KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

Functioneel Reactief Programmeren op Ingebedde Systemen

Functional Reactive Programming on Embedded Devices

Ben Calus

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotoren:

Prof. dr. Ir. F. Piessens
Dr. D. Devriese

Assessoren:

Prof. dr. T. Holvoet
Dr. J.T. Mühlberg

Begeleider:

B. Reynders

Academiejaar 2015 – 2016

Voorwoord

In dit dankwoord wil ik me richten tot alle mensen die hebben bijgedragen tot het afwerken van deze masterproef.

Als eerste wil ik alle mensen van de DistriNet onderzoeksgroep bedanken die me hebben geholpen bij dit werk. Daarbij gaat een extra dankwoord uit naar Bob en Dominique die mij steeds bijstonden met goede raad en nuttige feedback.

Ik wil ook mijn vrienden en vriendin bedanken voor de morele ondersteuning. Onder hen wil ik Stig en Astrid extra bedanken om dit werk na te lezen.

Als laatste wil ik mijn ouders, Marc en Linda, bedanken voor alle goede zorgen en kansen die zij mij hebben gegeven. De hoeveelheid steun dat ik van jullie krijg is moeilijk in woorden uit te drukken.

Ben Calus

Inhoudsopgave

Samenvatting

Event-gedreven applicaties zijn vaak het slachtoffer van moeilijk op te sporen fouten door expliciete wijziging van toestand. Net omdat er vaak veel verschillende events het systeem bereiken kan het vooral voor grote applicaties verrassend moeilijk zijn de toestand steeds op een consistente manier aan te passen. Net zoals grafische gebruikersinterfaces, zijn ingebedde systemen ook typisch event-gedreven. Door de opkomst van Internet-of-Things applicaties, winnen ingebedde systemen qua relevantie heel wat terrein binnen het domein van het event-gedreven programmeren.

Functioneel Reactief Programmeren (FRP) is een alternatief dat gebruikt kan worden om event-gedreven systemen eenvoudiger te programmeren. FRP introduceert een declaratieve manier om event-gedreven applicaties te ontwikkelen waarbij de programmeur verlicht wordt van de taak om de toestand van het systeem manueel aan te passen.

We presenteren in dit werk een *Embedded Domain Specific Language* (EDSL) die een collectie FRP-primitieven toevoegt aan de hosttaal. Deze maken het mogelijk een FRP-programma op te stellen dat uiteindelijk uitgevoerd kan worden op een ingebed systeem. Omdat ingebedde systemen vaak slechts beschikken over beperkte hardwaremiddelen leggen we de focus op een efficiënte uitvoering van het FRP-programma.

We slagen hierin door met behulp van het Lightweight Modular Staging (LMS) framework een codegenerator te bouwen. Deze maakt het mogelijk om een FRP-programma, opgesteld in de EDSL-taal, om te vormen naar C-code. De gegenereerde code kan vervolgens zonder aanpassingen gecompileerd worden voor het ingebed systeem.

Door de flexibiliteit van deze codegenerator slagen we er in om gespecialiseerde C-code te genereren die geschikt is voor het Sancus framework. Sancus is een beveiligd module systeem dat het mogelijk maakt modulaire applicaties op een veilige manier uit te voeren op een ingebed systeem zonder dat de volledige softwarestack daarvan vertrouwd moet worden. Door specifieke ondersteuning in te bouwen voor dit platform kunnen we op twee terreinen een meerwaarde bieden. Het wordt mogelijk eerste-orde FRP-abstracties te gebruiken om applicaties voor Sancus te ontwikkelen. Daarnaast biedt de EDSL ook de mogelijkheid verschillende softwaremodules binnen Sancus op een type-veilige manier aan elkaar te koppelen.

Kernwoorden Functioneel Reactief Programmeren (FRP), ingebedde systemen, Lightweight Modular Staging, Sancus

