Onderzoek TLM

Onderzoek TLM extractie door SHaBE

9 mei 2011

Dennis van Leeuwen 07004222

Inhoudsopgave

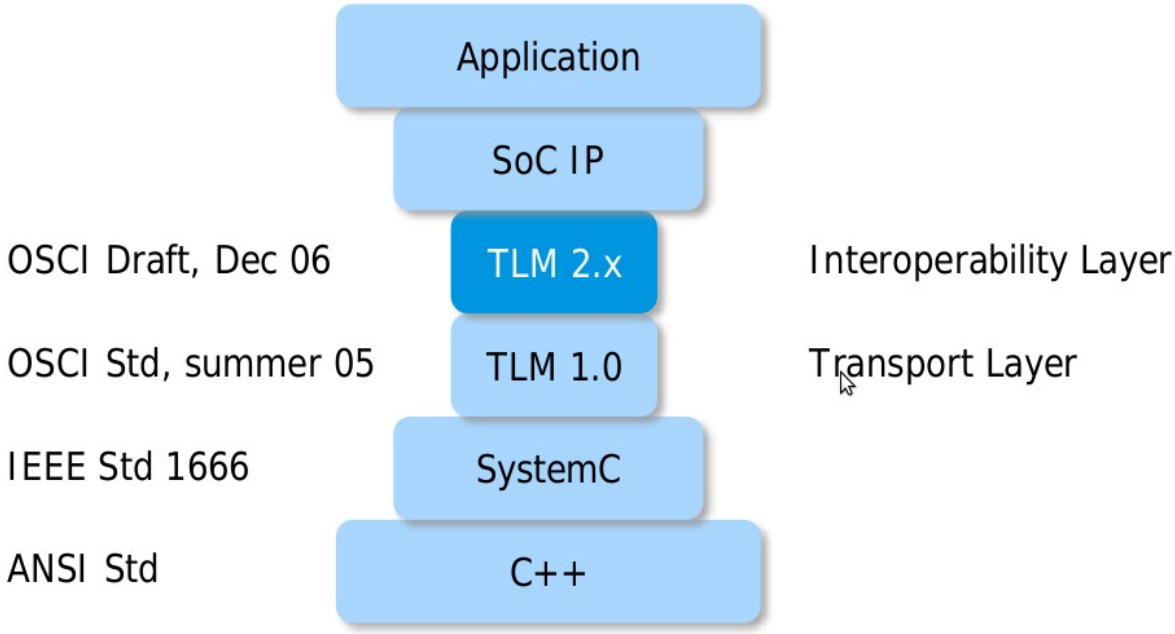
1. Inleiding

In dit document wordt het onderzoek naar de extractie van TLM uit SystemC modellen door SHaBE beschreven. Door dit onderzoek uit te voeren wordt het duidelijk of het mogelijk is om TLM uit een SystemC model te extraheren. Tijdens het onderzoek wordt er alleen gekeken of de onderdelen m.b.t. de hiërarchie van een SystemC/TLM model geëxtraheerd kan worden.
De hoofdvraag van dit onderzoek is daarom ook: “Is het mogelijk om TLM onderdelen die worden gebruikt in de hiërarchie van een SystemC model uit een SystemC model te extraheren?”. Om tot deze conclusie te komen wordt er in de eerste hoofdstukken onderzochte welke onderdelen er tot TLM behoren en of er al reeds programma's zijn die TLM uit een SystemC model kunnen extraheren en hoe deze programma's dat doen.
Vervolgens zal er door gebruik te maken van verschillende TLM SystemC voorbeelden worden onderzocht of het mogelijk is om TLM onderdelen uit deze SystemC/TLM voorbeeld modellen te extraheren. Bij ieder voorbeeld zal worden toegelicht of het mogelijk is om het voorbeeld te extraheren en hoe de gegevens die nodig zijn worden geëxtraheerd.

Het onderzoek wordt gedaan m.b.v. de huidige SystemC 2.2.0 standaard, TLM 2.0.1 standaard en er wordt gebruik gemaakt van GDB 7.2 en GCC 4.5.0.

2. Wat is TLM?

TLM staat voor Transaction Level Modelling. Met TLM kunnen de transacties tussen modules worden beschreven. Deze transacties worden uitgevoerd door gebruik te maken van sockets en een channels. Sockets zijn nieuw t.o.v. de SystemC standaard en zijn in feite uitgebreide porten.
Zoals in onderstaande afbeelding is te zien is TLM een uitbreiding van de SystemC library. De implementatie van TLM maakt gebruik van de SystemC library.



3. Wat is het verschil tussen SystemC en TLM?

Op de vraag “Wat is het verschil tussen SystemC en TLM?” zijn verschillende antwoorden mogelijk. Dit is namelijk afhankelijk van de manier hoe je de vraag stelt. Je kan het zien als, wat bevat SystemC en wat bevat TLM en wat is daardoor het verschil tussen deze twee. Echter is dat niet de manier waarop deze vraag gesteld werd. Het doel van deze vraag is om te onderzoeken wat nu het verschil maakt waarom er TLM is opgezet naast SystemC en wat maakt TLM nu anders als SystemC?

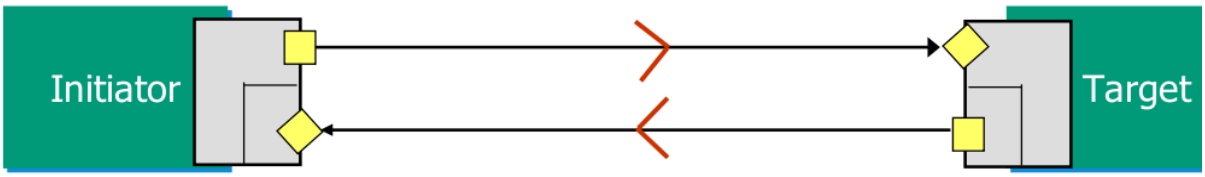
TLM is bedoeld om er een systeem op een abstracter niveau mee te beschrijven als met SystemC. TLM is bedoeld om de communicatie tussen modules te beschrijven. SystemC is bedoeld om de implementatie hiervan te beschrijven. Wanneer een abstracter beschreven systeem wordt gesimuleerd zijn er minder stappen nodig tijdens de simulatie, omdat het systeem minder gedetailleerd is beschreven. Systemen worden steeds complexer en om een gedetailleerd en complex systeem te testen is er veel tijd nodig, zowel voor het maken van de tests als voor het uitvoeren van de tests. Om deze tijd te reduceren wordt er gebruik gemaakt van een abstracter niveau om een systeem in te beschrijven.

4. Waar bestaat TLM uit?

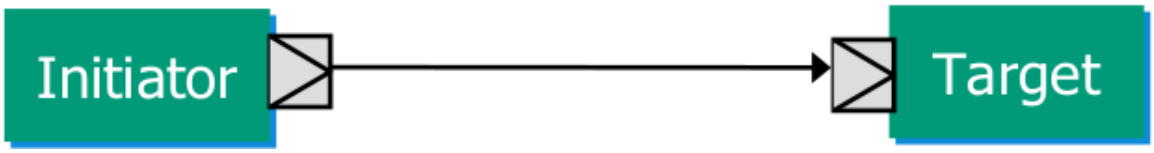
TLM is zoals gezegd een uitbreiding van de SystemC library en is vooral bedoeld om de overdracht van data tussen modules uit te breiden. Dit wordt gedaan door de onderstaande onderdelen in TLM te implementeren.

4.1. Sockets

Sockets zijn begin of eindpunten van een transaction. Het grootste verschil tussen SystemC en TLM is dat in TLM het systeem op een hoger abstractie niveau wordt beschreven. Zowel TLM als SystemC kunnen door elkaar worden gebruikt. Door alleen TLM te gebruiken wordt het gemodelleerde systeem alleen op een standaard manier op een abstracter niveau beschreven en kan daardoor sneller worden gesimuleerd tijdens de SystemC simulatie. In TLM bestaan er 2 type transport, blocking en non-blocking.
Om te kunnen bepalen of een socket een initiator(socket mag beginnen met zenden) of een target(mag alleen iets versturen als reactie op ontvangen data) is, zijn er 2 type sockets gedefinieerd in TLM: een initiator en een target socket. De initiator neemt het initiatief voor het zenden van een bericht. De target kan op dit bericht vervolgens een bericht terugsturen.



Omdat de target nooit het initiatief kan nemen om iets op eigen wil naar de initiator te sturen, maar omdat er wel tweerichtingsverkeer mogelijk is, wordt dit in het vervolg zoals in de onderstaande afbeelding weergegeven.



Tussen de initiator en target sockets kan een op een, een op meer of meer op een relatie bestaan. Om deze relaties tussen de sockets toe te staan, zijn er diverse type sockets in TLM gedefinieerd.

4.2. Generic Payload

Via sockets kan een bericht(In TLM termen een transaction) worden verstuurd. In TLM is er een standaard bericht gedefinieerd ook wel een Generic Payload genoemd. In de generic payload zijn diverse attributen aanwezig waarmee gespecificeerd wordt welke actie het bericht heeft((READ of WRITE) en welke data er in het bericht aanwezig is.

4.3. DMI

DMI staat voor Direct Memory Interface. Door het gebruik van DMI kan er eenmalig een aanvraag worden gedaan om toegang tot een bepaald stuk data in een module, dit voor lezen of schrijven.

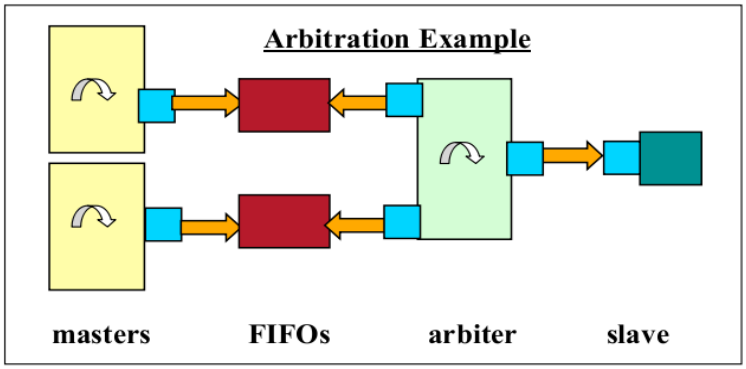
Wanneer meerdere modules toegang tot eenzelfde stuk geheugen hebben en dit stuk wordt gewijzigd, dan verliezen deze modules de toegang tot het stuk geheugen en moet er opnieuw een aanvraag tot het stuk geheugen worden gedaan.

4.4. Blocking Transport Interface en Non-blocking transport Interface

Tijdens het maken van een SystemC model kan er gebruik worden gemaakt van blocking en non-blocking transport. Dit zijn twee manier voor het modelleren van de communicatie tussen sockets. Omdat dit onderdeel een belangrijk onderdeel is van TLM, maar te groot is om goed uit te leggen en het niks met de hiërarchie van een TLM model te maken heeft, wordt dit niet verder besproken.

Naast bovenstaande onderdelen bevat TLM diverse componenten die nodig zijn om de transactions succesvol te laten verlopen.

4.5. tlm_fifo<T>



Wanneer meerdere masters via een arbiter transactions naar een slave kunnen sturen. Is het van belang dat er een FIFO tussen de master en de arbiter komt te staan, zodat als er meerdere transactions tegelijk door de masters worden gestuurd, dan kunnen deze eerst in de FIFO worden gebufferd, alvorens deze door de arbiter in behandeling worden genomen om aan de slave door te geven.

4.6. Extra type channels

M.b.v. de tlm_req_rsp_channel<REQ, RSP> en tlm_transport_channel<REQ, RSP> kunnen de transactions die nog moeten worden verstuurd bij blocking transport kunnen worden gebufferd. In deze channels zit een tlm_fifo module ingebouwd, die ervoor zorgt dat de berichten gebufferd worden.

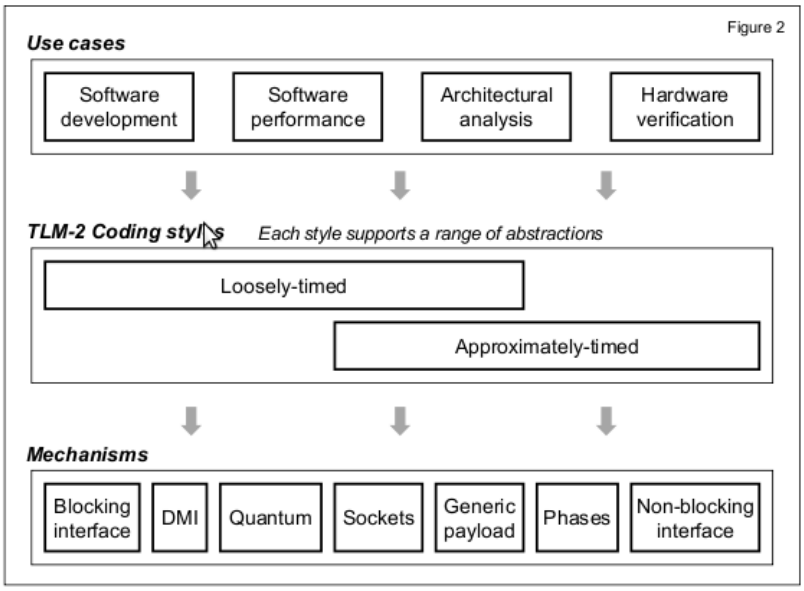
4.7. Extra type port

Om een module te kunnen debuggen is er de zogenaamde analyse port aan TLM toegevoegd. Via een analyse port is het mogelijk om data een module uit te sturen, zonder dat er via de port van de module die de data ontvangt iets terug kan worden gestuurd.

TLM 2.0 introduceert 2 nieuwe aspecten t.o.v. SystemC

- Loosely Timed (LT) modeling – “fast”
- Approximately Timed (AT) modeling – “accurate”

Dit zijn 2 mechanismes waarmee de timing van een model kan worden aangegeven. De keuze voor een van de timing mechanismes ligt geheel aan het doel waarvoor SystemC en TLM gebruikt wordt. Als timing belangrijk is en de snelheid van het uitvoeren van het model minder belangrijk is, is het verstandig om AT modelling te gebruiken. Wanneer timing minder belangrijk is dan de snelheid waarmee een model uitgevoerd kan worden, is het verstandig voor LT modelling te kiezen.



