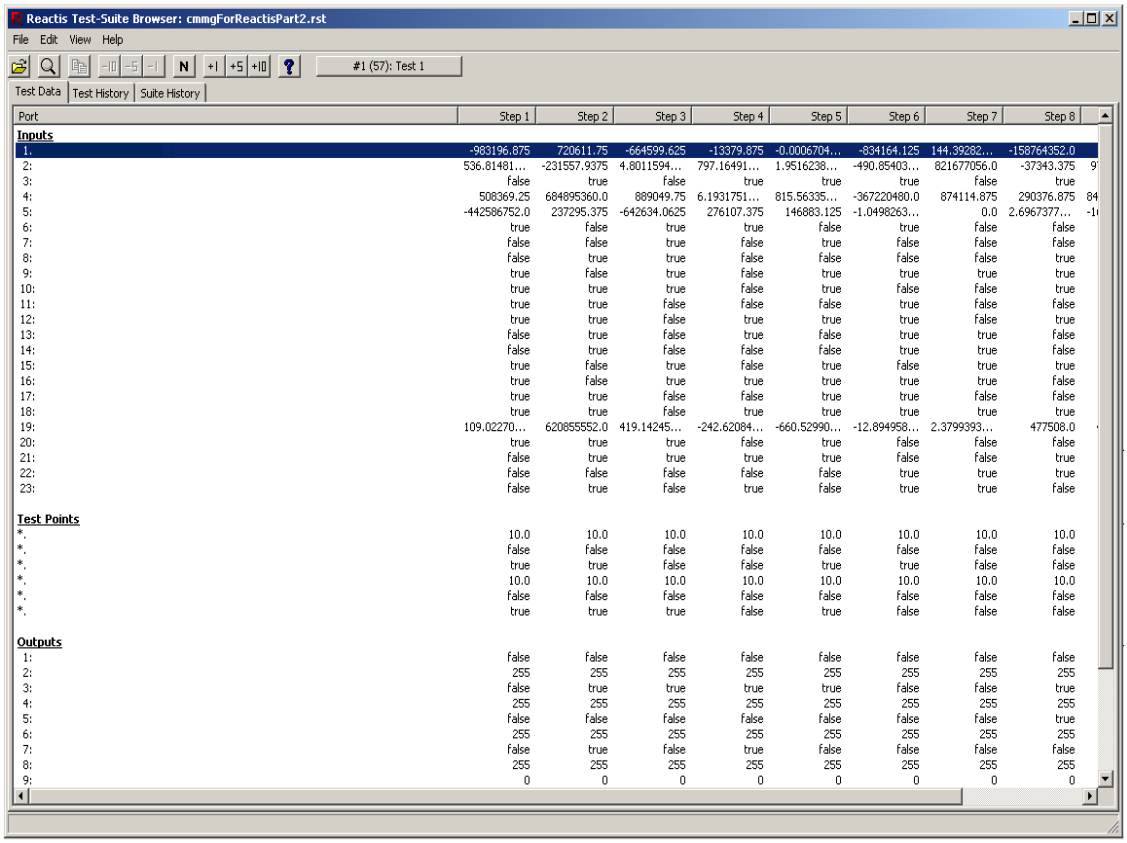
Figur 31 *Dialogruta som visas under testningens gång.*

När körningen är färdig produceras en rapport innehållande testäckning, täckning av krav och mål mm. Här kan användaren gå in i modellen och se vilka specifika krav och mål som täckts respektive inte täckts. Denna kan även se vilka delar av modellen som exekverats eller ej, se figur 32.



Figur 32 *Testrapport producerad av Validator.*

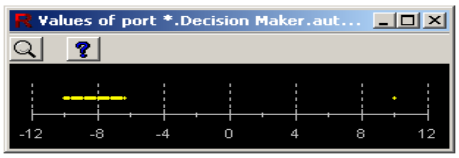
Validator producerar också en rapport där alla testfall sparats. Här visas samtliga värden på insignaler och utsignaler som använts under körningen, se figur 33.