Hoofdstuk 1

Inleiding

Event-gedreven applicaties zijn niet meer weg te denken uit de brede waaier van verschillende soorten computerprogramma's. Dagelijks komen we in contact met event-gedreven systemen. Denk daarbij maar aan een smartphone, laptop of webserver. Wat de applicaties op dit soort systemen gemeen hebben is dat het uitvoeringspad van het programma bepaald wordt door gebeurtenissen of events in het systeem. Deze events kunnen door een gebruiker gegenereerd worden via bijvoorbeeld een muisklik, het aanraken van een touchscreen of een HTTP-request. Op het reageren op events na, doet het systeem in principe niets. Het systeem wacht tot er een nieuw event ontvangen wordt om daarop dan met een bepaalde actie te reageren. Daarom noemt men deze event-gedreven systemen ook wel reactieve systemen. Naast grafische gebruikersinterfaces of GUI's, zijn Internet of Things (IoT) applicaties (of in het algemeen applicaties voor ingebedde systemen) van nature event-gedreven. In de plaats van events, gegenereerd door een gebruiker via een GUI, kunnen events in een ingebed systeem van verschillende afkomst zijn. Zo kunnen sensoren, drukknoppen maar ook interne timers ervoor zorgen dat het systeem actie onderneemt. Een typische IoT-applicatie bestaat uit meerdere ingebedde systemen of nodes die data uitwisselen in de vorm van events.

Hoewel event-gedreven applicaties al heel erg lang ontwikkeld worden is de manier waarop deze programma's geschreven worden nog steeds imperatief. Dit wil zeggen dat het programma op een zeer expliciete manier opgebouwd moet worden met een gebrek aan hogere abstracties. Ze zijn daarom vaak complex door de grote hoeveelheid conditionele logica. Het is doorgaans moeilijker om over event-gedreven applicaties te redeneren dan over een procedureel programma waarbij het uitvoeringspad vastligt. Dit zijn enkele redenen waarom er vaak extra fouten in event-gedreven applicaties sluipen die moeilijk op te sporen zijn. We merken dit nog steeds frequent in applicaties die vast lopen, niet doen wat er bedoeld wordt of zelfs afgesloten worden na verloop van tijd. Functioneel Reactief Programmeren of FRP is een programmeerparadigma dat het schrijven van event-gedreven applicaties eenvoudiger en daardoor ook minder gevoelig voor fouten probeert te maken. Het tracht dit te verwezenlijken door enkele nieuwe programmeerconstructies te introduceren die het

mogelijk maken een event-gedreven programma op een declaratieve manier samen te stellen. De toevoeging van deze abstracties voor event-gedreven applicaties laat toe om op een hoger niveau over dit soort programma's te redeneren. Frameworks die FRP-concepten aanbieden moeten intern typisch extra datastructuren bijhouden tijdens het uitvoeren van het programma. Dit zorgt ervoor dat FRP-applicaties vaak meer tijd en ruimte vereisen dan dezelfde applicatie ontworpen via de traditionele imperatieve stijl. Hoewel het schrijven van FRP-programma's eenvoudiger moet zijn aan de hand van FRP, betalen we daarvoor de prijs met extra overhead. Vooral voor applicaties op ingebedde systemen is dit een belangrijk aspect dat in rekening gebracht moet worden. Deze zijn vaak gelimiteerd in hoeveelheid geheugen en processorkracht waardoor efficiëntie belangrijker is dan bij een gemiddelde GUI applicatie. Nieuwe FRP-implementaties trachten deze overhead te beperken maar dit gaat vaak ten koste van de volledigheid van de FRP API.

Het doel van deze masterproef bestaat erin om ten eerste na te gaan welke eigenschappen van FRP kunnen zorgen voor een efficiëntieverlies. Daarbij aansluitend presenteren we een Embedded Domain Specific Language (EDSL) die FRP-concepten aanbiedt voor ingebedde systemen. Naast het generieke concept voor ingebedde systemen zullen we ons voor een groot gedeelte focussen op Sancus, een framework voor ingebedde systemen met een beveiligd module architectuur. We trachten daarbij de overhead van de FRP-programma's zo klein mogelijk te houden. We formuleren bij deze doelstellingen de volgende onderzoeksvraag:

Wat is er nodig om een efficiënte FRP API te implementeren voor ingebedde systemen en wat zijn de consequenties daarvan?