Uit alle bovenstaande beschrijvingen kan geconcludeerd worden dat de volgende onderdelen tot de hiërarchie van een TLM model behoren. Deze onderdelen zullen dus aan de extractie van SHaBE moeten worden toegevoegd

- Initiator sockets, tlm_initiator_socket
- Target sockets, tlm_target_socket
- TLM FIFO's, tlm_fifo
- Analyse ports, tlm_analysis_port
- tlm_req_rsp_channel
- tlm_transport_channel
- multi passthrough target sockets
- multi passthrough initiator sockets

Diverse onderdelen zoals:

1. Quantum
2. Generic Payload
3. DMI
4. Phases

zijn ook onderdelen van TLM. Deze onderdelen kunnen echter niet als een deel van de hiërarchie van een TLM model worden uitgedrukt. Daarom worden deze onderdelen tijdens het onderzoek buiten beschouwing gelaten.

5. Zijn er al reeds programma's die TLM kunnen extraheren uit SystemC modellen?

Om te kunnen onderzoeken of het mogelijk is om TLM uit SystemC modellen te extraheren wordt er onderzocht of er al programma's ontwikkeld zijn die al TLM uit een SystemC model kunnen extraheren. Tijdens het zoeken is er maar een programma gevonden dit TLM kan extraheren. Dit programma heet Lussy, hieronder een uitleg over Lussy

5.1. Lussy

Lussy is ontwikkeld zoals in de thesis over Lussy wordt gezegd als eerste toolkit die TLM uit SystemC code kan extraheren. Tijdens het onderzoek blijkt dat Lussy totnutoe ook het enige programma is wat TLM kan extraheren. Lussy gebruikt de geëxtraheerde gegevens om een grafische weergave te geven van het SystemC model. Lussy verkrijgt de gegevens over een TLM model echter door gebruik te maken van PINAPA. PINAPA is zoals in de thesis over SHaBE uitgelegd een tool die gebruikt maakt van een oudere versie van SystemC en gebruikt een verouderde en zelf gepatchde versie van GCC. Hierdoor wordt PINAPA als verouderd beschouwd. Omdat Lussy afhankelijk is van PINAPA, is Lussy dus niet geheel geschikt voor de extractie van TLM.

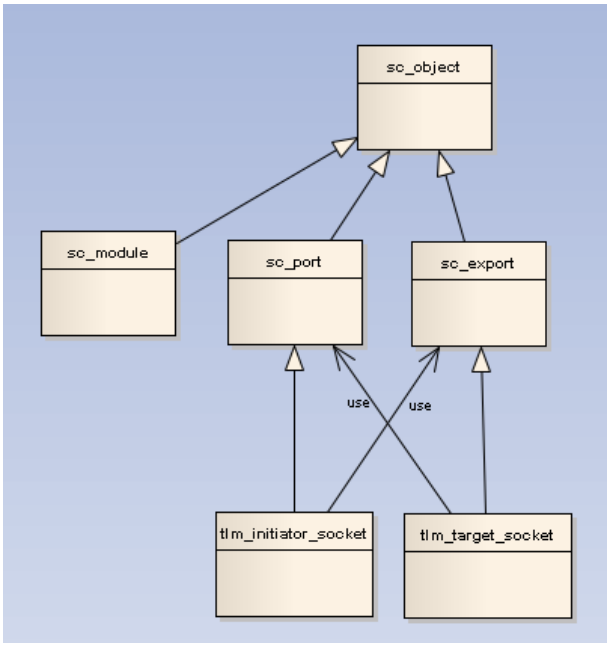
5.2. Conclusie

Zoals gezegd wordt Lussy niet geschikt geacht voor het extraheren van TLM uit SystemC code, waardoor het noodzakelijk is om zelf een volledig programma te schrijven welke de extractie van TLM uit SystemC kan uitvoeren.

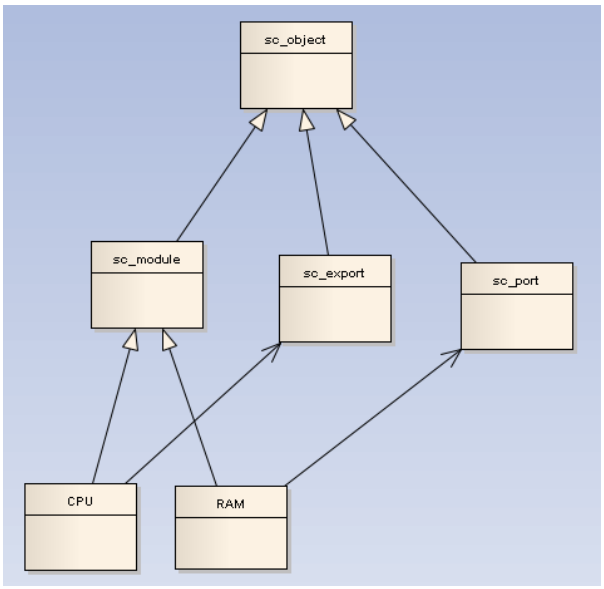
6. Tussentijdse conclusie

Nudat de onderdelen van TLM, de werking van deze TLM onderdelen is bestudeerd en is onderzocht of er al reeds bestaande programma's zijn die TLM kunnen extraheren, kan er worden geconcludeerd, dat het logischer wijze mogelijk zou zijn om SHaBE uit te breiden om de hiërarchie van de TLM modules uit een SystemC model te extraheren. Deze conclusie wordt getrokken omdat de onderdelen uit TLM uitbreidingen zijn van de SystemC library. De TLM classes zijn voor het grootste gedeelte afgeleide klasses van de klasses uit de SystemC library.
Wanneer er een SystemC beschrijving wordt gemaakt in SystemC, worden de klasses van de SystemC library overerft om gebruik te kunnen maken van de functionaliteiten van SystemC. Deze uitleg wordt grafisch weergegeven in de onderstaande afbeeldingen.

Klasses gebruikt door TLM:



SystemC programma:



Omdat verondersteld wordt dat wanneer de objecten van de klasses CPU en RAM uit een SystemC model kunnen worden geëxtraheerd, dat ook de TLM initiator en target socket uit een SystemC model kunnen worden geëxtraheerd, omdat beide klasses afgeleide zijn van een klasse uit de SystemC standaard.

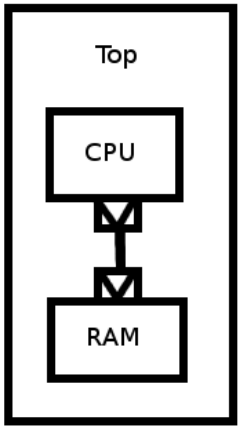
Omdat deze gedachte niet zomaar als conclusie kan worden aangenomen, moet er een praktisch voorbeeld worden gemaakt waarmee kan worden bewezen dat de extractie kan plaatsvinden of juist waarmee kan bewezen dat de extractie niet (volledig) kan plaatsvinden. In het volgende hoofdstuk wordt besproken hoe het programma SHaBE wordt uitgebreid om te onderzoeken hoe de extractie kan plaatsvinden en welke elementen er in SHaBE moeten worden gewijzigd.

7. Voorbeelden

Om te kunnen onderzoeken hoe de hiërarchie van de TLM onderdelen kan worden geëxtraheerd, zullen er eerst SystemC modellen moeten worden gemaakt, die kunnen worden gebruikt om TLM onderdelen uit te extraheren. Om uiteindelijk te onderzoeken of SHaBE kan worden gebruikt om TLM onderdelen te extraheren uit een SystemC model, worden deze voorbeelden gebruikt.

7.1. Voorbeeld 1 Sockets

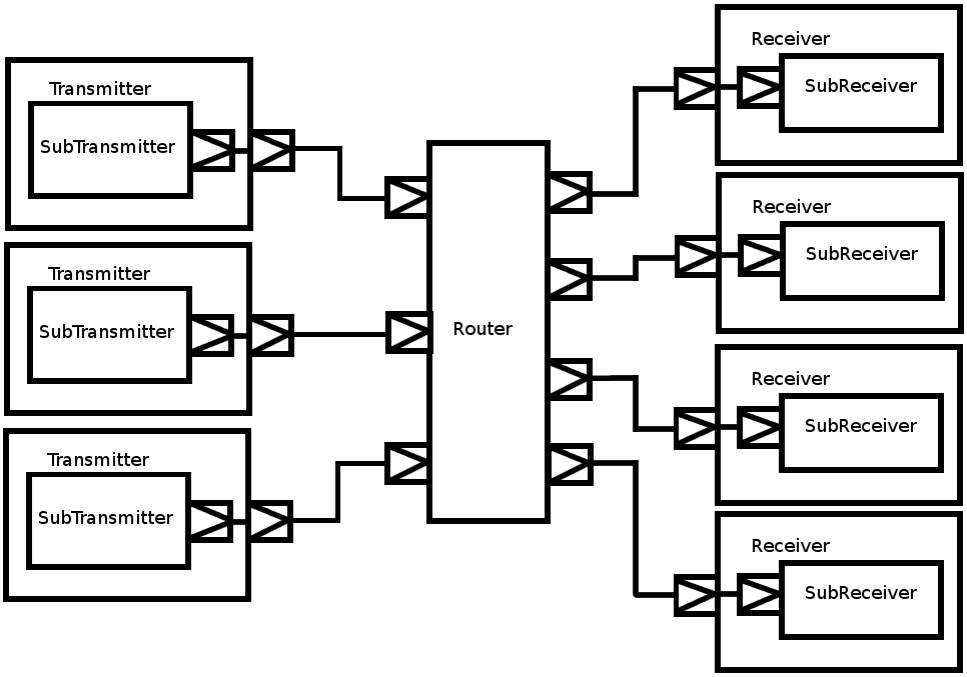
Hierin zijn de CPU en de RAM module gekoppeld aan elkaar via een simple_initiator_socket en een simple_target_socket. Dit zijn beide afgeleide van de initiator en de target socket uit TLM.



Afbeelding 1: Voorbeeld 1

7.2. Voorbeeld 2 Hiërarchisch verbonden sockets.

In voorbeeld 2 worden de sockets hiërarchisch aan elkaar verbonden.

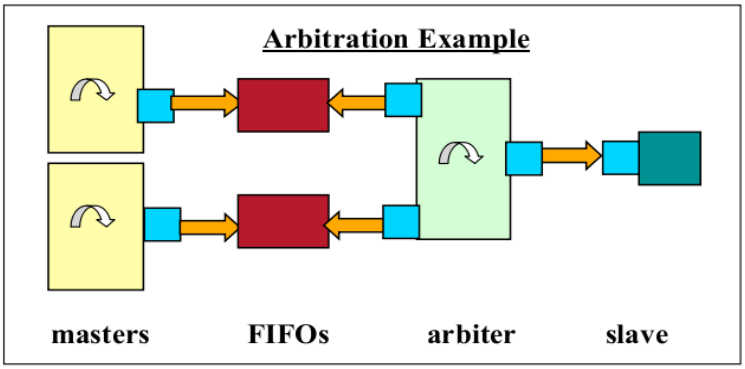


Afbeelding 2: Voorbeeld 2

7.3. Voorbeeld 3 (Met een tlm_fifo)

7.4. Voorbeeld 4 (Voorbeeld 3 met een tlm_req_resp_channel)

7.5. Voorbeeld 5 (Voorbeeld 3 met een tlm_transport_channel)

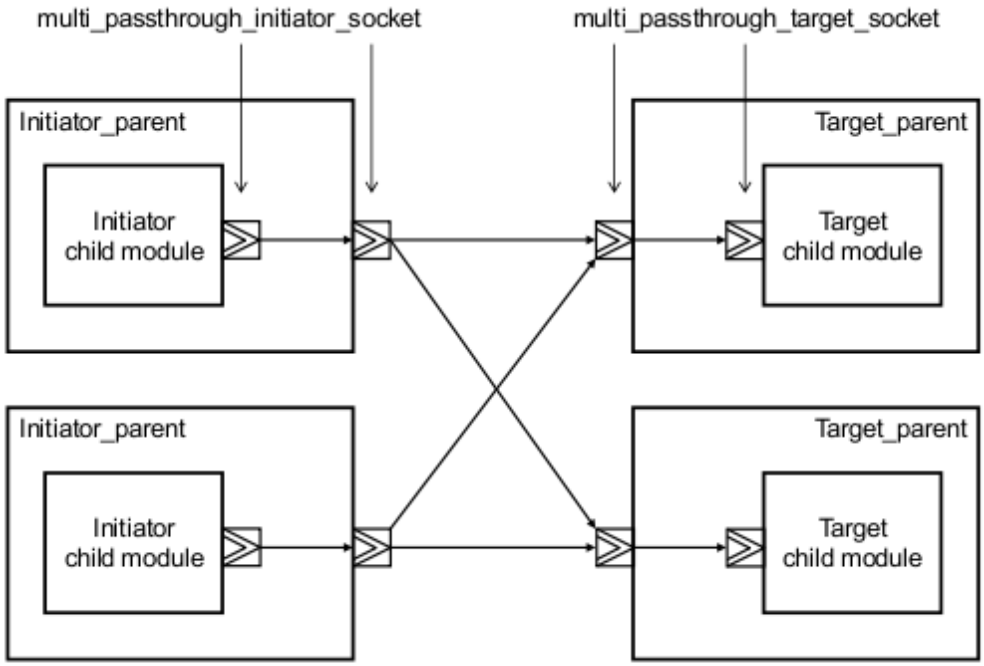


Afbeelding 3: Voorbeeld 3,4 en 5

In de voorbeeld 3,4 en 5 wordt er gebruik gemaakt van het FIFO principe.
In voorbeeld 3 wordt een tlm_fifo geïmplementeerd. Dit is een channel.

In de voorbeeld 4 en 5 worden er twee channels gebruikt die gebruik maken van de tlm_fifo, dit zijn de tlm_req_rsp_channel en de tlm_transport_channel.

7.6. Voorbeeld 6 Multiple passthrough sockets

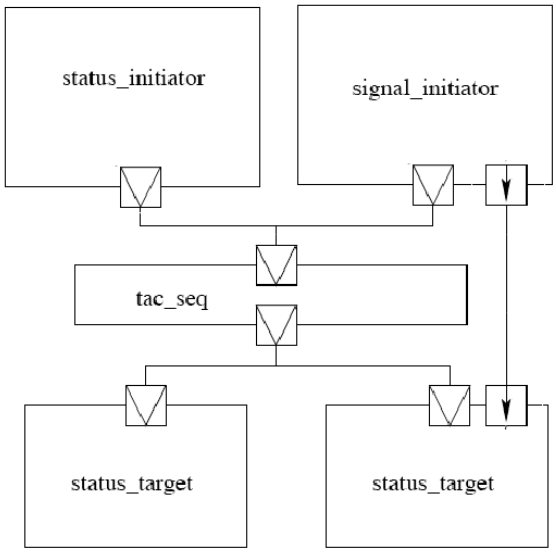


Afbeelding 4: Voorbeeld 6

In voorbeeld 6 worden multi pass through sockets gebruikt. Deze sockets zijn te vergelijken met multi ports in SystemC.