Port	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
Inputs								
1:	-983196.875	720611.75	-664599.625	-13379.875	-0.0006704...	-834164.125	144.39282...	-158764352.0
2:	536.81481...	-231557.9375	4.8011594...	797.16491...	1.9516238...	-490.85403...	821677056.0	-37343.375
3:	false	true	false	true	true	true	false	true
4:	508369.25	684895360.0	889049.75	6.1931751...	815.56335...	-367220480.0	874114.875	290376.875
5:	-442586752.0	237295.375	-642634.0625	276107.375	146883.125	-1.0498263...	0.0	2.6967377...
6:	true	false	true	true	false	true	false	false
7:	false	false	true	false	true	false	false	false
8:	false	true	true	false	false	false	false	true
9:	true	false	true	false	true	true	true	true
10:	true	true	true	false	true	false	false	true
11:	true	true	false	false	false	true	false	false
12:	true	true	false	true	true	true	false	true
13:	false	true	false	true	false	true	true	false
14:	false	true	false	false	false	true	true	false
15:	true	false	true	false	true	false	true	true
16:	true	false	true	true	true	true	true	false
17:	true	true	false	false	true	true	false	false
18:	true	true	false	true	true	true	true	true
19:	109.02270...	620855552.0	419.14245...	-242.62084...	-660.52990...	-12.894958...	2.3799393...	477508.0
20:	true	true	true	false	true	false	false	false
21:	false	true	true	true	true	false	false	true
22:	false	false	false	false	false	true	true	true
23:	false	true	false	true	false	true	true	false
Test Points								
*:	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
*:	false	false	false	false	false	false	false	false
*:	true	true	false	false	true	true	false	false
*:	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.0
*:	false	false	false	false	false	false	false	false
*:	true	true	true	false	true	false	false	false
Outputs								
1:	false	false	false	false	false	false	false	false
2:	255	255	255	255	255	255	255	255
3:	false	true	true	true	true	false	false	true
4:	255	255	255	255	255	255	255	255
5:	false	false	false	false	false	false	false	true
6:	255	255	255	255	255	255	255	255
7:	false	true	false	true	false	false	false	false
8:	255	255	255	255	255	255	255	255
9:	0	0	0	0	0	0	0	0

Figur 33 *Testfallsrapport, här kan användaren se värden på alla in- och utsignaler som genererats under valideringen.*

Användaren har också möjlighet att lägga in testpunkter i modellen innan körningen startas om hon är intresserad av värdet på någon specifik signal som inte är in- eller utsignal. Till både testpunkter och in- och utsignaler i de olika testfallen finns diagram som användaren kan öppna och utnyttja för att analysera fördelningar av värden på signalerna, se figur 34.



Figur 34 *Fördelningsdiagram för värden vid en given testpunkt. I detta fall ligger ett stort antal av signalens värden mellan -10 och -6, och sedan ett värde vid +10.*

Användaren har även möjlighet att ladda upp de testfall som genererats under valideringen för att sedan simulera dem. På så sätt kan man undersöka varför ett givet krav falsifierats, men även för att närmare granska testäckningen. Vid simuleringen färgsätts de olika blocken och signalerna för att visa vilka delar som täcks eller inte.

De enskilda kraven sparades i separata s k modellfiler, och valideringen kördes sedan på de enskilda filerna. En stor fördel var att användaren kunde ha flera modellfiler öppna samtidigt och dessutom modifiera och spara efter behag under körningarnas gång. Validator kräver alltså inte att programmet förblir oanvänt under exekveringen, som t ex SDV. Då valideringen kunde ta upp emot 10 minuter beroende på programinställningarna sparades avsevärd tid.

Efter att ha testat alla krav separat i modellen grupperades kraven efter områden i åtta modellfiler för att därefter köras. Genom detta förfarande kunde eventuell förekomst av motsägande krav valideras samtidigt. Då detta fungerade bra och ingenting tycktes försämras lades samtliga krav in i en och samma modellfil och kördes vid ett tillfälle. Inga stora skillnader i krav, mål eller testtäckning fanns vid de olika körningarna. Totalt falsifierades fem krav, d v s precis som vid verifieringen med SDV.

8.3.5 Testning för förbättring av testtäckning

De tester som utfördes på hela modellen gav mediokra resultat för testtäckningen, något som också ledde till icke-tillförlitliga kravresultat. För att försöka förbättra testtäckningen utfördes ett flertal modifieringar. Samtliga försök som förklaras nedan utfördes på en modellfil innehållande samtliga krav.