In het vervolg van de tekst worden eerst de verschillende programmeerparadigma's zoals event-gedreven en reactief programmeren verder uitgediept met extra aandacht voor functioneel reactief programmeren. We bespreken daarbij enkele bestaande implementaties die een belangrijke invloed hebben gehad op FRP. In een volgende sectie gaan we dieper in op de technieken die in aanmerking komen om een efficiënte FRP-implementatie te ontwikkelen, meer bepaald het Lightweight Modular Staging framework. In de laatste sectie van het eerste hoofdstuk geven we iets meer achtergrond over Sancus, het framework met een beveiligd module architectuur. Het derde hoofdstuk toont aan hoe de EDSL eruit ziet van de gebruikerskant. Dat wil zeggen dat we de FRP API voorstellen zoals we deze aanbieden in de ontwikkelde EDSL aan de hand van een voorbeeld. Daarna gaan we in het vierde hoofdstuk in detail in op de basisimplementatie van de EDSL die het mogelijk maakt om FRP-programma's op te bouwen. Pas in het vijfde hoofdstukken brengen we aan wat er zoal nodig is om deze basis concreet toepasbaar te maken voor het Sancus framework. Aan de hand van de ontwikkelde EDSL kunnen we in hoofdstuk zes onze implementatie evalueren. We doen dit op basis van ons eigen werk in vergelijking met een aantal frameworks met gelijkaardige doelstellingen. In het zevende en laatste hoofdstuk zullen we ten slotte via deze bevindingen een besluit formuleren op de onderzoeksvraag en het werk concluderen.

Hoofdstuk 2

Achtergrond

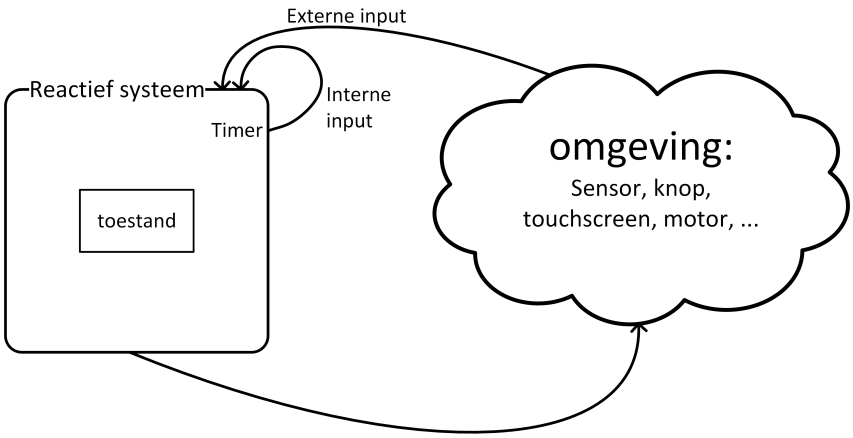
We komen dagelijks in aanraking met reactieve systemen. In dit hoofdstuk gaan we eerst in op event-gedreven programmeren in het algemeen. Daarin wordt besproken hoe event-gebaseerde programma's traditioneel worden opgebouwd en hoe dit kan leiden tot problemen. Daarna gaan we dieper in op functioneel reactief programmeren in het domein van event-gedreven applicaties. Vervolgens komen een aantal werken aan bod die belangrijk zijn geweest bij het sturen van de ontwikkeling van FRP. We bespreken enkele verschillende interpretaties en de abstracties die aangeboden worden om een aantal problemen van event-gedreven applicaties tegen te gaan. Zoals al aangehaald in de inleiding zullen we in dit werk een EDSL presenteren waarin deze FRP-concepten aangeboden worden. In het tweede deel van dit hoofdstuk wordt toegelicht hoe Lightweight Modular Staging (LMS) framework ons in staat stelt om zo'n EDSL te bouwen. Een laatste deel van dit hoofdstuk schetst in het kort een achtergrond voor Sancus, een beveiligde architectuur voor netwerkgeconnecteerde ingebedde systemen.

2.1 Event-gedreven programmeren

Event-gedreven programmeren is een traditioneel paradigma voor het programmeren van reactieve systemen. Een typisch reactief systeem is weergegeven in figuur 2.1. We beginnen met een definitie van event-gedreven applicaties en hoe events in een reactief systeem worden afgehandeld. Daarna gaan we wat dieper in op de oorzaak van de moeilijkheden bij event-gedreven programma.

2.1.1 Definitie

Event-gedreven applicaties zijn programma's waarbij het uitvoeringspad (sequentie van uitgevoerde instructies) van het programma bepaald wordt door events. Deze events kunnen afkomstig zijn van acties van onder andere een gebruiker, sensordata of een HTTP-request. Dit soort systemen wacht en doet niets tot er een event optreedt. Het systeem kan al dan niet een toestand bezitten. Indien het systeem geen toestand heeft (*stateless*) zal de reactie van het systeem op het event enkel van



Figuur 2.1: Reactief systeem

dit event afhangen. Het komt echter vaker voor dat het systeem wel een toestand heeft (*stateful*). In dit tweede geval hangt de actie niet alleen af van het opgetreden event maar ook van de huidige staat waarin het systeem verkeert. Eventueel wordt na de actie ook de toestand van het systeem aangepast.