7.7. Voorbeeld 7 Analyse porten

In voorbeeld 7 wordt een analyse port gebruikt. Via een analyse port kan een module alleen gegevens wegschrijven. Dit is bijvoorbeeld handig voor het bijhouden van een log.



Afbeelding 5: Voorbeeld 7

7.8. Voorbeeld 8 Sc_port's en Sc_export's

Voorbeeld 8 is gelijk aan voorbeeld 1, echter wordt er in voorbeeld 8 een sc_port ipv een tlm_initiator_socket en een sc_export ipv een tlm_target_socket gebruikt.
Voorbeeld 8 wordt tijdens de analyse fase van iteratie 1 toegevoegd, omdat blijkt dat er een nieuw programma wordt gemaakt en sc_port en sc_export worden gebruikt bij het koppelen van tlm_fifo en dus ook bij tlm_fifo channels.

8. Extractie van de voorbeelden

In dit hoofdstuk wordt er per voorbeeld besproken in hoeverre SHaBE gebruikt kan worden voor de extractie van TLM uit een SystemC model. Voordat de extractie aangepast wordt, wordt ieder voorbeeld geëxtraheerd door SHaBE om te analyseren in hoeverre SHaBE het voorbeeld kan extraheren. Vervolgens wordt er uitgelegd hoe het ontwerp van de verandering eruit komt te zien in de opbouw van de extractie. Tijdens de implementatie zijn er vaak wijzigingen t.o.v. het ontwerp gemaakt. Samen met de eventuele problemen, worden deze besproken tijdens de implementatie. In de testfase worden er tests uitgevoerd die tijdens de ontwerpfase zijn gemaakt. In het hoofdstuk test wordt er uitgelegd welke onderdelen er getest zijn en wat de testresultaten en aanpassingen zijn geweest.

9. Voorbeeld 1 – tlm sockets

9.1. Analyse

In voorbeeld 1 worden de sockets uit de TLM library geïmplementeerd. Om te onderzoeken tot in hoeverre dit voorbeeld kan worden gebruikt tijdens de extractie van TLM en SystemC wordt het model in SHaBE geladen en wordt er gekeken hoe de hiërarchie van het geëxtraheerde model eruitziet. Dit wordt gedaan voordat SHaBE aangepast wordt. De XML die SHaBE genereert is te zien in bijlage A.
Hierin is te zien dat SHaBE zowel een `tlm_utils::simple_initiator_socket` object als een `tlm_utils::simple_target_socket` object naar een sc_export en een sc_port extraheert. Zoals in de afbeelding in het hoofdstuk van de tussentijdse conclusie is te zien, klopt dit resultaat met de structuur van TLM en SystemC. Een `tlm_utils::simple_initiator_socket` is namelijk een afgeleide van een sc_port en heeft een verwijzing naar een sc_export. Een `tlm_utils::simple_target_socket` is een afgeleide van sc_export en heeft een verwijzing naar een sc_port.
Tijdens de analyse van het programma SHaBE wordt duidelijk dat SHaBE op een overzichtelijkere en uitbreidbare manier kan worden opgebouwd, waardoor nieuw te extraheren objecten in de toekomst beter kunnen worden toegevoegd aan SHaBE. Omdat dit onderzoek gericht is op het onderzoeken van de mogelijkheid om TLM onderdelen te extraheren is er gekozen om een nieuw programma naast SHaBE te ontwikkelen. M.b.v. dit programma wordt de extractie van TLM onderdelen verzorgt. Omdat de extractie m.b.v. GDB moet worden gedaan, wordt ervoor gekozen om de klasses uit SHaBE die de communicatie met GDB verzorgd te herbruiken. Deze klasses zijn gestructureerd opgebouwd waardoor het herbruiken van deze componenten wordt vergemakkelijkt.

9.2. Ontwerp

De applicatie die wordt ontwikkeld bestaat uit drie onderdelen:

1. Het extraheren van de gegevens
2. Het opbouwen van de hiërarchie
3. Het opslaan van de hiërarchie

9.3. Extraheren van de gegevens

In de analyse is onderzocht en ondervonden dat de `tlm_utils::simple_initiator_socket` en `tlm_utils::simple_target_socket` objecten worden omgezet naar sc_export en sc_port objecten uit de SystemC library. Om te voorkomen dat de objecten uit de TLM library worden omgezet naar SystemC objecten zal de extractie van TLM objecten voor de extractie van SystemC objecten moeten plaatsvinden. Omdat er een programma naast SHaBE wordt ontwikkeld, hoeft hier nu nog geen rekening mee te worden gehouden, maar dit is een gegeven waar toekomstige ontwikkelaars van SHaBE wel rekening mee moeten houden.

9.4. GDB/MI

Om de gegevens uit de executable van het SystemC model te halen wordt er gebruik gemaakt van de GDB debugger. Deze debugger kan worden aangeroepen aan de hand van commando's op te geven via de commandline.

Echter heeft GDB ook een zogenaamde MI(Machine Interface). Via deze MI kan de debugger vanuit een applicatie worden aangeroepen en wordt er volgens een protocol gecommuniceerd.
Tijdens de ontwikkeling van SHaBE heeft de opdrachtgever al voor de GDB/MI interface gekozen, omdat blijkt dat er op deze manier via een standaard manier kan worden gecommuniceerd, wordt er besloten om de gegevens voor de extractie van TLM ook via de GDB/MI interface te laten verlopen. Omdat er via een gedefinieerd protocol wordt gecommuniceerd is het mogelijk om de resultaten die worden ontvangen te parsen en de geparste gegevens te verwerken.
Deze parser bestaat al reeds en kan goed worden gebruikt tijdens de TLM extractie omdat de parser gestructureerd is opgebouwd.
Hieronder een voorbeeld van een commando en een resultaat welke via GDB/MI worden verstuurd.

```
Commando:
(gdb)
111-exec-return

Resultaat:
111^done,frame={level="0 ",func="callee3",
args=[{name="strarg",
value="0x11940 \"A string argument.\""}],
file="../../../devo/gdb/testsuite/gdb.mi/basics.c",line="18"}
```

Om het formaat wat ontvangen wordt te parsen wordt er gebruik gemaakt van de parser die in SHaBE wordt gebruikt. Deze parser parset alle commando's en resultaten die ontvangen worden in een boomstructuur. Door de boom uit te lezen kunnen de benodigde resultaten worden verkregen.
SHaBE extraheert op dit moment verschillende ScObjectInfo objecten m.b.v. de GDB debugger uit een SystemC model. Deze ScObjectInfo objecten bevatten onder andere de naam, type en de hiërarchische naam van het geëxtraheerde SystemC object. De naam van de klasse waarvan het object is afgeleid wordt ook opgeslagen.
Aan de hand van deze informatie kan worden bepaald welke afgeleide van de klasse ScObject er moet worden aangemaakt. Om dit overzichtelijker en uitbreidbaarder te maken dan dat dit in SHaBE is gedaan, wordt er gekozen van het toepassen van het design pattern Factory method.
Dit wordt gedaan omdat dit design pattern geschikt lijkt om de volgende redenen:

1. Het object creëren wordt onafhankelijk van de logica die bepaald welk object er aangemaakt moet worden.
2. Uitbreidbaarheid van SHaBE wordt op deze manier verhoogd, omdat het hierdoor gemakkelijker wordt om nieuwe afgeleide van de klasse ScObject te extraheren.

In dit voorbeeld worden de volgende objecten geëxtraheerd:

1. sc_module
2. tlm_initiator_socket
3. tlm_target_socket

Bij ieder sc_object horen attributen die later moeten worden opgeslagen in de SCMDL. Hieronder een overzicht van de gegevens die uit de SystemC geëxtraheerd moeten worden en in de SCMDL moeten worden opgeslagen.

- name(naam van de variabele waarin de afgeleide van het sc_object in de SystemC code is opgeslagen. Wanneer RAM een afgeleide van sc_object is, is ram01 de name in het c++ voorbeeld: *RAM ram01;*)
- type(most-derived klasse van het object)
- systemc-name(hiërarchische naam, bestaat uit de name en de name van de parents)
- systemc-type(systemc type)
- address(adres van het object)

Omdat er in eerste instantie alleen een name, address en parentaddress nodig zijn om de hiërarchie te kunnen opbouwen, worden alleen deze 3 gegevens geëxtraheerd.
Het parentaddress wordt niet in het SCMDL formaat opgeslagen, maar het parentaddress is wel nodig om de relatie te leggen tussen welke modules er parents zijn van andere modules of initiator en target sockets.

9.5. Breakpoints plaatsen

Om de diverse gegevens over de te extraheren objecten te weten te komen, wordt er gebruik gemaakt van breakpoints. Deze breakpoints kunnen met behulp van GDB/MI worden geplaatst.

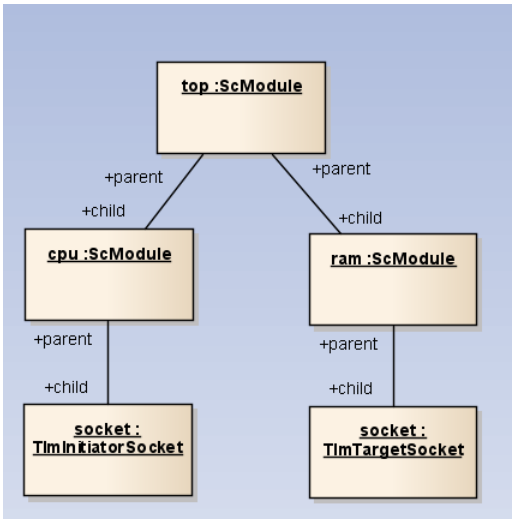
De breakpoint wordt in de constructor geplaatst, omdat in de constructor van “the most base” klasse de naam en het parentaddress van de objecten worden geïnitialiseerd en deze worden vervolgens niet meer veranderd.

Door het eerder besproken design pattern toe te passen kan er ook per te extraheren afgeleide van sc_object een breakpoint worden geplaatst. In SHaBE werd namelijk in de meest abstracte klasse een breakpoint geplaatst (sc_object) en werd er vervolgens door het uitlezen van de callstack omlaag geredeneerd wat de most-derived klasse is. Door een breakpoint in de constructor van de sc_object klasse te gebruiken wordt het breakpoint geraakt bij ieder object wat in SystemC en TLM worden aangemaakt. Door deze top → down benadering is het noodzakelijk geworden om met veel if/else statements het type van het object te proberen te achterhalen en vervolgens zal met veel if/else statements aan de hand van dit type een object van een van de afgeleide klasses van ScObject in SHaBE moeten worden aangemaakt. Hierdoor wordt de onderhoudbaarheid van het programma slechter. Als er per te extraheren klasse een breakpoint kan worden geplaatst waardoor een down → top benadering mogelijk is en er bij het bereiken van het breakpoint al duidelijk is welke afgeleide van sc_object er geëxtraheerd gaat worden.
Het nadeel van deze methode is wel dat er nu afgeleide van de sc_object klasse niet geëxtraheerd worden, als er op de klasse geen breakpoint is geplaatst. Echter staat van te voren vast welke type sc_objecten er geëxtraheerd kunnen worden uit TLM en kan hiermee rekening worden gehouden door in iedere constructor van een te extraheren sc_object een breakpoint te plaatsen.

De opdrachtgever heeft tijdens de ontwikkeling van SHaBE gekozen voor een down-top benadering, omdat dit het voordeel met zich meebracht dat de template argumenten van een klasse via de top → down benadering kunnen worden verkregen door het bekijken van de callstack. Op de callstack staat namelijk welke functie de huidige functie heeft aangeroepen, waardoor de template argumenten kunnen worden opgezocht.
Om deze gegevens te weten te komen kan er op verschillende plekken een breakpoint worden gezet om de gegevens over de sockets te extraheren.
Er is gekozen om de breakpoints in de tlm_base_initiator_socket en tlm_base_target_socket te plaatsen. Dit wordt gedaan, omdat deze klasses de meest abstracte klasses zijn in TLM. Om zelf sockets te definiëren kan de gebruiker zelf een afgeleide van de klasse tlm_target_socket of tlm_initiator_socket definiëren.
Om dat tlm_target_socket en tlm_initiator_socket geen meerwaarde bieden qua functionaliteit en beide geen extra variabele bevatten ten opzichte van tlm_base_target_socket en tlm_base_initiator_socket wordt er gekozen om het breakpoint te plaatsen in de tlm_base_target_socket en tlm_base_initiator_socket constructors. Het voordeel van het plaatsen van het breakpoint op deze plek is, dat het 1 stap minder kost om naar de meest abstracte klasse sc_object te gaan. Uit de sc_object klasse moeten de naam en het adres van de parent van de socket worden geëxtraheerd.

9.6. Hiërarchie opbouwen

Uit voorbeeld 1 worden 3 modules(TOP, RAM en CPU) en 2 sockets(Een initiator en een target socket) geëxtraheerd. Deze modules en socket zijn volgens de rechts afgebeelde boom aan elkaar te koppelen via het address en het parentaddress.



Om deze boomstructuur te implementeren zijn er 2 opties:

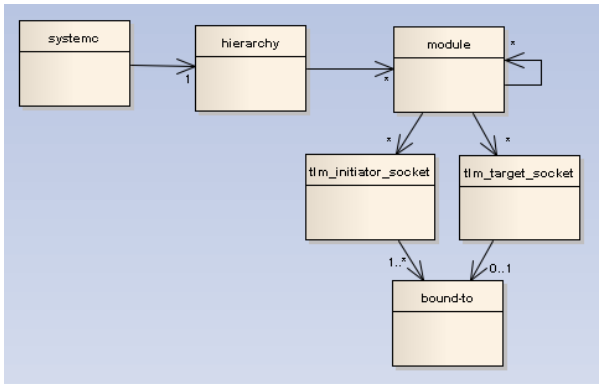
1. Het ScObject object bevat een lijst met alle ScObject's welke een child zijn van het ScObject.
2. Er wordt een TreeNode klasse ontwikkeld welke een verwijzing bevat naar een ScObject en een lijst van verwijzingen naar andere TreeNodes welke verwijzingen bevatten naar ScObject's die child zijn van het ScObject

Er wordt voor optie 2 gekozen, omdat bij optie 2 de objecten van de klasse ScObject puur gebruikt blijven worden om de gegevens die geëxtraheerd zijn in op te slaan. De klasse TreeNode is een klasse die een verwijzing naar child TreeNode's kan bevatten zoals in een boomstructuur. Op dit moment is een module het enige ScObject wat andere ScObject's kan bevatten, dus is het raar als een ScObject een lijst met child ScObjecten gaat bevatten, terwijl maar een van de afgeleide klasses genaamd ScModule child ScObjects heeft. Wanneer ScObject namelijk een lijst gaat bevatten met andere ScObjecten, dan moet ScObject ook een functie krijgen waarmee er ScObjecten worden toegevoegd aan deze lijst.