1. Värdena på insignalerna ändrades och sattes till vad som ansågs vara lämpliga. Ett flertal olika försök utfördes med olika värden på insignalerna, men detta resulterade i att testtäckningen, liksom måltäckningen, försämrades betydligt. MC/DC sjönk till strax under 20 % och besluts- och villkorstäckningen sjönk till drygt 35 %. Inga skillnader upptäcktes vad gällde kravtäckningen.
2. Inställningarna för testningen ändrades för att undersöka om antalet steg och testfall kunde påverka resultatet. Test utfördes för både väldigt korta simuleringar och väldigt långa simuleringar. Några nämnvärda förändringar noterades inte, inte heller för testtäckningen.
3. Sista ansatsen var att dela upp modellen i tre mindre delar och sedan utföra tester på delarna, innehållande respektive krav. Detta gav stora resultat, för modelldel nummer två. Testtäckningen och måltäckningen ökade markant, men samtidigt erhölls liknande resultat även för kravtäckningen. Ett stort antal krav som tidigare satisfierats falsifierades sålunda, se tabell 6. För modelldel ett och modelldel tre förbättrades testäckningen i betydligt mer begränsad utsträckning, men inga nya krav falsifierades. Täckta mål var ungefär de samma som vid tidigare validering, viket sannolikt tyder på att resultaten från kravtäckningen var förhållandevis godtyckliga. Med stor sannolikhet skulle även ett flertal krav hos dessa modelldelar falsifieras om testtäckningen ökade.

Tabell 6 *Valideringsresultat för uppdelad modell.*

Modelldel	Täckta krav	Täckta mål	MC/DC (%)	Beslutstäckning (%)	Villkorstäckning (%)
1	0/4	2/4	52	71	88
2	14/20	19/20	81	90	93
3	1/5	7/8	53	59	70

För att vidare undersöka de tre delmodellerna sattes även här insignalerna till specifika värden eller intervall av värden. Kravtäckningen förbättrades då, men till priset av sämre testtäckning och även sämre måltäckning vilket sammantaget medförde att valideringsresultaten blev mindre tillförlitliga.

8.3.6 Förbättring av kravvalideringen

I syfte att reducera antalet krav som falsifierats under valideringen av den uppdelade modellen undersöktes konstruktionen av samtliga krav. Flertalet försök gjordes för att omformulera uttrycken, och även diagramkrav byggdes. Den begränsade tid som stod till förfogande tillät inte mer omfattande försök, och några resultatförbättringar uppnåddes inte.

8.3.7 Felaktigheter

Vid ett flertal tillfällen under valideringen uppstod fel på insignalernas datatyper. Orsaken till detta klargjordes inte, eftersom problemet var lätt att lösa genom en enkel knapptryckning: *port types* och sedan *synchronize* i Validators programfönster.

Fel uppkom också då ett diagramkrav innehållande ett Stateflowdiagram skulle läggas in i Validator-modellen. Ett felmeddelande om att en .mex fil inte kunde tas bort uppkom, trots att denna .mex fil faktiskt togs bort av Validator. Vad detta berodde på kunde inte förklaras. Det var inte möjligt att använda något diagramkrav innehållande Stateflowdiagram under implementeringen.

Validator hade vid ett flertal tillfällen stora problem med att köras då Matlab var i gång parallellt. Validator frös gång på gång, men om detta berodde på Validator eller på att datorn som användes hade för liten kapacitet förblev oklart. Detta var dock störande, eftersom Matlab kontinuerligt användes parallellt med Validator under implementeringen. Dessutom krävde Validator relativt lång tid för nedstängning och omstart.

9 Slutsatser och diskussion

Det finns fortfarande vissa oklarheter inom området modellcheckning och formell verifikation efter detta examensarbetes slutförande. Det finns alltjämt flera aspekter som borde studeras och utvärderas innan några konkreta och definitiva slutsatser kan dras. Inom industrin är formell verifiering en ny och spännande metod för att bättre och snabbare finna fel i system. Även om adekvata verktyg ännu inte finns är tekniken intressant om ännu inte helt mogen för att marknadsföras.