2.1.2 Eventafhandeling in 3 fasen

Het afhandelen van inkomende events gebeurt in 3 fasen. In de eerste fase wordt het event opgevangen en wordt het type van het event bepaald. De tweede fase noemen we dispatching en deze hanteert een mapping van het event type naar een bepaalde eventhandler. Via deze mapping kan de juiste eventhandler aangeroepen worden. In de derde fase wordt de eventhandler (of callback functie) uitgevoerd waardoor de gewenste actie wordt uitgevoerd. De meeste frameworks die event-gedreven programmeren toelaten verwachten alleen dat de programmeur de nodige eventhandlers voorziet voor de verschillende events die kunnen optreden. De eventhandler wordt gekoppeld aan de eventgenerator. De twee eerste fasen worden automatisch afgehandeld en ook de mainloop waarin het programma wacht op events is vaak reeds voorzien.

2.1.3 Moeilijkheden

Hoewel event-gedreven programma's al heel erg lang ontwikkeld worden, blijven ze erg kwetsbaar voor fouten en zijn ze typisch moeilijk te onderhouden. Dit is enerzijds te wijten aan het feit dat het uitvoeringspad van het programma niet vastligt. Dit maakt redeneren over het programma moeilijker. Omdat het uitvoeringspad niet vastligt, is ook het testen van deze programma's moeilijk. Het opsporen van fouten wordt bemoeilijkt door de enorme hoeveelheid verschillende uitvoeringspaden door de combinaties van events die zich kunnen voordoen. Heel wat projecten [13, 15, 16] trachten technieken aan te bieden om event-gedreven systemen te kunnen testen.

Anderzijds bestaat de code vaak uit grote hoeveelheden van conditionele logica om bij de combinatie van een soort event en de huidige toestand de correcte actie uit te voeren. In een complex programma met vele toestandsvariabelen kan dit al snel leiden tot inconsistenties bij het aanpassen van de toestand. De relatie tussen een bepaald event en de ondernomen actie is meestal wel duidelijk, maar de relatie tussen de huidige toestand van het systeem en de ondernomen actie is vaak troebel. Het is ook duidelijk dat deze aanpak niet goed schaalt naar grotere projecten waarbij de context of toestand van het programma steeds groter wordt. Dit soort fouten (*bugs*) zijn erg moeilijk op te sporen en zorgen voor onverwacht gedrag van de applicatie. Om u hiervan een idee te geven, geeft men in [12, p.24] een concreet voorbeeld van deze moeilijkheden. In de code van Adobe's desktopapplicaties is maar liefst 1/3 van alle code toegewijd aan logica om events af te handelen. Men heeft verder vastgesteld dat 1/2 van de bugs die gerapporteerd worden betrekking hebben tot deze code.

Het probleem met de traditionele aanpak (eventhandler gebaseerd) om applicaties te ontwikkelen voor reactieve systemen ligt in principe ook bij de talen waarin deze ontworpen worden. Het reactieve systeem kan voorgesteld worden als een sequentie van toestanden waarin het verkeert. Eender welk event kan ervoor zorgen dat het systeem in een nieuwe toestand terecht komt. Het probleem met traditionele general-purpose imperatieve talen is dat er geen ondersteuning is voor deze constante wijziging van de toestand. Dit wil zeggen dat de event handlers (en daarbij dus de programmeurs) verantwoordelijk zijn om de toestand steeds consistent aan te passen. Wanneer de toestand slechts partieel wordt aangepast zal dit leiden tot problemen.

Wat nodig is om dit probleem aan te pakken, is een programmeertaal/paradigma waarin het constant wijzigen van de toestand van het systeem direct in de taal ondersteund wordt en de programmeur van deze taak wordt verlicht. Enkel door de verantwoordelijkheid voor consistente toestandswijziging te verleggen van de programmeur naar een onderliggend framework, kan het probleem structureel aangepakt worden. Dat is het moment waarop functioneel reactief programmeren in beeld komt.