Wanneer dit wordt gedaan, wordt het OOP Liskov substitution principe geschonden, omdat derived klasses hierdoor juist minder kunnen als in de base klasse wordt gesuggereerd. Niet alle afgeleide klasses van ScObject bevatten namelijk andere ScObject's tijdens de SystemC simulatie en hoeven dus geen functie krijgen waarmee er childs aan hun lijst kunnen worden toegevoegd. Andere optie is om alleen de klasse ScModule een lijst met child ScObject's te geven. Hier wordt niet voor gekozen, omdat het dan nodig is om ieder ScObject te proberen te casten naar een ScModule om te kijken of dit object een ScModule is, als dit een ScModule is kan vervolgens de lijst met de child ScObjects worden opgehaald. Er wordt dus ook niet voor dit alternatief gekozen, omdat casten in dit geval niet nodig hoeft te zijn door het gebruik maken van een TreeNode.

9.7. Opslaan van de hiërarchie

Om de hiërarchie van het geëxtraheerde voorbeeld op te slaan, zal er een ontwerp worden gemaakt voor de uitbreiding van het SCMDL formaat.
Elke module kan net als een sc_port een initiator of een target socket bevatten, daarom worden er twee tags toegevoegd: <tlm_target_socket> en <tlm_initiator_socket>.
Omdat een initiator altijd aan een target is gekoppeld en visa versa wordt er bij de <tlm_initiator_socket> en <tlm_target_socket> tag een child tag toegevoegd genaamd <bound-to> in deze tag staat de naam van de tlm_target_socket of tlm_initiator_socket waaraan deze socket is verbonden.



Hierin zijn een aantal tags gedefinieerd. Hieronder een uitleg.

systemc tag
Om compatible te blijven met de eerder gemaakte versie van SHaBE is deze tag toegevoegd. Systems tag bevat 1 tag genaamd hierarchy.

hierarchy tag
In de hierarchy tag worden de afgeleide van ScObject klasse opgeslagen

module tag
Een van de afgeleide van ScObject is een ScModule. Een object van de klasse ScModule bevat meerdere afgeleide van de klasse ScObject, waardoor een module tag meerdere module, initiator of target sockets tags kan bevatten.

tlm_initiator_socket tag en tlm_target_socket tag
In de tlm_initiator _socket en tlm_target_socket tag wordt de data, zoals het address en de naam van de socket opgeslagen. In de initator tag is altijd minimaal één bound-to tag aanwezig, omdat een initiator altijd aan 1 target socket is gekoppeld. Daarnaast kan een initiator een hierarchical binding hebben.

Dit betekent dat de initiator aan de initiator van de module boven zijn module is gekoppeld.
Een target socket heeft altijd nul of één bound to tag. Een target socket kan namelijk alleen op een hierarchische wijze worden gekoppeld.

bound-to tag
In de bound-to tag wordt gespecificeerd aan welke tlm_initiator _socket of tlm_target_socket de bovenliggende tlm_initiator _socket of tlm_target_socket is verbonden.
Om onderscheid te maken tussen een hierarchical binding(Een verbinding tussen twee sockets op een ongelijk diepte niveau. Dit is een verbinding tussen een initiator en initiator of target en target socket) en een primitive binding(een verbinding tussen initiator en target socket op een gelijk diepte niveau) wordt er in de bound to tag een attribuut gespecificeerd waarmee wordt aangegeven wat voor binding het is.

9.8. Diagrammen
In bijlage D is het klassendiagram te zien welke is ontworpen.
Hieronder een overzicht van functionaliteiten van de klasses.
Het globale overzicht van de extractie is te zien in het sequence diagram in bijlage E. Hierin is te zien dat het HierarchyObjectExtractor object ervoor zorgt dat alle Extractor's worden geïnitialiseerd en dat de breakpoints worden gezet. Het HierarchyObjectExtractor object laat vervolgens de debugger werken en wacht totdat de debugger een breakpoint tegenkomt. Wanneer dit gebeurd wordt de BreakpointHandler aangeroepen die het breakpoint afhandelt. Dit wordt net zolang gedaan totdat het einde van de elaboration fase is bereikt. Nu worden alle geextraheerde ScObject's opgehaald en wordt er door de HierarchyTreeBuilder een HierarchyTree gemaakt. Om de hiërarchie op te slaan als SCMDL bestand wordt de saveAsSCMDL functie aangeroepen van de HierarchyTree.

9.8.1. BreakPointHandler, BreakPoint en CustomBreakPoint
Om een breakpoint in het SystemC model te zetten kan de BreakPointHandler worden gebruikt. Door een afgeleide van de breakPoint klasse aan de register functie van BreakPointHandler mee te geven wordt de BreakPoint geregistreerd. Wanneer een CustomBreakPoint is geregistreerd en de meegegeven functie wordt geraakt, zal de callback functie die is meegegeven aan de constructor van de CustomBreakPoint worden aangeroepen. Dit kan zowel een memberfunctie als een gewone functie zijn. Om een breakpoint te registreren gebruikt de BreakPointHandler de klasse GDB. GDB is een klasse die communiceert met GDB via de GDB/MI die herbruikt wordt uit SHaBE.

9.8.2. HierarchyObjectExtractor, ScObjectExtractor en afgeleide, ScObject en afgeleide en ScObjectInfo
Om verschillende ScObjecten te extraheren wordt er gebruik gemaakt van ScObjectExtractors. Voor iedere ScObject klasse die geëxtraheerd kan worden, wordt er een afgeleide van de klasse ScObjectExtractor aangemaakt. Deze klasse zorgt voor het zetten van de juiste breakpoints en het afhandelen van BreakPoint callbacks. De afgeleiden van de klasse ScObjectExtractor die gebruikt worden, worden bij de start van het programma aan de HierarchyObjectExtractor

toegevoegd. HierarchyObjectExtractor zorgt ervoor dat alle ScObjectExtractors hun breakpoints plaatsen en dat GDB het SystemC model doorloopt en zo nodig het bereiken van een breakpoint doorgeeft aan de BreakPointHandler.

9.8.3. HierarchyBuilder, HierarchyTree en HierarchyNode en ScObject

Zodra de HierarchyObjectExtractor alle ScObject's heeft geëxtraheerd, moet de hiërarchie worden opgemaakt. Dit wordt gedaan door alle ScObject's mee te geven aan de HierarchyBuilder. Deze HierarchyBuilder maakt een nieuwe HierarchyTree aan waarin zich verschillende HierarchyNode's bevinden die ieder naar een ScObject verwijzen.
Door de saveAsSCMDL() functie van de HierarchyTree aan te roepen, zal de hiërarchie worden opgeslagen als SCMDL.

9.9. Implementatie

Tijdens de implementatie zijn er diverse wijzingen gemaakt t.o.v. het gemaakte ontwerp.
Toepassen van de factory method:
Het idee van dit design pattern was om ScObjectInfo objecten die worden aangemaakt tijdens de extractie door te geven aan ieder afgeleide van de ScObjectExtractor klasse, zodat een van de afgeleide van de ScObjectExtractor klasses het ScObjectInfo object gebruikt om de juiste afgeleide van ScObject aan te maken. Tijdens de implementatie bleek echter, dat het niet nodig bleek om ScObjectInfo objecten te gebruiken, omdat iedere afgeleide van de ScObjectExtractor alleen breakpoints plaats bij de punten waarop de afgeleide van ScObjectExtractor zelf een afgeleide van ScObject moet aanmaken. De callback memberfunctie die wordt toegevoegd aan het breakpoint is een memberfunctie van de afgeleide van ScObjectExtractor, waardoor de afgeleide van ScObjectExtractor wordt aangeroepen wanneer deze een bepaald object van een van de afgeleide van de ScObject klasse moet aanmaken.

Tijdens de implementatie hebben zich ook een aantal problemen voortgedaan en moesten er bepaalde keuzes worden gemaakt. Hieronder een overzicht.

Probleem:
http://tdistler.com/2008/11/13/debugging-c-templates-brekpoints-and-gdb
Breakpoint bij functies van een template classes zetten. Template klasses worden namelijk door de compiler omgezet naar de juiste klasses tijdens het compileren.
Voorbeeld:

```
template<typename T>
class Help
{
public:
	T returnMeAValue();
};

Help<int> a;
```

Help<std::string> a;

dan worden er tijdens het compileren 2 klasses aangemaakt. 1 met de template int en een klasse met de template string. Deze worden dan gecompileerd en worden gebruikt in het programma.

Oplossing:
Vraag de functienaam op bij GDB:
b tlm::tlm_base_target_socket(<TAB> te geven.
De debugger zoekt dan zelf alle mogelijkheden van aanvullingen op en returned deze.
Door vervolgens te zoeken naar ::klassenaam(wordt de volledige naam van de constructor gevonden waardoor het zetten van een breakpoint op de constructor mogelijk is.

Alternatief:
Regelnummer gebruiken om een breakpoint op te zetten. Deze oplossing brengt echter het nadeel met zich mee dat het regelnummer van de functies kunnen veranderen wanneer er een volgende versie van TLM wordt uitgebracht. Functienamen daarintegen kunnen ook veranderen, alleen wordt er aangenomen dat dit minder snel zal gebeuren, omdat de functienamen in de TLM standaard zijn vastgelegd.

Probleem:
Lijkt alsof het niet mogelijk is een parent klassenaam te extraheren omdat hier templates in verwerkt zijn.
Oplossing: Probleem lag hier niet aan, maar het commando wat opgegeven wordt met de template naam erin moet geescaped worden, omdat de template argumenten gescheiden kunnen worden door spaties. Bij normale functie namen kunnen er geen spaties aanwezig zijn, waardoor dit probleem zich nooit eerder heeft voortgedaan. \” \”
Aan dit probleem is kostbare tijd verloren gegaan.
Resultaat GDB klasse aangepast.

Probleem:
Keuze:
Als er een breakpoint op de constructor wordt gezet en de breakpoint wordt geraakt, is het nog niet duidelijke welk parent adres en naam de tlm target socket heeft. Dit komt, omdat de base klasses pas aan het eind van de constructor volledig zijn geinitialiseerd.
Oplossing:
GDB gebruiken en wachten totdat de functie is afgelopen.
Dit kan worden gedaan door GDB opdracht te geven om de functie uit te voeren en bij de return call-back te geven.
Een alternatief zou kunnen zijn om een watchpoint te zetten op de variabele parent address en de naam en vervolgens te wachten op een trigger van GDB die aangeeft dat de variabele worden veranderd. Omdat dit alternatief onoverzichtelijk wordt qua implementatie en helemaal niet onderhoudbaar is,

wanneer er later meerdere variabele uit de klasses moeten worden geëxtraheerd, wordt er niet voor dit alternatief gekozen.

Breakpoint later zetten wordt heel ingewikkeld ivm zoeken van de juiste regel etc en met templates. Dus wordt er een commado aan de GDB klasse toegevoegd waarmee GDB runt totdat de functie gaat returnen. Dan wordt er net zoals bij de breakpoints een event getriggerd als de functie klaar is.

Binding wordt geextraheerd door een BindingExtractor. Voor ieder binding van een ScObject is er een extractor zoals een TlmInitiatorSocketBindingExtractor en een TlmTargetSocketBindingExtractor. Deze BindingExtractors extraheren een ExtractedBinding. Dit is een binding tussen adressen van 2 ScObject's.
Nadat de extractie heeft plaatsgevonden kan de binding tussen de 2 ScObjecten worden gemaakt. Omdat niet ieder ScObject een binding kan bevatten, zo bestaat er geen binding tussen een module en een socket, wordt er een nieuwe afgeleide klasse van ScObject gemaakt, ScBindedObject. Klasses zoals TlmInitiatorSocket en TlmTargetSocket zijn afgeleide van ScBindedObject, omdat zij gebind kunnen worden. Een TlmInitiatorSocket kan een primitieve binding en een hiërarchische binding hebben en een TlmTargetSocket kan alleen een hiërarchische binding bevatten. Tijdens dit voorbeeld zijn hiërarchische bindings nog niet van belang en worden daarom bij het bespreken van het ontwikkelen van het volgende voorbeeld besproken.

9.10. Testen

Om te testen wordt er gebruik gemaakt van UnitTest++, TinyXML en ValGrind. M.b.v. UnitTest++ en TinyXML kan er een unit test worden gemaakt. Met deze unit test kan worden gecontroleerd of de gegenereerde SCMDL gelijk is aan de verwachte SCMDL.
Valgrind wordt gebruikt voor het elimineren van memoryleaks. In de programmeertaal C++ kunnen er programma's worden gemaakt waarin dynamische geheugen wordt aanvraagd, zodra dit geheugen niet door het programma wordt vrijgegeven ontstaat er een memory leak. Door het gebruik van Valgrind kunnen deze memory leaks worden opgespoord.
Voor het testen wordt Valgrind op de volgende manier aangeroepen:
valgrind --leak-check=full --show-reachable=yes --tool=memcheck /home/dennis/Bureaublad/workspace/MySHaBE/Debug/MySHaBE

Uit deze test kwam dat het Singleton pattern zoals deze geïmplementeerd is een memory leaks bevat.
Na nader onderzoek komt dit omdat de getInstance() methode er als volgt uitziet:

```
BreakPointHandler* BreakPointHandler::getInstance()
{
	if(_instance == NULL)
		_instance = new BreakPointHandler();
	return _instance;
}
```

Probleem:
De member variabele _instance is een statische private member variabele van de klasse BreakPointHandler van het type BreakPointHandler*.
Deze variabele wordt nooit gedelete, omdat de BreakPointHandler zelf verantwoordelijk is voor het vrijgeven van het geheugen omdat deze klasse het geheugen ook heeft ingenomen. Dit doet de BreakPointHandler nu niet.
Oplossing:
Het geheugen in de getInstance functie niet op een dynamische maar op een statische manier gebruiken. Dit kan door niet een Membervariabele _instance te gebruiken, maar een lokale statische BreakPointHandler. Om niet het returntype van de functie te veranderen wordt het adres van dit lokale object gereturned. Hierdoor blijft de rest van het programma onveranderd, want slecht de implementatie van de getInstance methode is veranderd en niet het returntype. De nieuwe implementatie van getInstance ziet er als volgt uit:

```
BreakPointHandler* BreakPointHandler::getInstance()
{
      static BreakPointHandler brk;
      return &brk;
}
```

Er kan ook voor worden gekozen om aan het eind van de functie main het volgende te doen: `delete BreakPointHandler::getInstance();`
Dit alternatief wordt niet gekozen, omdat het erg raar is om een BreakPointHandler object vanuit main te deleten, als het niet zeker is, ofdat er ook wel eens een BreakPointHandler instance is gebruikt. En ten tweede is de main functie helemaal niet verantwoordelijk voor het vrijgeven van het geheugen welke BreakPointHandler in beslag zou kunnen hebben genomen.