9.1 Testning eller Formell Verifiering?

Testning och formell verifiering är två olika metoder för att undersöka ett systems korrekthet. Teknikerna är sinsemellan mycket olika. Det är svårt att med någon säkerhet hävda att den ena metoden är avsevärt bättre än den andra. Testning är en väl använd och beprövad metod som fungerar bra trots att omfattande testning aldrig kan leda till absolut visshet om att ett system är hundraprocent korrekt. Produkter utvecklas i allt snabbare takt, och blir mer och mer komplicerade. Tiden för validering och verifiering krymper oavbrutet beroende på företagens vilja att produkter så snabbt som möjligt ska lanseras på marknaden. Detta leder i sin tur till att validering och verifiering måste utvecklas för att den ska ta kortare tid utan att det slutgiltiga resultatet påverkas negativt. Utvecklingen har påskyndats genom automatisering av existerande testningsprocesser, men nyare metoder kan emellanåt vara att föredra. Olika former av formell verifiering kan definitivt vara en väg att gå för industrin, särskilt som den modellbaserade utvecklingen ökar och enklare modellcheckningsverktyg finns att tillgå eller är under utveckling. Det är svårt att med någon större säkerhet förutsäga vilken väg utvecklingen kommer att ta, men under överskådlig tid är en kombination av de båda teknikerna att föredra. Det rör sig om två olika tekniker som sannolikt kommer att visa sig ha sina starkare och svagare sidor. Sålunda kan vissa krav vara alltför komplicerade för att med lätthet kunna uttryckas hos ett modellcheckningsverktyg. I sådana fall erbjuder testning ett bra komplement. Testning kan också användas som grundmetod, medan modellcheckning kan användas för vissa specifika, säkerhetskritiska krav som är vitala för modellen och som absolut inte får visa sig felaktiga.

9.2 Verktygen

De två verktygen som undersökts, SDV och Reactis Validator, har båda sina för- och nackdelar, vilka beskrivits ovan i avsnitt 5 och 6. Som helhet kan sägas att Validator är det entydigt mest utvecklade verktyget även om det är svårt att uppnå riktigt hög testtäckning. Validators stora fördel är att det klarar av viktiga Simulinkblock och har ett flertal synnerligen användbara funktioner. Sålunda är det okomplicerat att lägga in testpunkter, så att användaren lätt kan undersöka och analysera alla signaler som finns i modellen. Detta är inte lika lätt i SDV där användaren måste lägga in diagram länkade till signalen och sedan köra en simulering av testharnessen som produceras vid verifieringen. Validator är svagare i användarvänlighet där SDV vinner överlägset. Samtidigt är SDV väldigt fyrkantigt och icke-flexibelt jämfört med Validator. Användarvänligheten är egentligen SDVs enda uppenbara fördel. Verktyget behöver utvecklas ännu mycket mera för att kunna anses vara ett lämpligt och användbart verktyg. Särskilt måste SDV utvidga blockhanteringen som för tillfället är påfallande undermålig. Verktyget är också påtagligt underutvecklat och borde enligt min mening inte ha lanserats överhuvudtaget. Med så stora modellmodifikationer som behövs är det omöjligt att använda det på ett sätt som garanterar tillförlitlig verifiering. Det bör dock tilläggas att SDV har stor potential om blockhanteringen - och även metoder för att kunna

uttrycka mer komplicerade krav – utvecklas. The Mathworks ska ändå ha erkännande för att de vågat pröva någonting helt nytt. Det negativa med Validator är att resultaten av kravställningen känns uppenbart godtycklig och varierar betydligt mellan körningarna. Dessutom fungerar Validator illa även på relativt små modeller, vilket leder till mycket extra arbete för användaren. Det bör sannolikt finnas andra verktyg som är bättre än de båda som använts och undersökts i detta examensarbete.

9.2.1 Sammanfattande fördelar och nackdelar med verktygen

Simulink Design Verifier

Fördelar

- Även om formell verifikation inte är en ny företeelse, representerar det något nytt inom den modellbaserade utvecklingen, och nyheter är intressanta.
- Enkelt verktyg, innehåller inga onödiga valmöjligheter eller funktioner som försvårar utnyttjandet.
- Användarvänligt och lätt att lägga in krav och bevisåtaganden i modellen.
- Klarar tillfredsställande stora modeller. Inga uppdelningar krävdes.
- Snabb verifiering, oftast klar på några sekunder.
- Verifieringsresultaten är tillförlitliga och beror ej av andra parametrar som t ex testtäckning eller dylikt.
- Lätt att spara skapade verifieringsblock i bibliotek för dokumentation och senare användning.
- Verktyget är under ständig utveckling och kommer förhoppningsvis inom en snar framtid att vara bättre och framförallt kunna hantera fler viktiga block.