2.2 Functioneel reactief programmeren

FRP is een programmeerparadigma dat specifieke constructies aanbiedt om impliciet met een constant wijzigende toestand om te gaan. De actie uitgevoerd door een reactief systeem is niet alleen afhankelijk van het huidige opgetreden event, maar ook van de events die reeds hebben plaatsgevonden. Deze geschiedenis van events is slechts een andere verwoording voor de toestand van het systeem. De toestand wordt namelijk gewijzigd bij elk event dat optreedt en op die manier zijn verschillende gebeurtenissen verantwoordelijk voor de toestand waarin het systeem zich momenteel bevindt. In plaats van de ad-hoc technieken die in klassieke event-gedreven paradigma's gehanteerd worden om events af te handelen, biedt FRP een meer structurele oplossing. Om het probleem fundamenteel aan te pakken haalt het zijn inspiratie uit puur functionele talen. Daarbij is er geen probleem met wijzigende

toestand omdat alle waarden constant zijn. Eens een waarde gedefinieerd is kan deze niet meer veranderen. Dit heeft als gevolg dat functies steeds hetzelfde resultaat geven waardoor het redeneren over deze programma's eenvoudiger wordt. Omdat de systemen die we bekijken nu eenmaal gekenmerkt worden door een wijzigende toestand doorheen de tijd, schiet een pure functionele taal tekort om dit uit te drukken. Er is in een reactief systeem nu eenmaal nood aan waardes die kunnen veranderen. Functioneel reactief programmeren tracht dit op te lossen met een model waarin het zelf de nodige constructies aanbiedt om een wijzigde toestand te modelleren en terwijl toch nog te kunnen profiteren van de voordelen van functioneel programmeren. De kracht van FRP ligt in het uitdrukkingsvermogen om waardes die doorheen de tijd wijzigen te kunnen definiëren en deze vervolgens als primitieve waarden te kunnen behandelen. Op die manier kunnen we nog steeds een functionele manier van programmeren hanteren terwijl we toch een systeem kunnen modelleren dat zijn toestand zal wijzigen als reactie op een event.

FRP is echter niet bedacht in de context van reactieve systemen en functioneel reactief programmeren bestaat dan ook in verschillende interpretaties. We vervolgen deze sectie met de groei die functioneel reactief programmeren heeft doorgemaakt om te eindigen bij een model, event-gedreven FRP, dat het beste aansluit bij reactieve systemen. Om de lezer daarna iets meer voeling te geven met deze nieuwe concepten geven we een conceptuele voorstelling van FRP met een voorbeeld.

2.2.1 Definities en geschiedenis

Functioneel reactief programmeren is een declaratief programmeermodel om reactieve systemen te programmeren. Het heeft een specifieke semantiek om met asynchrone data om te kunnen gaan. Declaratief betekent dat de aandacht uitgaat naar 'wat' er voorgesteld moet worden door het programma. Dit staat gekant tegen het meer traditioneel imperatief programmeren. Dit vereist meer 'hoe' een applicatie opgebouwd wordt. We zagen net al dat FRP sterk geïnspireerd werd door concepten uit functionele talen, waarbij de semantiek aangepast werd naar het concept van asynchrone datastromen. Op deze manier biedt FRP ingebouwde ondersteuning voor asynchrone datastructuren, wat een noodzakelijke abstractie is bij event-gedreven programmeren zoals voordien aangegeven. In de eerste formulering van FRP was de focus echter anders. Om dit in kaart te brengen overlopen we een aantal verschillende interpretaties van FRP en hoe deze uiteindelijk uitmonden in een voorstelling voor event-gedreven programma's. Aan het einde van deze sectie gaan we dan ook vooral dieper in op deze semantiek omdat we deze zullen hanteren in dit werk, namelijk event-gedreven FRP.

Traditionele FRP en RT-FRP

We beginnen bij het begin en dat is het originele onderzoek rond FRP door Elliott en Hudak in hun werk Functional Reactive Animation [6]. In dit onderzoek rond animaties onderzoeken ze de sterke (maar ongewenste) correlatie tussen *wat*

er gemodelleerd wordt en *hoe* dit daarna gepresenteerd wordt. Om daarin bij te springen worden twee nieuwe datatypes gedefinieerd, *Behaviors* en *Events*. Deze worden daarin geïntroduceerd om animaties op een natuurlijkere manier te kunnen beschrijven. Het idee was dat behaviors gebruikt worden als waarden die veranderen doorheen de tijd. Denk daarbij bijvoorbeeld aan een afbeelding die kan wijzigingen doorheen de tijd, of anders verwoord een animatie. Dit is in essentie hoe dit idee ontstaan is. Events daarentegen zijn een stroom van discrete gebeurtenissen die bijvoorbeeld muisklikken kunnen voorstellen en mogelijk een invloed hebben op de behaviors. Deze twee nieuwe concepten zien er iets formeler uit als volgt:

Behavior : continu gedefinieerde waarde die wijzigt doorheen de tijd. We kunnen deze voorstellen als een functie van de tijd naar een concrete waarde van een bepaald type *a*.