9.10.1. UnitTest++ en TinyXML

In de acceptatietest wordt getest of de SCMDL die door SaTHE wordt gegenereerd wel is opgebouwd volgens de verwachte SCMDL. Tijdens de acceptatietest is er rekening gehouden met de volgende aspecten van het SCMDL formaat.

1. XML elementen in beide SCMDL bestanden hoeven niet in dezelfde volgorde te staan.
2. De adressen van de ScObject's wordt tijdens de SystemC simulatie op een willekeurige manier aangemaakt. Dit betekent dat het tijdens de test niet nodig is om te controleren of de adressen uit de gegenereerde SCMDL van bijvoorbeeld de modules en sockets gelijk zijn aan die van de adressen in de verwachte SCMDL.
3. De adressen worden indirect getest, want als een ScObject een parent heeft, dan moet het address van de parent overeenkomen met het parent address van het ScObject. Als deze niet overeenkomen zou de structuur van de SCMDL niet kloppen. De hiërarchie van de ScObject's uit de SCMDL wordt wel getest in de acceptatie test.

10. Voorbeeld 2 – tlm hiërarchische sockets

10.1. Analyse

Wanneer voorbeeld 2 geëxtraheerd wordt door SaTHE, moeten er hiërarchisch bindings tussen tlm sockets worden geëxtraheerd. Om te controleren of dit de enige wijziging is die gemaakt moet worden, wordt SaTHE gebruikt of het voorbeeld te extraheren. Een gedeelte van dit resultaat is te zien in bijlage B. Hierin is duidelijk te zien dat de hiërarchische bindingen nog niet gemaakt worden. De initiator socket is wel al reeds aan een target socket gebonden, echter zijn de twee initiator sockets nog niet aan elkaar gebonden.

10.2. Ontwerp & Implementatie

Om bindings te extraheren worden er breakpoints in de memberfuncties tlm_base_initiator_socket::bind en tlm_base_target_socket::bind geplaatst.
Dit wordt gedaan door een afgeleide van de BindingExtractor.
Er wordt gekozen om een nieuwe extractor klasse aan te maken voor Bindings, omdat Bindings geen ScObject's zijn en waardoor het dus onlogisch zou zijn om een afgeleide van de ScObjectExtractor te maken waarmee Bindings worden geëxtraheerd. Het ontwerp is te zien in bijlage C.
Er wordt gekozen om een ExtractedBinding klasse aan te maken, waar het adres van het begin van een binding en het adres van het eind van een binding in kan worden opgeslagen. Door het gebruik van een object van de klasse Binder kunnen deze adressen van de bindings worden gekoppeld aan afgeleide klasses van ScBindedObject.
Omdat niet ieder ScObject Bindings kan bevatten, wordt er een klasse ScBindedObject gebruikt. Hiervoor wordt gekozen, omdat het Liskov principe nu niet voor bijvoorbeeld de klasse ScModule wordt geschonden. Een ScModule heeft namelijk geen verbindingen met andere ScObject's.
De in voorbeeld 1 ontwikkelde TlmInitiatorSocket en TlmTargetSocket worden nu afgeleide van ScBindedObject ipv ScObject, omdat beide sockets gebind kunnen worden aan andere sockets. Een TlmInitiatorSocket kan worden gebind aan TlmInitiatorSocket en TlmTargetSocket en een TlmTargetSocket alleen aan een TlmTargetSocket. Om bindings op te slaan, wordt er in ScBindedObject een lijst bijgehouden met objecten waaraan de ScBindedObject is verbonden. Om dit te doen wordt het composite design pattern gebruikt. In ScBindedObject worden daarom ScObject's opgeslagen.
Dit wordt ook gedaan, want hierdoor ontstaat er een abstractere manier om de koppeling tussen ScBindedObjects op te slaan.

10.3. Testen

Om voorbeeld 2 te kunnen testen wordt het bij voorbeeld 1 gemaakte testprogramma uitgebreid zodat ook wordt getest of de bound-to tag wel correct toegevoegd wordt. Voorafgaand aan het uitvoeren van de test wordt er nogmaals een SCMDL bestand gemaakt, waarin de te verwachte testresultaten zijn opgeslagen. Tijdens het extraheren van voorbeeld 2 doet zich een probleem voor wat te maken heeft met de functie select welke in de functie GDB::getsWhileSendingStdinToGdb() aangeroepen wordt. Omdat deze klasse is herbruikt, wordt ervoor gekozen geen aandacht aan dit probleem te besteden en het probleem door te geven aan de auteur van de klasse.

11. Voorbeeld 8 – SystemC ports

11.1. Analyse

In voorbeeld 8 zijn er een aantal nieuwe onderdelen te extraheren, zoals sc_export en sc_port. Deze worden apart gebruikt van andere SystemC objecten in voorbeeld 8 en daarom wordt SaTHE eerste aangepast om voorbeeld 8 te extraheren.
In voorbeeld 8 worden een sc_port en sc_export gebruikt. Deze worden direct aan elkaar gekoppeld(zonder channel ertussen) omdat SystemC channels in de verdere voorbeelden niet worden gebruikt en dus niet geëxtraheerd hoeven te worden. Tevens is SHaBE al in staat om een SystemC channel te extraheren, waardoor het niet nodig is de SystemC channel extractie aan SaTHE toe te voegen.

11.2. Ontwerp

Tijdens het maken van SaTHE is er bewust voor gekozen om een klassenstructuur te gebruiken waarbij er makkelijk nieuwe sc_objecten kunnen worden geëxtraheerd. Dit wordt duidelijk tijdens het maken van het ontwerp om voorbeeld 8 te extraheren. Om de sc_port, sc_export toe te voegen hoeven er slechts 2 afgeleide van de ScObject klasse, ScExport en ScPort, te worden gemaakt en twee afgeleide van de ScObjectExtractor, , te worden gemaakt, om ScExport's en ScPort's te extraheren.
Tijdens het maken van het ontwerp wordt er nog niet gelet op de binding tussen de sc_port en sc_export. Dit wordt verplaatst naar het ontwerp van voorbeeld 3

11.3. Implementatie

Tijdens de implementatie heeft zich een probleem voortgedaan, namelijk dat er sc_port's en sc_export's uit tlm_initiator_socket's en tlm_target_socket's worden geëxtraheerd. Dit komt doordat tlm sockets uit sc_port's en sc_export's bestaan. Om te voorkomen dat de tlm sockets niet als sc ports geëxtraheerd worden, wordt de BreakPointHandler klasse aangepast, zodat deze een functie krijgt waarmee alle breakpoints tijdelijk kunnen worden uitgeschakeld. Door de extractie functie van de sockets aan te passen, zodat alle breakpoints worden uitgeschakeld wanneer er een socket wordt geëxtraheerd. Wanneer de tlm_initiator_socket of tlm_target_socket constructor wordt uitgevoerd worden ook de sc_port en sc_export constructor in de tlm_initiator_socket of tlm_target_socket uitgevoerd, omdat dit base klasses van de tlm sockets zijn. Door alle breakpoints uit te schakelen wanneer het breakpoint in de tlm_initiator_socket of tlm_target_socket is bereikt, worden er geen sc_port's en sc_export's geëxtraheerd.
Tijdens de implementatie wordt duidelijk dat de bindingen tussen sc_port en sc_export heel anders worden gemaakt dan tussen sockets. In beschrijving van de ontwikkeling van voorbeeld 3 wordt beschreven hoe deze bindingen zijn ontwikkeld.

11.4. Testen

Om het voorbeeld te testen wordt de huidige test aangepast zodat ook sc_port's en sc_export's kunnen worden getest.

12. Voorbeeld 3 – tlm fifo

12.1. Analyse

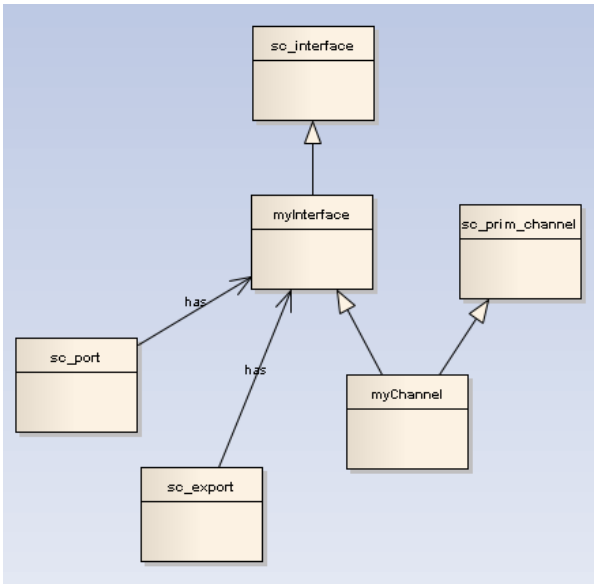
In de analysefase van voorbeeld 8 is uitgelegd dat sc_port's en sc_export's nodig zijn om de bindingen met tlm_fifo's te maken. Wanneer voorbeeld 3 gebruikt wordt tijdens de extractie is dit te zien.
De tlm_fifo die in voorbeeld 3 wordt gebruikt, wordt niet geëxtraheerd, ook niet als een ander sc_object. Dit komt omdat een tlm_fifo een sc_prim_channel(primary channel) is en sc_prim_channel's worden (nog) niet door SaTHE geëxtraheerd.
Omdat er een kleine kans is dat SHaBE dit voorbeeld wel zou kunnen extraheren, omdat SHaBE sc_port's, sc_export's en primary channels kan extraheren, wordt dit gecontroleerd. Er blijkt dan echter dat ook SHaBE de tlm_fifo niet kan extraheren, ook niet als primary channel. Omdat SHaBE dit niet kan, wordt de extractie door SaTHE ontwikkeld.

12.2. Ontwerp

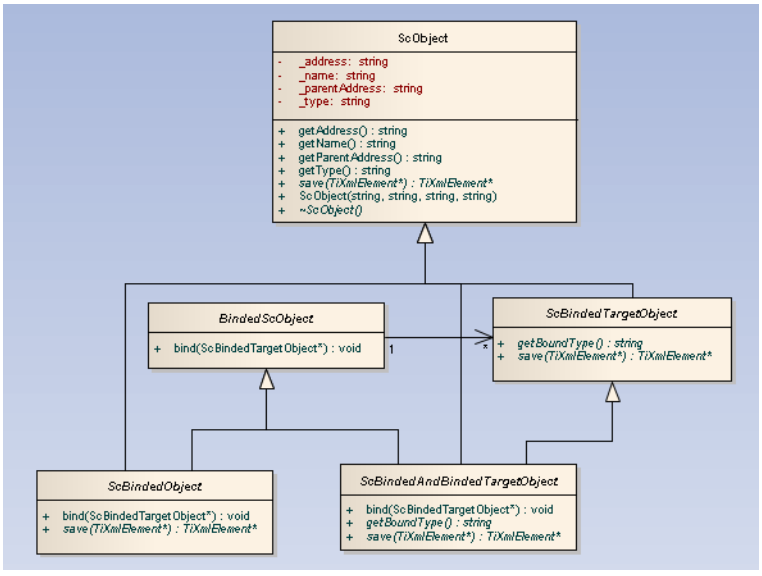
Tijdens het ontwerp wordt er rekening mee gehouden dat een tlm_fifo niet het enige primary channel is wat door SaTHE geëxtraheerd moet gaan worden, want door SHaBE worden er al verschillende primary channels geëxtraheerd. Wanneer SaTHE uitgebreid wordt met functionaliteiten van SHaBE is het erg makkelijk om al mechanismes ontworpen te hebben waarmee primary channels kunnen worden geëxtraheerd. Dit komt de uitbreidbaarheid ten goede.
Omdat iedere primary channel andere data kan bevatten, wordt er een abstracte klasse ScPrimaryChannel aangemaakt, waarin de data die bij ieder primary channel hoort wordt opgeslagen. Omdat bijvoorbeeld een tlm_fifo een apart soort primary channel is waaruit bijvoorbeeld ook de grootte van de fifo kan worden geëxtraheerd, wordt er een TlmFifoExtractor ontwikkeld waarmee specifiek alleen tlm_fifo objecten worden geëxtraheerd en er wordt een afgeleide van ScObject genaamd TlmFifo ontwikkeld waar de geëxtraheerde data uit een tlm_fifo in kan worden opgeslagen.
Om in de SCMDL onderscheid te kunnen maken tussen een primary channel uit SystemC en een primary channel uit TLM, kan er een tlm_primary_channel tag worden toegevoegd aan de SCMDL. Dit wordt niet gedaan, omdat in het attribute type van een primitive-channel tag al af te leiden is of het een systemc type of een tlm type is. De primitive-channel tag is een tag die al gedefinieerd is in SCMDL. In deze tag worden de door SHaBE geëxtraheerd primary channels in opgeslagen.

Omdat tijdens het extraheren van het vorige voorbeeld geen rekening is gehouden met het extraheren van de bindingen, wordt dat wel tijdens de ontwikkeling van de extractie van dit voorbeeld gedaan. sc_port's kunnen namelijk worden gekoppeld aan sc_export's en interfaces en daarom is het van belang om deze te extraheren.

De koppeling van een sc_port aan een tlm_fifo is heel anders als de koppeling van een tlm_initiator_socket aan een tlm_target_socket. Een sc_export wordt namelijk gebonden aan een sc_interface. Wanneer een sc_port aan een sc_export wordt gebonden, wordt de sc_interface van de sc_export opgehaald en de sc_port wordt aan deze sc_interface gekoppeld. Hierdoor ontstaat onderstaande structuur.



Zoals gezegd is een tlm_fifo een sc_prim_channel en de in SystemC gebruikte channels erfen gewoonlijk over van een interface en van een sc_channel of sc_prim_channel. Wanneer het adres van de tlm_fifo wordt vergeleken met de adressen waaraan de sc_port's zijn verbonden, komen deze niet overeen. Wanneer het adres van de interfaces van de tlm_fifo worden vergeleken met de adressen waaraan de sc_port's zijn gekoppeld komen deze wel overeen!
Dit betekent dus dat wanneer een ScObject een Binding kan ontvangen deze Binding niet alleen op het adres van de this pointer hoeft te zijn aangesloten.
Dit komt door de multiple inheritance. Hierbij is de this pointer niet gelijk aan beide base klasses.
Om deze reden wordt er een nieuwe structuur opgezet waarbij ScBindedObject's alleen gekoppeld kunnen worden aan afgeleide van ScBindedTargetObject's. Het model van deze structuur is in onderstaande afbeelding te zien.