Nackdelar

- Alldeles för många Simulinkblock som ej är kompatibla med SDV. Otillfredsställande att verktyget inte klarar av t ex bussblock som är särskilt användbara. Kräver allt för stora modifikationer för att för närvarande vara ett lämpligt verktyg.
- Väldigt fyrkantigt verktyg, har svårt att beskriva komplicerade krav såsom tidsberoende krav och liknande. I vissa fall kan dessa konstrueras i Stateflowdiagram, men även då kan tidsberoendet skapa problem.
- Kravblocken kan bara läggas på insignalerna och inte mitt i modellen, vilket avsevärt försvårar användandet.
- Inte möjligt att verifiera motsägande krav samtidigt.
- Inte möjligt att modifiera modellen under körningarna, vilket leder till onödig väntetid.
- Påfallande dålig testtäckning vid verifieringen. Dock är bevisningsdelen av verktyget inget testtäckningsverktyg.
- Testningsdelen av SDV är bristfällig. Verktyget kan inte användas på modeller av den här storleken, inte heller på små delar av den. För automatisk testning bör annat verktyg användas.
- Dyrt, den billigaste licensen kostar ca 70000 kr för ett år.

Reactis Validator

Fördelar

- Stor flexibilitet vid konstruerandet av krav och mål.
- Möjlighet att köra många krav samtidigt. Dessutom möjlighet att validera motsägande krav samtidigt.
- Möjlighet att modifiera modellfiler samtidigt som körning pågår.
- Möjlighet att ha flera modellfiler öppna samtidigt.
- Kompatibelt med väsentliga Simulinkblock, t ex s-funktioner och bussar. Kräver blott små modifikationer av grundmodellen.
- Möjligheten att lägga in testpunkter så att intressanta signaler kan undersökas vid testkörningen.
- Producerar täckningsrapport där samtliga delar av modellen kan undersökas för täckningsgrad. Rapporten kan dessutom exporteras till html och sparas.
- Samtliga testfall som genereras kan sparas och laddas upp för simulering i Validator.

Nackdelar

- Dålig testtäckning vid körningar på hela modellen, uppdelning krävdes för att täckningen skulle förbättras nämnvärt. Dåligt för ett verktyg som av det egna företaget anses kunna klara påtagligt stora modeller (Reactis Validator User's Guide, 2008). Grundmodellen som användes vid implementeringen var dessutom förhållandevis liten och borde inte ha orsakat några storleksproblem för Validator.
- Väldigt svårt att få en tillfredsställande grad av testäckning även efter modifieringar.
- Uppdelning av modellen är tidskrävande då vissa delar av modellen måste modifieras för att bli kompatibla med Validator. T ex klarar Validator inte av utsignaler som består av bussar utan samtliga utsignaler måste separeras.
- Långsam validering. För att uppnå tillfredställande resultat krävdes mellan 5-12 minuter.
- Konstruerandet av krav och mål är väldigt komplicerat och kräver separat övning. Svårt att uttrycka krav på signaler som ska vara sanna ibland. Detta borde inte vara några problem då diagramkrav kan bildas innehållande if-block, men stora problem uppstod när detta gjordes.
- Resultaten är ganska godtyckliga beroende på hur hög graden av testtäckning är.
- Inga bra motexempel vid falsifiering av krav föreslås vilket leder till svårigheter att förstå varför ett krav inte uppfylls, och därigenom också svårt att finna fel gjorda av användaren.
- Verktyget låser sig ofta och kräver sedan lång tid för nedstängning och omstart. Låsningarna uppträder ofta när Matlab är igång parallellt.
- Dålig användarmanual.
- Klarar ej av att hantera bussar som utsignaler, vilket kräver extra modifieringar vid uppdelningar av en modell.