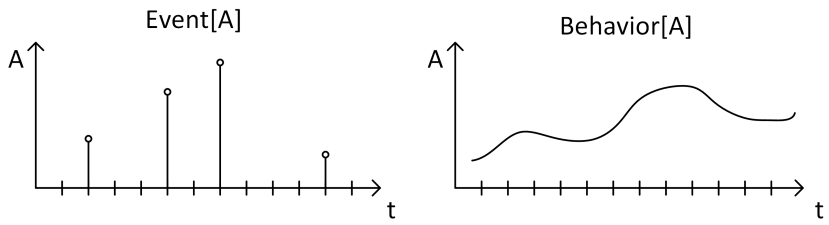
$$\text{Behavior a} = \text{Time} \longrightarrow \text{a}$$

Het verschil met een traditionele variabele is dat een behavior een concrete waarde op een concreet tijdstip voorstelt. Dit wil zeggen dat het resultaat van de evaluatie van deze variabele afhangt van het tijdstip van evaluatie.

Event : sequentie van discrete gebeurtenissen (*events*), waarbij elk gebeurtenis een concrete waarde draagt. We noemen dit daarom ook vaak een eventstroom, wat een beter beeld geeft over de sequentie van gebeurtenissen. We kunnen deze voorstellen als een lijst van paren.

$$\text{Event a} = \text{Lijst}[(\text{Time, a})]$$

In dit originele onderzoek is tijd een erg belangrijke factor. Zowel events als behaviors zijn sterk verbonden met het concept van tijd en in de DSL die men in dit werk ontwikkelde wordt tijd zeer expliciet gemodelleerd. Een rechtstreeks gevolg daarvan is dat behaviors worden voorgesteld als *continu* wijzigende variabelen, net zoals de tijd. Dat wil zeggen dat op elke moment deze data wijzigt zoals we in figuur 2.2 zien in een grafische voorstelling van events en behaviors. Deze keuze bleek achteraf de grootste bron van inefficiëntie. Zelfs wanneer de waarde van een behavior hetzelfde blijft, veronderstelt men in de originele formulering van FRP dat deze waarde continu wijzigt. Dit kan in de implementatie zorgen voor onnodige herberekeningen van deze behaviorwaarden. Door de keuze van een *lazy* evaluerende hosttaal, Haskell in dit geval, kan dit leiden tot excessief ruimte- en tijdsgebruik dat moeilijk te traceren is. Wanneer een behavior dat al gedurende enige tijd niet meer geïnspecteerd is, opnieuw geëvalueerd wordt, kan dit in Haskell omwille van luie evaluatie het uitvoeren van een reeks opgestapelde berekeningen vereisen. Deze reeks berekeningen die moeten uitgevoerd worden om de behaviorwaarde opnieuw up-to-date te brengen zorgt enerzijds voor een groot ruimtegebruik. Anderzijds introduceert dit ook een grote tijdsvertraging in het systeem om ze daadwerkelijk te voltooien. Zelfs wanneer een implementatie ontwikkeld wordt waarin dit probleem is aangepakt, laat het nog steeds toe om in de gedefinieerde FRP-programma's excessief ruimte- en tijdsgebruik



Figuur 2.2: Event vs. continu wijzigend Behavior

te introduceren. Om deze reden is het erg moeilijk om voor deze criteria garanties te kunnen bieden voor de uitvoering van de FRP-applicaties. Deze garanties vielen echter buiten beschouwing in de originele formulering en voor heel wat applicaties die profiteren van het FRP-framework was dit ook geen probleem. Pas wanneer men zich wou richten op *real-time* applicaties was er nood aan formele garanties omtrent uitvoeringskost.

Omdat in de originele formulering van FRP de uitvoeringskost buiten beschouwing wordt gelaten, was het een logische volgende stap om dit in rekening te brengen voor *real-time* applicaties. In een volgend onderzoeksproject van Hudak et al. wordt Real-time FRP [17] voorgesteld waarin men zich op meer *resource*gelimiteerde applicaties richt. Daarin staat centraal om de uitvoeringskost van het FRP-programma te kunnen bepalen nog voor het is uitgevoerd en liefst van al wil men ook dat deze kost constant gehouden kan worden. Een belangrijke vaststelling in dit werk is het polymorfisme tussen Events en Behaviors uit de originele formulering:

$$\text{Event a} \approx \text{Behavior (Option a)}$$

Dit betekent concreet dat een event voorgesteld kan worden als een behavior dat ofwel een waarde a bevat (Some a) ofwel geen waarde draagt (None). In dit laatste geval betekent dit dat er geen event optreedt. Zowel Events als Behaviors worden zo geünificeerd in een nieuw datatype waarbij men nog steeds uitgaat van continu wijzigende waarden:

$$\text{Signal a} = \text{Time} \rightarrow \text{a}$$

Mede dankzij dit eenvoudigere model slaagt men erin de uitvoeringskost vast te leggen door slechts een subset van FRP aan te bieden. Om dit te verwezenlijken splitst men het framework op in een ongelimiteerde basistaal en een gelimiteerde reactieve taal waarin signalen slechts op een veilige manier gemanipuleerd kunnen worden. Door aansluitend de DSL op een efficiënte manier te implementeren, kan men de nodige garanties bieden wat niet mogelijk was bij met de ongelimiteerde reactieve taal uit FRP. Het is dus duidelijk dat het garanderen van efficiëntie ten koste gaat van de expressiviteit voor signalen. De reactieve taal kan niet langer beschikken over hogere-orde abstracties en kan dus niet langer profiteren van volledige lambda-calculus en recursie.

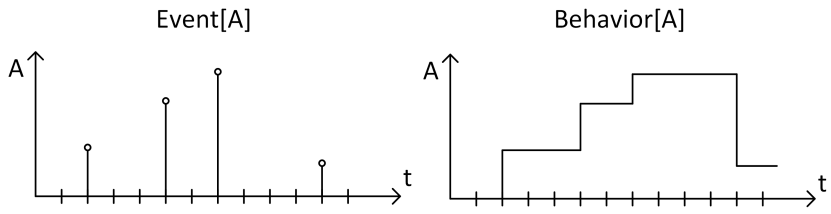
Verder onderzoek vloeide voort uit de vaststelling dat heel wat systemen extra interessante eigenschappen hebben die een rechtstreekse invloed hebben op FRP. Heel wat domeinen zijn niet alleen resource gelimiteerd, maar hebben de eigenschap event-gedreven te zijn. Ook andere systemen die niet noodzakelijk gelimiteerd zijn qua middelen, zijn event-gedreven. Omdat FRP en RT-FRP deze eigenschap niet in beschouwing neemt, volgde daaruit ook een nieuw onderzoeksproject, namelijk event-gedreven FRP.

Event-gedreven FRP

Event-gedreven FRP of E-FRP [18] is een veralgemening die ontstaan is uit RT-FRP waarin een signaal als discreet wordt beschouwd. Dit wil zeggen dat signalen niet langer continu wijzigen maar slechts wijzigen als reactie op events die optreden in het systeem. Dit stemt natuurlijker overeen met event-gedreven programma's aangezien er geen wijziging moet doorgevoerd worden in het systeem behalve wanneer er een event optreedt. Net door deze vaststelling moet in dit framework de notie van tijd niet meer expliciet gemodelleerd worden. Verandering van toestand wordt enkel en alleen gedreven door events. Elke eventstroom kan zorgen voor een *tick* waarop het systeem naar een nieuwe toestand gaat. Een groot aantal domeinen vallen in dit puur event-gedreven idee. E-FRP werd ontwikkeld om robots te besturen, welke typisch reageren op events van de buitenwereld. Naast ingebedde systemen die ook gelimiteerd zijn qua resources, zijn ook andere domeinen zoals GUI's typisch event-gedreven. Twee zeer interessante projecten omtrent GUI's en webapplicaties zijn de Elm taal [3, 5] en flapjax [9]. Een conclusie in het werk van Hudak et al. die naar voren wordt gebracht is dat deze aanpak zeer geschikt is voor het programmeren van interrupt-gedreven micro-controllers. Applicaties voor deze platformen worden klassiek in een imperatieve programmeertaal geprogrammeerd.

E-FRP gebruikt opnieuw Behaviors en Events als concepten om asynchrone datastromen en de toestand van het reactieve systeem voor te kunnen stellen. Figuur 2.3 stelt het verschil voor tussen een event en een behavior, met als belangrijk verschil dat behaviors nog slechts op discrete momenten wijzigen. Dit zorgt ervoor dat overbodige herberekeningen niet meer aan de orde zijn zoals het geval was bij continu gedefinieerde behaviors. Wanneer we de samenwerking tussen events en behaviors wat beter bekijken, stellen we vast in de figuur dat de behaviorwaarde alleen nog wijzigt bij het optreden van events.