Omdat er objecten zoals een TlmInitiator zijn die zowel Bindings kunnen aangaan als Bindings kunnen ontvangen is er een klasse ScBindedAndBindingTargetObject waar ScObject's van af kunnen worden geleid die Bindings kunnen aangaan en kunnen ontvangen. De reden waarom BindedScObject niet overerft van een ScObject is omdat er dan als er een ScBindedAndBindedTargetObject wordt aangemaakt, deze 2 dezelfde base klasses heeft, waardoor er redundante gegevens worden opgeslagen.

De klasse tlm_fifo heeft standaard al 10 verschillende interfaces. Dit zijn allemaal directe en indirecte afgeleide van sc_interface. De grafische weergave hiervan is te zien in bijlage F. Interfaces maken ook gebruik van multiple inherintance waardoor het noodzakelijk is om ook in de interfaces op zoek te gaan naar hun base klasses en de adressen van de base klasse objecten te extraheren. De

adressen van al de interfaces worden opgeslagen in het ScBindedTargetObject. Om de interfaces te extraheren moet er in de breakpoint handle functie, die wordt aangeroepen wanneer er een nieuwe tlm_fifo wordt aangemaakt, worden gezocht of er een afgeleide van tlm_fifo wordt gebruikt en daardoor misschien nog meer interfaces gebruikt. Zodra de meest derived klasse is gevonden(in het voorbeeld gewoon tlm_fifo) worden alle base klasses opgezocht en daarvan ook weer alle bases klasses, etc. Wanneer de most base class is gevonden, wordt er gecontroleerd of dit een afgeleide van sc_interface is, zoja, dan moeten al de adressen van deze interfaces worden opgeslagen. Dit ziet er in pseudocode in een recursieve functie als volgt uit:

```
function boolean isScInterface(classname)
{
	if(classname == sc_core::sc_interface)
		return true;
	else
		for each baseclass of classname as baseclass
		{
			if(isScInterface(baseclass))
			{
				address = getAddress(baseclass);
				store address;
				return true;
			}
		}
	return false;
}
```

12.3. Implementatie

Tijdens de implementatie is ervoor gekozen om de tlm_fifo te extraheren door een breakpoint op de functie tlm_fifo::init() te plaatsen. Deze init functie wordt in de constructor aangeroepen. Door de init functie te gebruiken en niet de constructor wordt er voorkomen dat er eerst een breakpoint op de constructor moet worden gezet en er vervolgens moet worden gewacht totdat de constructor is beëindigd. Door het breakpoint direct op de init functie te zetten kan er gelijk worden begonnen met de extractie zonder dat er gewacht moet worden totdat alle variabele van de base klasse sc_object zijn geïnitialiseerd.
Om vervolgens alle mogelijke interfaces te extraheren van een tlm_fifo is het nodig dat alle interfaces van alle sc_exports worden geëxtraheerd en dat deze in de TlmFifo worden opgeslagen. Dit zijn namelijk de interfaces waarmee een buitenstaande sc_port kan worden verbonden aan de tlm_fifo.

12.4. Testen

Omdat de sc_port's en sc_export's aan channels en interfaces zijn verbonden ipv direct aan elkaar, wordt er in de SCMDL een primitive channel tag gebruikt waaraan zowel een sc_port als sc_export verbonden kan zijn.

13. Voorbeeld 4 – tlm-req-resp-channel

13.1. Analyse

Wanneer voorbeeld 4 wordt geëxtraheerd m.b.v. SaTHE worden er diverse geëxtraheerde gegevens ontdekt. De SCMDL van de extractie is in bijlage G te vinden.
Wanneer er naar de TLM source code wordt gekeken blijkt een tlm_req_resp_channel een afgeleide van een sc_module te zijn. De module bevat 6 sc_export's en 2 tlm_fifo's. Wanneer dat wordt vergeleken met het SCMDL bestand dat tijdens de analyse is geëxtraheerd, blijken de resultaten geheel overeen te komen.

13.2. Ontwerp

Zoals tijdens de analyse is ontdekt, is een tlm_req_resp_channel een speciaal soort channel welke als module is geimplementeerd. Omdat een tlm_req_resp_channel een speciaal soort module is wordt ervoor gekozen, om de tlm_req_resp_channel niet als gewone module op te slaan in de SCMDL maar als tlm_req_resp_channel. Hiervoor wordt de tag tlm_req_resp_channel gebruikt. Om de koppeling van deze channel met andere modules en porten wel volledig te ondersteunen is het nodig om de sc_export's die zich in de tlm_req_resp_channel bevinden in de tlm_req_resp_channel tag op te slaan. Alleen dan kan er in de port tag een bound-to tag worden gespecificeerd waarin de naam van de juiste sc_export is opgeslagen. Er had ook gekozen kunnen worden om de bound-to tag in een sc_port op te slaan zodat deze aan een tlm_req_resp_channel is gekoppeld. Hiermee wordt echter de naam en het type van de export uit de tlm_req_resp_channel waaraan de sc_port is gekoppeld niet opgeslagen. Hierdoor gaat er informatie uit het SystemC model verloren. Op deze manier lijkt het ook ofdat de tlm_req_rsp_channel alle interfaces bevat, terwijl het juist alle sc_export's in de tlm_req_rsp_channel zijn die de interfaces bevatten.
De tlm_fifo's die in de tlm_req_resp_channel aanwezig zijn, zijn in de TLM code als protected gedeclareerd, waardoor het niet nodig is om deze in de tlm_req_resp_channel op te slaan, omdat de tlm_fifo's niet van buiten de module in TLM gebruikt kunnen worden.

13.3. Implementatie

Tijdens de implementatie wordt er gekozen om dezelfde methodes van extractie toe te passen. Omdat de tlm_req_resp_channel een sc_module en de sc_module niet geëxtraheerd moet worden, maar de tlm_req_resp_channel wordt er gebruik gemaakt van het tijdelijk uit en aanzetten van breakpoints.
Tijdens de analyse zijn er gegevens geëxtraheerd die anders waren als gedacht. Zo zouden de sc_exports geen child van top moeten zijn maar childs van tlm_req_resp_channel. Tijdens de implementatie wordt deze fout in de extractie gevonden. Wanneer de variabele this op de stack wordt gebruik om het huidige address op te zoeken moet er niet frame 1 maar frame 0 worden gebruikt!
Om een tlm_req_rsp_channel te extraheren wordt er een breakpoint in de constructor en in de bind_export functie die in de constructor wordt aangeroepen gebruikt. Dit wordt gedaan, omdat de sc_export's die in de tlm_req_rsp_channel constructor geïnitialiseerd worden eerst moeten worden geëxtraheerd, terwijl de base klasse, sc_module, juist niet geïnitialiseerd moet worden. Door bij het raken

van de breakpoint in de constructor de extractie van sc_module uit te schakelen en vervolgens GDB verder te laten runnen, kunnen de tlm_fifo's en sc_export's worden geëxtraheerd. De tlm_fifo blijkt ook te moeten worden geëxtraheerd, omdat de tlm_fifo de interface bevat waaraan de sc_export's gekoppeld zijn en zij geven de interface door aan een sc_port.

13.4. Testen

Uitvoer van SaTHE is getest en de uitvoer bleek te kloppen met de verwachte uitvoer.

14. Voorbeeld 5 - tlm-transport-channel

14.1. Analyse

Een tlm transport channel is net als een tlm req rsp channel een channel. De tlm transport channel is net als de tlm req rsp channel geimplementeerd in TLM als afgeleide van een sc_module.

14.2. Ontwerp

De export's en de tlm req rsp channel die in de tlm transport channel zitten moeten worden geëxtraheerd, omdat sc_port's alleen met de sc_export's verbonden kunnen worden zodra de sc_export verbonden is met een interface. In dit geval is de interface een tlm fifo uit de tlm req rsp channel. Daarom wordt er gekozen om ook de tlm req rsp channel in de tlm transport channel te extraheren. Hiervoor wordt hetzelfde in- en uitschakel mechanisme gebruikt als er wordt gebruikt om een tlm req rsp channel te extraheren.

14.3. Implementatie

Tijdens de implementatie van de extractie van dit voorbeeld is er besloten dat het beter is om te stoppen met het extraheren van dit voorbeeld omdat de opdrachtgever het belangrijker vindt om de convenience sockets te extraheren uit voorbeeld 9 t/m 12.

14.4. Testen

Het gedeelte van de extractie dat geïmplementeerd is, is getest door de te testen SCMDL aan te passen.

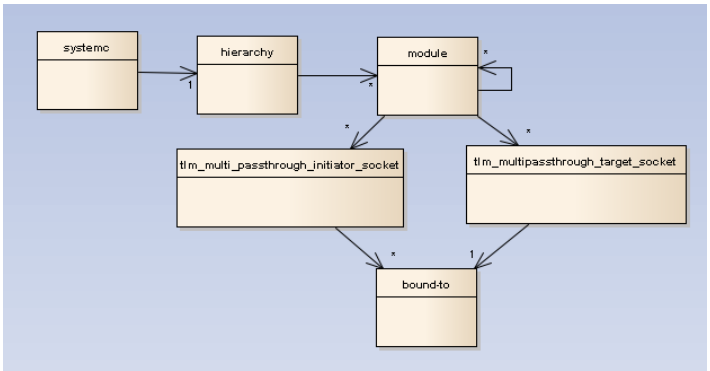
15. Voorbeeld 6 – multi sockets

15.1. Analyse

tlm_multi_passthrough_ socket's zijn sockets die meerdere verbindingen kunnen ontvangen als kunnen aangaan. Deze sockets moeten apart worden geextraheerd, omdat het sockets zijn met een andere werking dan de sockets die er tot nu toe zijn geëxtraheerd.

15.2. Ontwerp

Om multi passthrough sockets te kunnen extraheren, wordt er een breakpoint in de constructor van de multi passthrough socket gezet. Omdat een multi passthrough socket een afgeleide klasse is van een tlm socket, moet de extractie van een tlm socket worden uitgeschakeld wanneer het breakpoint in de constructor wordt geraakt.



De hiërarchie van de SCMDL kan dan worden opgebouwd zoals te zien in de rechter afbeelding. Een tlm multi passthrough initiator socket kan namelijk aan meerdere socket verbonden zijn, terwijl een tlm multi passthrough target socket juist door meerdere andere sc_object's verbonden kan zijn. Hierdoor kunnen dus meerdere sc_object's met 1 tlm multi target socket verbonden zijn.

15.3. Implementatie

Zoals hierboven is beschreven kan een tlm multi passthrough socket op bijna dezelfde manier worden geëxtraheerd als een gewone tlm socket. Dit geldt ook voor de bindingen tussen de multi passthrough en tlm sockets.

15.4. Testen

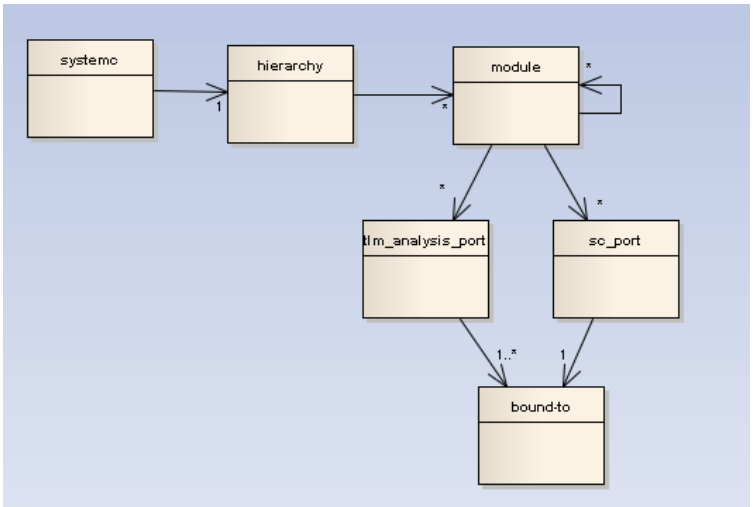
Tijdens de testfase zijn er geen problemen ontstaan.

16. Voorbeeld 7 – tlm_analysis_port

16.1. Analyse

In voorbeeld 7 wordt er een tlm_analysis_port gebruikt. Een tlm_analysis_port is qua gebruik niet veel anders dan een gewone port. Er kan door de analysis port alleen data naar de verbonden ports worden geschreven. Een analysis port kan echter wel verbonden zijn met meerdere andere ports.

16.2. Ontwerp



Een analysis port kan dus worden verbonden met meerdere ports. Hierdoor is het noodzakelijk om meerdere bound-to tags in de tag van een analysis port te kunnen opslaan in SCMDL. De structuur van de SCMDL ziet er dan als volgt uit.

Om de tlm_analysis_port te extraheren wordt er een TlmAnalysisPortExtractor en een TlmAnalysisPort aan SaTHE toegevoegd. De TlmAnalysisPortExtractor wordt afgeleid van de ScObjectExtractor net als alle andere SystemC extractors. Deze extractor maakt vervolgens een TlmAnalysisPort aan wanneer er een tlm_analysis_port wordt geextraheerd. Het breakpoint wordt in de constructor geplaatst. Wanneer het breakpoint is geraakt wordt er gewacht totdat de constructor is beëindigd en worden de variabele uit het ScObject geëxtraheerd.

Net zoals sc_port's en sc_export's worden tlm_analysis_port's met elkaar verbonden door een interface. Deze is bij de standaard tlm_analysis_port altijd hetzelfde, maar het adres kan verschillen per simulatie, waardoor het noodzakelijk is om deze interface te extraheren en net zoals in de sc_export's en primary channels op te slaan.

16.3. Implementatie

Tijdens de implementatie is duidelijk geworden dat tlm_analysis_port's bijna op precies dezelfde manier kunnen worden geëxtraheerd als sc_export's.

16.4. Testen

Om voorbeeld 6 te testen is er net zoals voor de andere voorbeelden een test gemaakt. Tijdens het uitvoeren van de test is echter wel opgemerkt dat er nog niet getest wordt of een tlm_analysis_port aan een sc_export gekoppeld kan worden.