9.2.1 Kravspecifikationen

För att valideringen och verifieringen med SDV och Reactis och sannolikt även med andra verktyg, ska fungera och slutföras så snabbt som möjligt behövs en bra och konkret kravspecifikation. Detta är särskilt viktigt om verifieringen ska utföras av någon som inte medverkat i framtagandet av modellen. Vid denna implementering var detta fallet, och det

fanns flera svårigheter med kravdokumentet. Dels förekom ett antal krav som inte var uppdaterade, t ex signalbegränsningar och dylikt som inte överensstämde med modellen. Detta bidrog bl a till att alla konstanter dubbelkollades, vilket tog extra tid. Dels förekom flera krav på signaler och block som inte längre fanns kvar i modellen efter omfattande revideringar. Även i detta fall försvåras användningen om användaren själv inte deltagit i varit med vid designandet av modellen. En rekommendation är att alltid revidera kravspecifikationen vid då revideringar av modellen utförs.

9.3 Formell Verifikation, någonting för Scania?

Efter detta examensarbetes slutförande kan sägas att formell verifiering definitivt är en intressant ansats för industrin i sin helhet och därför också för Scania. Dock är SDV, som undersökts i detta arbete, inte tillräkligt utvecklat för att kunna anses som användbart i nuläget. Eftersom detta examensarbete endast har studerat modellbaserad utveckling kan inga entydiga slutsatser dras om tillförlitligheten av formell verifiering för system uppbyggda av sedvanlig C-kod. Dessutom har endast ett verktyg för formell verifiering undersökts, vilket leder till att inga slutsatser kan dras var gäller formell verifikation som metod.

Rekommendationen för Scania är sålunda att undersöka fler verktyg för formell verifiering innan det integreras i utvecklingsprocessen. Troligtvis finns det ett flertal verktyg för detta ändamål som är avsevärt bättre och mer välutvecklat är SDV. Även om den modellbaserade utvecklingen på Scania huvudsakligen sker i Simulink är det redan idag osäkert om verktyg integrerade i Simulink verkligen är de bästa eller, än mer tveksamt, kommer att vara det i framtiden.

Som tidigare betonats bör formell verifiering kombineras med testning snarare än ersätta den. Hur detta ska ske kan bedömas först när fler och bättre verktyg har uppmärksammats. För det fall ett enastående och helt tillförlitligt verktyg hittats kan formell verifiering komma att bli den säkra metoden för att hitta fel. På så sätt skulle man kunna vara helt säker på, eller näst intill säker på, att en produkt är felfri vid lanseringen.

Om formell verifiering någon gång i framtiden integreras i Scanias utvecklingsprocess är det nödvändigt att potentiella användare först erbjuds adekvat utbildning. Poängen med många av de nyare verktyg för formell verifikation som finns på marknaden är att användaren ska slippa konfronteras med den bakomliggande matematiken och logiken. Även om verktygen kommer att bli alltmer användarvänliga, är det likväl nödvändigt för användaren att förstå vad som händer vid verifieringen. Om inte så är fallet kommer den inte att utföras korrekt. Därmed kommer verifikationen heller inte bli tillfredsställande. Vidare måste hela utvecklingsprocessen designas med medvetenhet att formell verifikation ska användas. Verifieringen måste gå hand i hand med modelldesignandet annars kommer svårigheter med bla kompatibilitet troligen att uppstå.

9.3.1 Fortsatt arbete

Inom området formell verifiering finns betydligt mer att studera och undersöka än vad som av tidsskäl har varit möjligt i detta examensarbete. Ytterligare ett antal veckor skulle behövas för att undersöka de utvärderade verktygen i den utsträckning som behövs för att få en rättvisare bild av hur de fungerar och vad de förmår hantera. En mer erfaren användare skulle utan tvivel haft bättre förutsättningar för att lösa de problem som uppmärksammats i detta examensarbete.

Testning och analys av fler likartade verktyg, t ex Embedded Validator är en förutsättning för bättre evaluering av potentialen hos formell verifiering. Den omständigheten att SDV inte är ett tillräckligt utvecklat verktyg betyder inte att formell verifiering är en metod som utan vidare ska förkastas. Ytterligare ett examensarbete som undersöker flera andra verktyg rekommenderas. Genomförandet bör dock avvakta tills fler verktyg har lanserats på marknaden.

10 Referenser

Amla N., Du, X., Keuhlmann, A., Kurshan, R.P. & McMillan, K.L., (2005). *An analysis of SAT-based model-checking techniques in an industrial environment*. Cadance Design Systems. www.cadence.com/company/cadence_labs/kuehl_CHARME_2005_Analysis.pdf 2008-04-14

Andersson, J., (2005). *Automatic test vector generation and coverage analysis in model-based software development*. Masters Thesis, University of Linköping.