17. Voorbeeld 9, 10 en 11 – Convenience sockets

17.1. Analyse

In voorbeeld 9, 10 en 11 worden verschillende convenience sockets gebruikt. Hieronder een lijst met de volledige namen:

1. simple_initiator_socket
2. simple_target_socket
3. simple_initiator_socket_tagged
4. simple_target_socket_tagged
5. passthrough_target_socket
6. passthrough_target_socket_tagged

Daarnaast zijn er 2 multi sockets die ook convenience sockets worden genoemd:
multi_passthrough_initiator_socket
multi_passthrough_target_socket
Deze zijn echter al gebruikt voor extractie in voorbeeld 6 en worden daarom niet in deze voorbeelden gebruikt.
Al deze sockets zijn afgeleide sockets van de tlm_initiator of tlm_target_socket. Wanneer een tlm socket een convenience socket is, dan moet deze als convenience socket worden geëxtraheerd en niet als tlm_initiator of tlm_target_socket daarom wordt er tijdens de extractie een mechanisme gebruikt waarmee er per Extractor kan worden aangegeven of deze andere Extractors moet blokkeren. Hiermee kan dan worden voorkomen dat een tlm target socket en een afgeleide hiervan als 2 verschillende sockets worden geëxtraheerd.

17.2. Ontwerpfase

In de vorige hoofdstukken is er al gesproken over een zogenaamd mechanisme om andere extractors uit te schakelen. Dit mechanisme is in deze fase uitgebreid, zodat bijvoorbeeld bindingen die in een simple initiator socket worden gemaakt niet ook geëxtraheerd worden(omdat deze bindingen alleen intern gebruikt worden).

17.3. Implementatie

Doordat de extractie van tlm sockets al opgezet was, kon de extractie van de in 17.1 genoemde convenience sockets gemakkelijk toegevoegd worden. De extractie van deze sockets lijkt erg veel op elkaar.

17.4. Testen

Doordat er in de vorige voorbeelden geen onderscheid werd gemaakt tussen de extractie van een tlm socket of afgeleide, moeten de tests van deze voorbeelden nu aangepast worden.

18. Voorbeeld 12 – Hiërarchische channel

18.1. Analyse

In voorbeeld 12 wordt gebruik gemaakt van socket, waarbij de afhandeling van een ontvangen bericht door de module waar de socket in zit wordt gedaan. In sommige voorbeelden uit de TLM standaard voorbeelden wordt dit ook gebruikt. Om de module een bericht af te laten handelen, moet de module overerven van de forward of backward tlm interface.

18.2. Ontwerp

Wanneer dit voorbeeld wordt geëxtraheerd moeten er twee hiërarchische channels ontstaan, omdat er tussen de socket en de module nu ook een verbinding is ontstaan.

18.3. Implementatie

Om dit voorbeeld te kunnen extraheren, moeten er ook interfaces van een module worden geëxtraheerd. De socket is namelijk verbonden aan een object van een klasse die overerft van de benodigde interface, de sc_module en tlm interface. In SHaBE wordt dit een hiërarchische channel genoemd. Hieronder een stuk van voorbeeld 12 waarin er een hiërarchische channel wordt gebruikt.

```
using namespace tlm;
using namespace sc_core;
class RAM: public tlm_fw_transport_if<>, public sc_module
{
public:
        tlm_target_socket<> socket;
        SC_CTOR(RAM): socket("RAMSocketNAME"){
                socket.bind(*this);
        }
        virtual ~RAM();

        void b_transport(tlm_generic_payload& trans, sc_time& t);
        tlm_sync_enum nb_transport_fw(tlm_generic_payload& trans, tlm_phase& p, sc_time& t);
        bool get_direct_mem_ptr(tlm_generic_payload& trans, tlm_dmi&  dmi_data);
        unsigned int transport_dbg(tlm_generic_payload& trans);
};
```

Op de regel `socket.bind(*this);` is te zien dat een tlm socket een binding aangaat met een object van een klasse die is afgeleid van een tlm interface en van een sc_object. Omdat het object waarmee verbonden wordt overerft van de juiste interface, wordt dit gezien als de implementatie van de callback functies van de tlm socket. De klasse RAM zorgt dan ook voor het afhandelen van de transactie.

Tijdens de implementatie van de extractie van van de verschillende voorbeelden zijn er verschillende problemen ontstaan. Hieronder een overzicht van de problemen.

Zoals te zien in het stuk code hierboven erft de klasse RAM over van 2 klasses, tlm_fw_transport_if en sc_module.

```
class RAM: public tlm_fw_transport_if<>, public sc_module
```

Wanneer RAM volgens bovenstaande regel wordt afgeleid van deze klasses ontstaat er geen probleem, want wanneer de constructor van RAM wordt

aangeroepen wordt eerst de base klasse interface geïnitialiseerd en daarna wordt de base klasse sc_module geïnitialiseerd. Wanneer er een sc_module wordt aangemaakt, wordt er een breakpoint geraakt, omdat dan de constructor van sc_module wordt aangeroepen. Wanneer de RAM module wordt geëxtraheerd wordt er gekeken wat de most-derived klasse is en in welke variabele of membervariabele deze is opgeslagen.
Vervolgens wordt er gekeken of RAM is afgeleid van een afgeleide van sc_interface en wordt de variabele gecast naar al base klasses van RAM die een afgeleid van sc_interface zijn.
Maar zodra RAM overerft van een `public virtual tlm_fw_transport_if<>` dan blijkt dat het opvragen van alle base klasses van tlm_fw_transport_if niet meer kan en GDB geeft dan de volgende foutmelding "`virtual baseclass botch`"
Hierdoor is het onmogelijk geworden om de interfaces van de module te extraheren, omdat er niet meer achterhaald kan worden of tlm_fw_transport_if in dit geval een afgeleide van sc_interface is, waardoor het hiërarchische channel niet altijd kan worden geëxtraheerd.

Een ander probleem is dat wanneer RAM overerft van de tlm interface en de sc_module dat het overerven in deze volgorde moet `class RAM: public tlm_fw_transport_if<>, public sc_module`. Zodra de interface en sc_module worden omgedraaid en de tlm interfaces worden geextraheerd, dan wordt er ook een foutmelding "`virtual baseclass botch`" door GDB teruggegeven. Hoe dit kan is onduidelijk, omdat er nu helemaal geen virtuele klasses gebruikt worden in het voorbeeld, alleen in de TLM source code. Omdat de foutmelding nu al 2 keer is voorgekomen en het een redelijk kritische fout is voor SaTHE wordt er contact opgenomen met het GDB ontwikkelteam om de GDB fout te proberen op te lossen.

Tijdens de implementatie fase wordt er een nieuwe versie van GDB, versie 7.3, uitgebracht. Het GDB ontwikkelteam heeft aanbevolen om deze versie te proberen. Wanneer er nu wordt overerft van een virtuele klasse dan treedt de "`virtual baseclass botch`" melding niet meer op. Het voorbeeld kan dan echter om een andere nog onbekende reden niet volledig geëxtraheerd worden.

18.4. Testen
Omdat het voorbeeld niet kon worden geëxtraheerd op de manier hoe het voorbeeld in eerste instantie was geïmplementeerd is de test aangepast.

19. Conclusie

Uit het onderzoek naar de extractie van TLM kan worden geconcludeerd dat TLM kan worden geëxtraheerd uit een SystemC model wanneer er gebruik wordt gemaakt van de dynamische aanpak m.b.v. GDB. Aan de extractie van TLM door SaTHE zit op dit moment wel een beperking, omdat GDB op dit moment de virtuele base klasses van een C++ klasse nog niet altijd kan extraheren. Hierdoor kunnen bijvoorbeeld de hiërarchische channels tussen een tlm socket en een sc_module niet in alle gevallen worden geëxtraheerd.

20. Aanbevelingen

Tijdens het onderzoek zijn er diverse keuzes gemaakt, waardoor er bepaalde aspecten van SaTHE nog niet geheel geïmplementeerd zijn:

Extractie van de C++ variabelenaam wanneer dit een pointer naar een object is of wanneer het een array van pointers naar objecten is.

Op dit moment wordt er gegokt wat de juiste variabelenaam zou kunnen zijn, door altijd de eerste variabelenaam die van het juiste type is te extraheren. Wanneer er echter gebruik wordt gemaakt van bijv Base* b = new Derived, dan werkt deze techniek niet. De beste oplossing zou zijn om de adressen of de SystemC naam van het sc_object te vergelijken zodra de constructor van het object is beëindigd.

Extractie van virtuele base klasses

Op dit moment ondersteund GDB het opzoeken van de virtuele base klasse objecten van een object nog niet. Wanneer GDB dit wel ondersteund is het aan te raden om de extractie van de adressen van de virtuele base klasse objecten aan te passen. Dit probleem is doorgegeven aan het GDB ontwikkelteam en is te zien in deze thread http://sourceware.org/bugzilla/show_bug.cgi?id=13041

21. Bronnen

Informatie over TLM en afbeeldingen van voorbeeld 3 en 4
http://ens.ewi.tudelft.nl/Education/courses/et4351/SystemC-TLM.pdf

Lussy
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.93.1467&rep=rep1&type=pdf

EASY example
http://www.cs.rice.edu/~vardi/comp607/chapter2.pdf

TLM Examples
http://class.ece.iastate.edu/cpre588/documents/RosSwa05A.pdf

22. Bijlagen

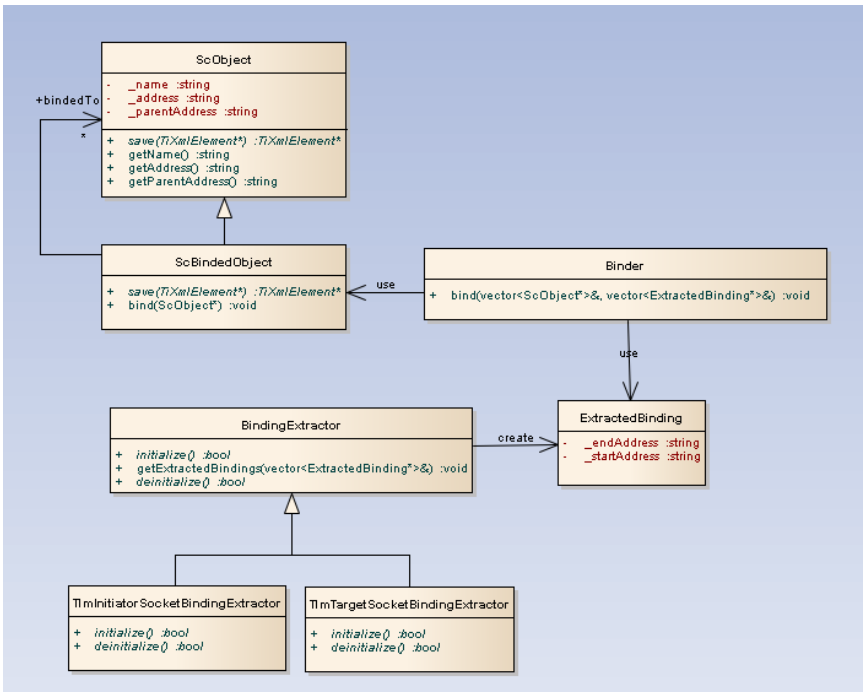
22.1. Bijlage A Voorbeeld 1 Extractie Analyse

22.2. <?xml version="1.0" encoding="UTF-8" ?>
<systemc-model xmlns="http://shabe.sourceforge.net/systemc-model" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://shabe.sourceforge.net/systemc-model http://shabe.sourceforge.net/systemc-model/systemc-model.xsd" name="/home/dennis/Bureaublad/workspace/TLMExample01/Debug/TLMExample01">
  <hierarchy>
    <module name="top" type="Top" systemc-name="Top" systemc-type="sc_module" address="0xbffff3a8">
      <module name="*cpu" type="CPU" systemc-name="Top.CPU" systemc-type="sc_module" address="0x80a7948">
        <port name="" type="sc_core::sc_port<tlm::tlm_fw_transport_if<tlm::tlm_base_protocol_types>, 1, (sc_core::sc_port_policy)0>" systemc-name="Top.CPU.CPUInitiatorSocket" systemc-type="sc_port" address="0x80a79a8" />
        <export name="" type="sc_core::sc_export<tlm::tlm_bw_transport_if<tlm::tlm_base_protocol_types> >" systemc-name="Top.CPU.CPUInitiatorSocket_export_0" systemc-type="sc_export" address="0x80a79d0" />
      </module>
      <module name="*ram" type="RAM" systemc-name="Top.RAM" systemc-type="sc_module" address="0x80aabd8">
        <port name="" type="sc_core::sc_port<tlm::tlm_bw_transport_if<tlm::tlm_base_protocol_types>, 1, (sc_core::sc_port_policy)0>" systemc-name="Top.RAM.RAMTargetSocket_port_0" systemc-type="sc_port" address="0x80aac50" />
        <export name="" type="sc_core::sc_export<tlm::tlm_fw_transport_if<tlm::tlm_base_protocol_types> >" systemc-name="Top.RAM.RAMTargetSocket" systemc-type="sc_export" address="0x80aac38" />
      </module>
    </module>
  </hierarchy>
  <behavior />
</systemc-model>