Bérard, B., Bidoit, M., Finkel, A., Laroussinie, F., Petit, A., Petrucci, L., Schnoebelen, Ph. & McKenzie, P., (2001). *Systems and Software Verification; Model Checking Techniques and Tools*. Springer-Verlag Berlin Heidelberg. ISBN: 3-540-41523-8

Black, P.E., (1998). *Dictionary of Algorithms and Data Structures*. National Institute of Standards and Technology. U.S. Department of Commerce. http://www.nist.gov/dads/ 2008-07-29

Bouali, A. & Dion, B., (2005). *Formal-Verification for Model-Based Development*. Occupant Safety, Safety-critical Systems and Crashworthiness. SAE International.

Clark, E. & Kroening, D., (2006). *ANSI- C Bounded model checking user manual*. Technical report, School of Computer Science, Carnegie Mellon University Pittsburgh Pennsylvania. http://www.cprover.org/cbmc/doc/cbmc-techreport.pdf 2008-07-29

Eriksson, L-H., (2008). Via e-post, 2008-05-25

Holzmann, G.J., (1997). *The Model Checker SPIN*. IEEE Transactions on Software Engineering, vol. 23 No. 5, May 1997. http://citeseer.ist.psu.edu/holzmann97model.html 2008-07-29

Holzmann, G.J., (2000). *Logic Verification of ANSI-C code with SPIN*. Bell laboratories, Lucent Technologies. Proc. SPIN2000, Springer Verlag, LNCS 1885, pp. 131-147. http://spinroot.com/spin/Doc/spin2000.pdf 2008-07-29

Huth, M. & Ryan, M., (2004), *Logic in Computer Science*. Cambridge University Press, New York, USA. ISBN: 0-521-54310X

Mattson, L. & Niska, R., (2003). Mjukvarutestning, *Gör man som man ska eller gör man som man vill?* Examensarbete, Luleå Tekniska Universitet.

Nilsson, U., (2002). *Industriell användning av formella metoder*. Institutionen för datavetenskap, Linköpings Universitet. www.ida.liu.se/~ulfni/ceniit 2008-04-10

Ouimet, M., (2005). *Formal Software Verification: Model Checking and Theorem Proving*. Embedded Systems Laboratory Technical Report. Massachusetts Institute of Technology. http://esl.mit.edu/publications/ESL-TIK-00214.pdf 2008-07-29

Palshikar, G.K., (2004). *An introduction to model checking.* Embedded Systems Design.

2004-02-12. www.embedded.com/columns/technicalinsights 2008-04-14

Ranville, S., (2004). *Case Study of Commercially Available Tools that Apply Formal Methods to a Matlab/Simulink/Stateflow Model.* In-Vehicle Networks and Software Electrical Wiring Harnesses and Electronics and Systems Reliability. SAE International.

Rao, A.C., McMurran R. & Jones, R.P., (2008). *A Critical Analysis of Model-Based Formal verification efforts within the Automotive Industry*. In-Vehicle Networks and Software, 2008. SAE International.

Schlich, B. & Kowalewski, S., (2006). *Model Checking C Source Code for Embedded Systems.* Embedded Software Laboratory, RWTH Aachen University. www-i11.informatik.rwth-aachen.de 2008-07-30.

Zheng, Y., Zhou, J. & Krause, P., (2007). *An Automatic Test Case generation Framework for Web Services.* Department of Computing, University of Surrey, Guildford UK. Journal of software vol. 2 No 3, pp. 64-77. Academy Publisher. www.academypublisher.com/jsw/vol02/no03/jsw02036477.pdf 2008-07-29

Embedded Validator, OSC Embedded Systems AG. dSPACE Inc. 2004. www.dspaceinc.com 2008-07-29

The Mathworks, (2007). *Simulink Design Verifier Users Guide v1.1*. www.mathworks.com 2008-07-29.

Bell Laboratories, Lucent Technologies, (2008). *Spin Model Checker*. www.spinroot.com 2008-07-29.