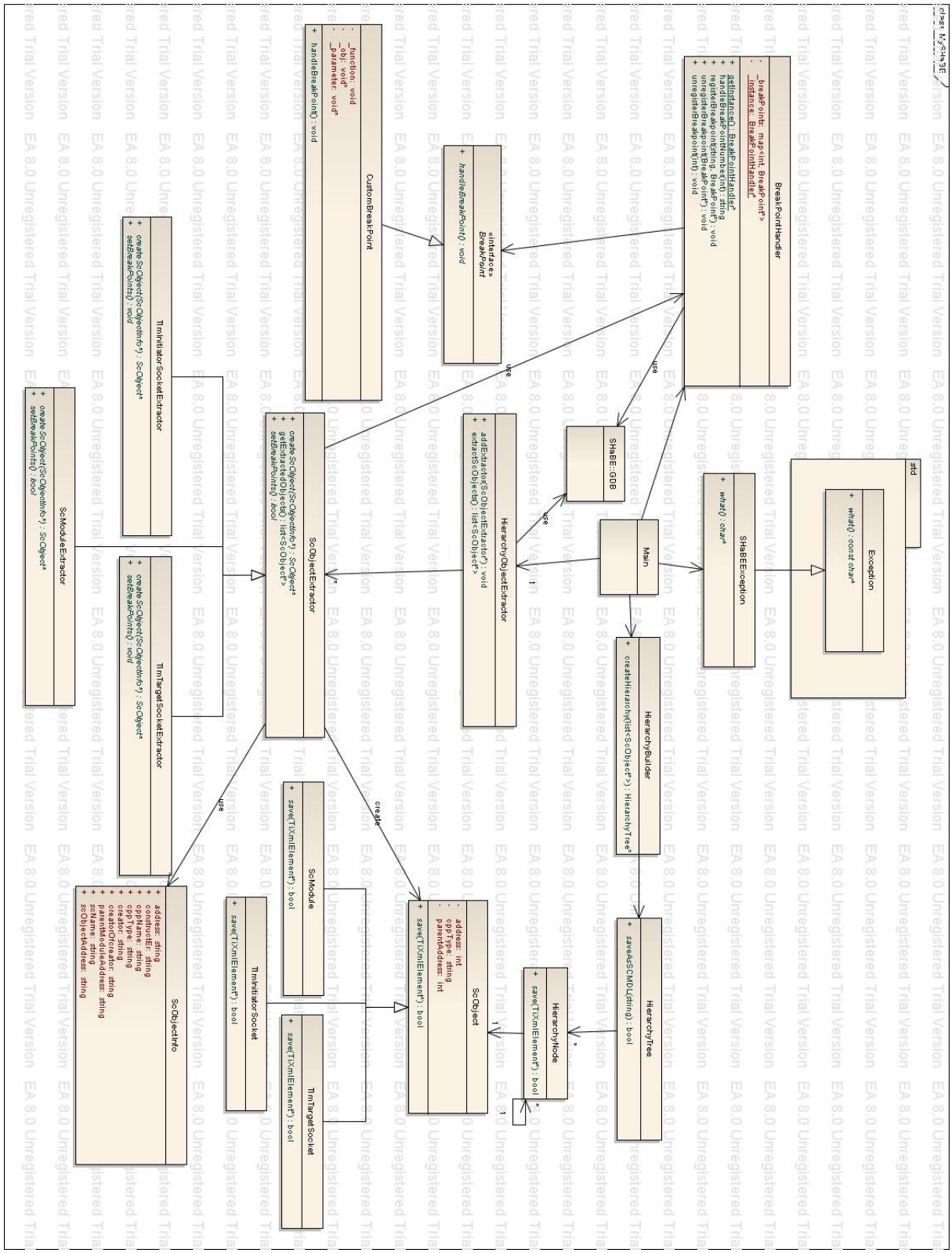
22.3. Bijlage B Voorbeeld 2 Extractie Analyse

<module systemc-name="Top.transmitter_2" address="0x80bb388">
        <module systemc-name="Top.transmitter_2.Sub" address="0x80bb3e4">
          <tlm_initiator_socket systemc-name="Top.transmitter_2.Sub.SubTransmitterSocket" address="0x80bb440" />
        </module>
        <tlm_initiator_socket systemc-name="Top.transmitter_2.TransmitterSocket" address="0x80bb4a4">
          <bound-to to="tlm_target_socket" systemc-name="Top.router.targetsocket_2" />
        </tlm_initiator_socket>
</module>

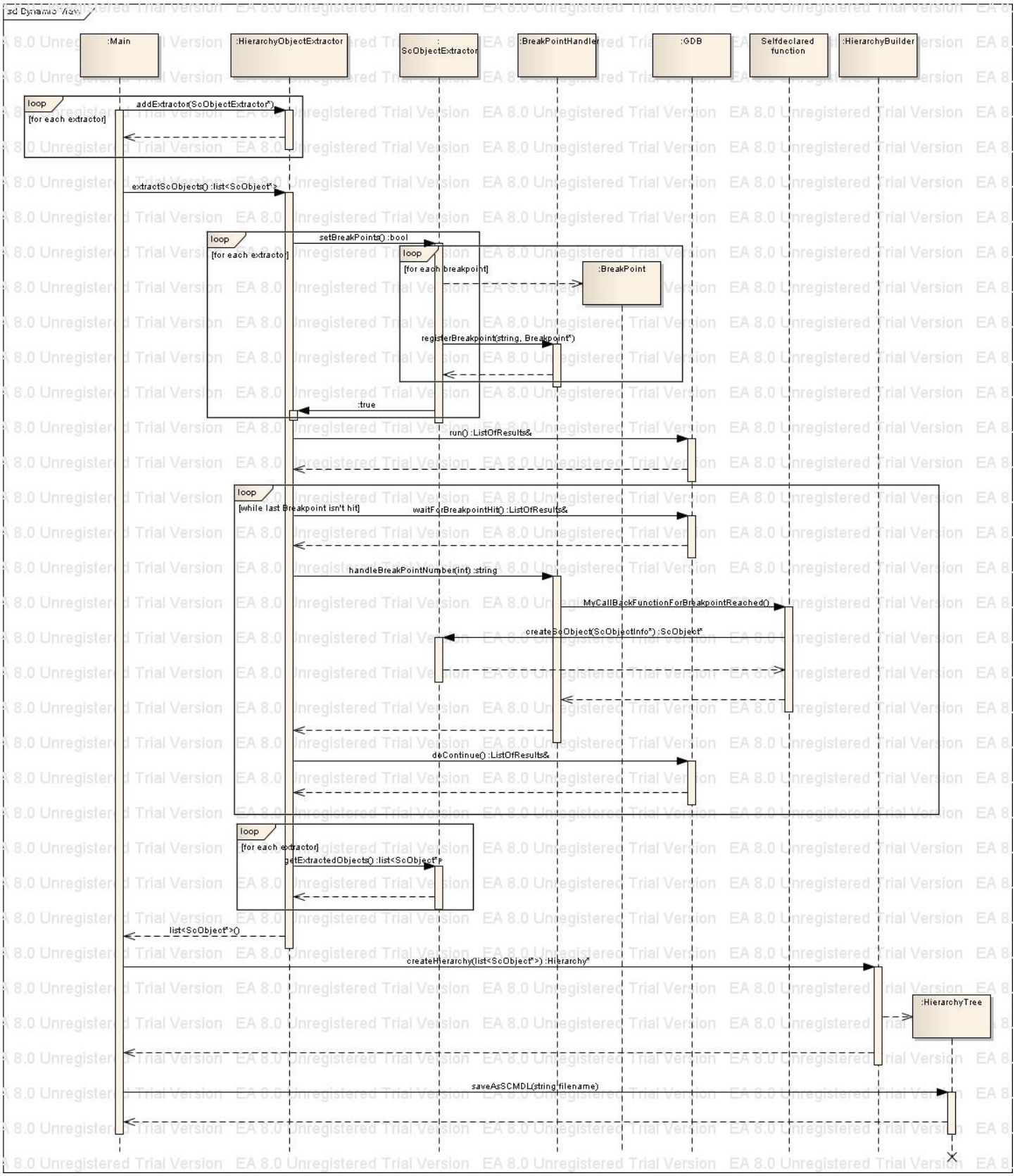
22.4. Bijlage C



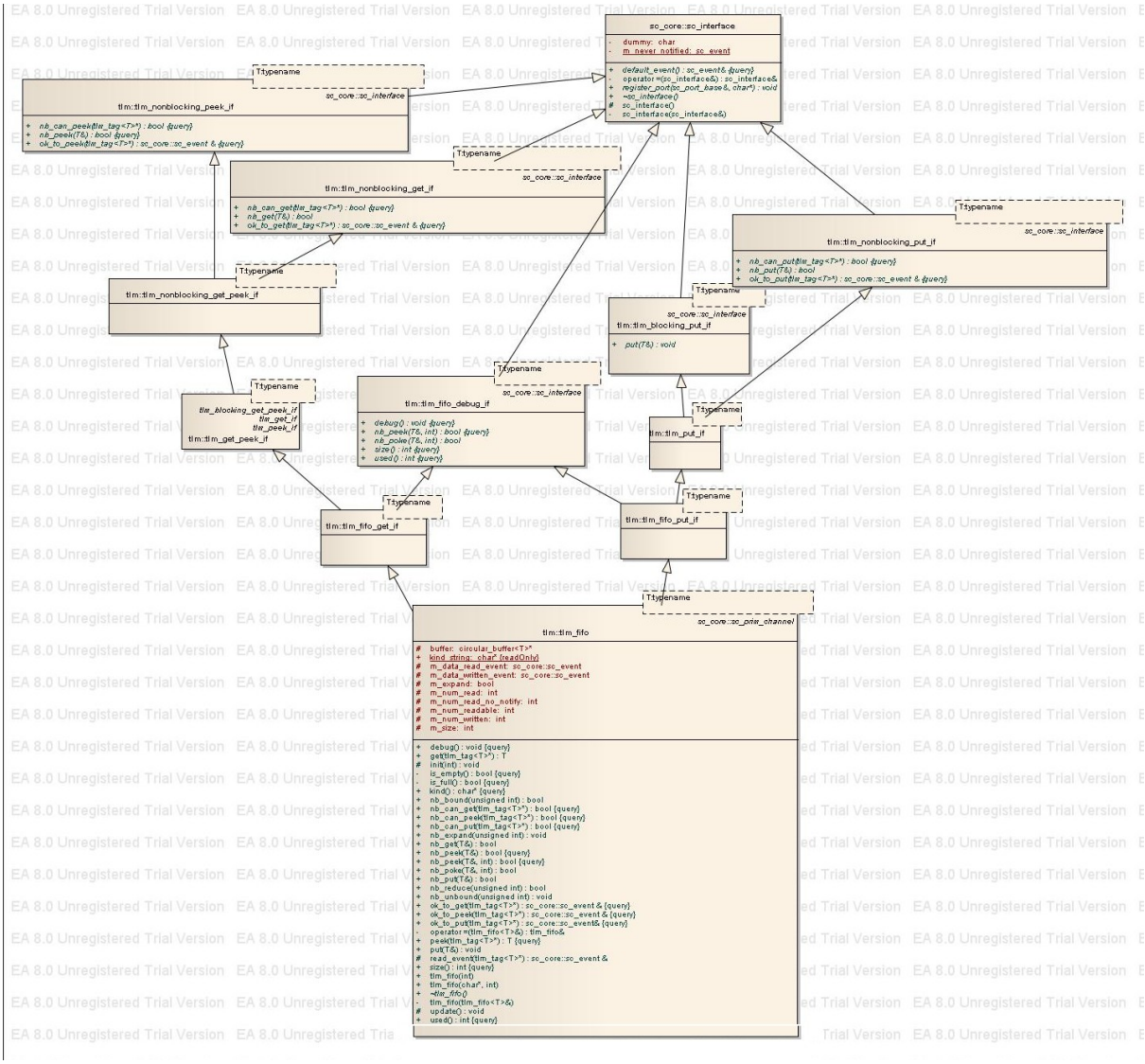
22.5. Bijlage D



22.6. Bijlage E



Bijlage F


sc_core::sc_interface
dummy: char
m_never_notified: sc_event
default_event() : sc_event& {query}
operator=(sc_interface&) : sc_interface&
register_port(sc_port_base&, char*) : void
~sc_interface()
sc_interface()
sc_interface(sc_interface&)
T typename
sc_core::sc_interface
tlm::tlm_nonblocking_peek_if
nb_can_peek(tlm_tag<T>*) : bool {query}
nb_peek(T&) : bool {query}
ok_to_peek(tlm_tag<T>*) : sc_core::sc_event & {query}
tlm::tlm_nonblocking_get_if
nb_can_get(tlm_tag<T>*) : bool {query}
nb_get(T&) : bool
ok_to_get(tlm_tag<T>*) : sc_core::sc_event & {query}
tlm::tlm_nonblocking_put_if
nb_can_put(tlm_tag<T>*) : bool {query}
nb_put(T&) : bool
ok_to_put(tlm_tag<T>*) : sc_core::sc_event & {query}
tlm::tlm_nonblocking_get_peek_if
tlm::tlm_blocking_put_if
put(T&) : void
tlm_blocking_get_peek_if
tlm_get_if
tlm_peek_if
tlm::tlm_get_peek_if
tlm::tlm_fifo_debug_if
debug() : void {query}
nb_peek(T&, int) : bool {query}
nb_poke(T&, int) : bool
size() : int {query}
used() : int {query}
tlm::tlm_put_if
tlm::tlm_fifo_get_if
tlm::tlm_fifo_put_if
sc_core::sc_prim_channel
tlm::tlm_fifo
buffer: circular_buffer<T>*
kind_string: char* {readOnly}
m_data_read_event: sc_core::sc_event
m_data_written_event: sc_core::sc_event
m_expand: bool
m_num_read: int
m_num_read_no_notify: int
m_num_readable: int
m_num_written: int
m_size: int
debug() : void {query}
get(tlm_tag<T>*) : T
init(int) : void
is_empty() : bool {query}
is_full() : bool {query}
kind() : char* {query}
nb_bound(unsigned int) : bool
nb_can_get(tlm_tag<T>*) : bool {query}
nb_can_peek(tlm_tag<T>*) : bool {query}
nb_can_put(tlm_tag<T>*) : bool {query}
nb_expand(unsigned int) : void
nb_get(T&) : bool
nb_peek(T&) : bool {query}
nb_peek(T&, int) : bool {query}
nb_poke(T&, int) : bool
nb_put(T&) : bool
nb_reduce(unsigned int) : bool
nb_unbound(unsigned int) : void
ok_to_get(tlm_tag<T>*) : sc_core::sc_event & {query}
ok_to_peek(tlm_tag<T>*) : sc_core::sc_event & {query}
ok_to_put(tlm_tag<T>*) : sc_core::sc_event& {query}
operator=(tlm_fifo<T>&) : tlm_fifo&
peek(tlm_tag<T>*) : T {query}
put(T&) : void
read_event(tlm_tag<T>*) : sc_core::sc_event &
size() : int {query}
tlm_fifo(int)
tlm_fifo(char*, int)
~tlm_fifo()
tlm_fifo(tlm_fifo<T>&)
update() : void
used() : int {query}

Bijlage G

```
<module systemc-name="TopNAME.reqRespChannelNAME_0" address="0x80c64d8">
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_0" address="0x80c6534">
        <bound-to to="primitive-channel" systemc-name="TopNAME.reqRespChannelNAME_0.fifo_0" />
        <bound-to to="primitive-channel" systemc-name="TopNAME.reqRespChannelNAME_0.fifo_1" />
    </export>
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_1" address="0x80c654c" />
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_2" address="0x80c6564">
        <bound-to to="primitive-channel" systemc-name="TopNAME.reqRespChannelNAME_0.fifo_0" />
        <bound-to to="primitive-channel" systemc-name="TopNAME.reqRespChannelNAME_0.fifo_1" />
    </export>
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_3" address="0x80c657c" />
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_4" address="0x80c6594" />
    <export systemc-name="TopNAME.reqRespChannelNAME_0.export_5" address="0x80c65ac" />
    <primitive-channel systemc-name="TopNAME.reqRespChannelNAME_0.fifo_0"
address="0x80c65c4" />
    <primitive-channel systemc-name="TopNAME.reqRespChannelNAME_0.fifo_1"
address="0x80c669c" />
</module>
```