Reactive Systems Inc., (2008). *Reactis v2008 Users Guide*. www.reactive-systems.com/reactis/doc/user/user009 2008-08-01

University of California at Berkley, (2005). *BLAST User's Manual*. http://embedded.eecs.berkeley.edu/blast/doc/blast.pdf 2008-07-29

Appendix A

A1. Block och funktioner ej stödda av SDV

Nedan följer en lista över samtliga funktioner och block från Simulinks bibliotek som inte stöds av SDV. Listan syftar till Simulink Design Verifier v 1.1 för MATLAB 2007b. För senare versioner av verktyget se www.mathworks.com

Funktioner Simulink Design Verifier inte stödjer

Variable-step solvers
Complex numbers

Block Simulink Design Verifier inte stödjer

Additional Discrete
Fixed-Point State-Space
Transfer Fcn Direct Form II
Transfer Fcn Direct Form II Time Varying
Increment/Decrement
Decrement Time To Zero
Continuous
Derivative
Integrator
State-Space
Transfer Fcn
Transport Delay
Variable Time Delay
Variable Transport Delay
Zero-Pole
Discontinuities
Dead Zone
Relay
Rate Limiter
Discrete
Discrete Filter
Discrete Derivative
Discrete State-Space
Discrete Transfer Fcn
Discrete Zero-Pole
Integer Delay
Tapped Delay
Weighted Moving Average
Logic and Bit Operations
Combinatorial Logic
Lookup Tables
Cosine
Direct Lookup Table (n-D)
Interpolation Using Pre-lookup
Lookup Table Dynamic
Sine
Pre-lookup

Math Operators
Magnitude-Angle to Complex
Real-Imag to Complex
Sine Wave Function
Trigonometric Function
Unary Minus
Weighted Sample Time Math
Ports & Subsystems
Model
Signal Attributes
Weighted Sample Time
Width
Signal Routing
Bus Assignment
Bus Creator
Bus Selector
Sinks
Stop Simulation
Sources
Band-Limited White Noise
Chirp Signal
From File
From Workspace
Random Number
Repeating Sequence
Repeating Sequence Interpolated
Signal Builder
Signal Generator
Sine Wave
Uniform Random Number
User-Defined
Embedded MATLAB Function
Fcn
Level-2 M-file S-Function
MATLAB Fcn
S-Function
S-Function Builder

A2. Block och funktioner ej stödda av Validator

Nedan följer en lista på de block och funktioner som finns att tillgå i Simulink men som inte stöds av Reactis Validator. Listan gäller för Reactis v2008 och MATLAB 2007b, för senare upplagor av verktyget se www.reactive-systems.com

Funktioner Reactis Validator inte stödjer

Expression Evaluation

ASSIGNIN

EVALIN

Model Construction/Modification

EVALIN	DELETE_PARAM
ADD_LINE	NEW_SYSTEM
ADD_PARAM	REPLACE_BLOCK
DELETE_BLOCK	SET_PARAM
DELETE_LINE	

Block Reactis Validator inte stödjer

Continuous

Derivative
Integrator
State-Space
Transfer Fcn
Transport Delay
Variable Time Delay
Variable Transport Delay
Zero-Pole

Discontinuous

Rate Limiter

Discrete

First-Order Hold
Weighted Moving Average

Lookup Tables

Cosine (output formula other than cos)
Direct Lookup Table (n-D)
Sine (output formula other than sin)

Math Operations

Algebraic Constraint
Complex To Magnitude-Angle
Complex To Real-Imag
Magnitude-Angle To Complex
Real-Imag To Complex
Math Function (Hermitian)
Permute Dimensions
Sine Wave Function (using external time)
Squeeze
Vector Concatenate

Model Verification

Assertion
Check Discrete Gradient
Check Dynamic Gap
Check Dynamic Lower Bound
Check Dynamic Range
Check Dynamic Upper Bound
Check Input Resolution
Check Static Gap
Check Static Lower Bound
Check Static Range
Check Static Upper Bound

Model-Wide Utilities

Block Support Table
Time-Based Linearization
Trigger-Based Linearization

Ports and Subsystems

Code Reuse Subsystem

Signal Routing

Manual Switch

Sinks

Stop Simulation
To File
To Workspace
XY Graph

Sources

Band-Limited White Noise
Chirp Signal
Clock
Pulse Generator (time based)
Random Number
Uniform Random number

User-Defined Functions

Embedded MATLAB function
Level-2 M-File S-function
MATLAB Fcn

Additional Discrete

Transfer Fcn Direct Form II
Transfer Fcn Direct Form II Time Varying