In [18] presenteren Hudak et al. een basis eerste-orde subset van FRP en geven aan hoe deze op een natuurlijke en correcte manier gecompileerd kan worden naar een imperatief programma. Net omdat ingebedde systemen erg beperkt zijn qua middelen is het vaak geen mogelijkheid een omvangrijk framework op het systeem uit te voeren dat toegang biedt tot programmeertalen met hogere abstracties. Daarom kiest men in dit werk voor het compileren van het FRP-programma naar een imperatief programma dat meteen bruikbaar is voor de micro-controller in kwestie. Dit betekent ook dat het belang van hogere programmeerabstracties minder evident wordt om



Figuur 2.3: Event vs. discreet wijzigend Behavior

aangeboden te worden in de reactieve taal omdat dit uiteindelijk omgezet moet worden naar een imperatieve taal. De vaststelling dat ze in hun project focussen op eerste-orde is daar een belangrijke indicatie voor en verdient een extra woordje uitleg.

2.2.2 Eerste-orde FRP vs. hogere-orde FRP

Zoals we hebben aangegeven zijn er heel wat varianten ontstaan met steeds een andere focus. Event-gedreven FRP is een variant van FRP die zich toespitst op systemen die event-gedreven zijn. Dit brengt met zich mee dat behaviors niet langer continu variërend zijn, maar discreet variërend. Evan Czaplicki, de maker van de Elm taal, geeft een zeer goed overzicht van de verschillende categoriëen waarin E-FRP opgedeeld kan worden in een recente presentatie [4]. De belangrijkste categorisatie daarin is het verschil tussen frameworks die een statische FRP-graaf aanbieden tegenover een dynamische graaf. We zijn nog niet ingegaan op deze graafvoorstelling die typisch aanwezig is bij een FRP-framework. In de implementatie van een FRP-framework wordt typisch een graaf opgesteld die de relaties tussen de verschillende events en behaviors vast legt. Als deze graaf statisch is, betekent dit dat het FRP-programma *at runtime* niets meer kan veranderen aan de onderlinge relaties tussen de verschillende datastromen. Zoals de term al aangeeft kan dit bij een dynamische graaf wel.

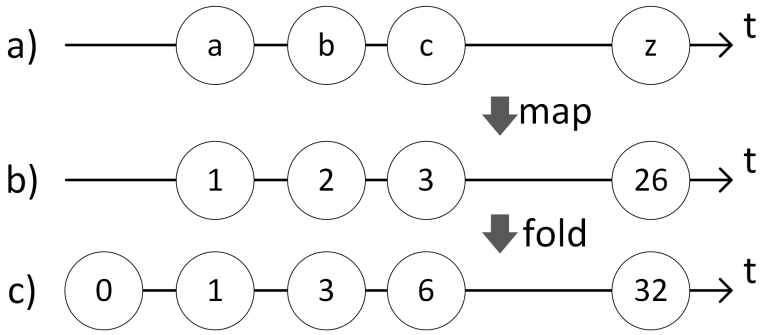
Men spreekt vaak ook van de orde van een FRP framework. Zo wordt eerste-orde FRP typisch gekarakteriseerd door een statische graaf en een synchrone eventafhandeling. Dit laatste betekent dat het reactieve systeem events steeds zal afhandelen in de volgorde waarin ze toekomen. Andere karakteristieken zijn dat signalen oneindig lang gedefinieerd zijn en steeds geconnecteerd zijn met de wereld. Hoewel wij het in dit werk zullen hebben over behaviors en events is dit gelijkaardig. Wat betreft de FRP-frameworks die een dynamische graaf aanbieden zijn er nog een aantal extra opdelingen zoals hogere-orde FRP of soms ook monadische FRP genoemd, arrowised FRP en asynchrone dataflow. We gaan hier niet verder in op deze varianten maar concreet zullen ze een aantal karakteristieken laten vallen om ze in te ruilen voor nieuwe karakteristieken als asynchrone eventafhandeling. Om de cirkel rond te maken kan de originele FRP-formulering gecategoriseerd worden als hogere-orde FRP. Het wordt gekenmerkt door een hoge expressiviteit en *equational reasoning*, maar geeft daar een bewijsbaar efficiënte implementatie voor op.

2.2.3 Conceptueel

Om nu vanuit deze geschiedenis van FRP de concepten wat te verduidelijken, maken we opnieuw de link met reactieve systemen en event-gedreven applicaties. Bij reactieve systemen krijgen we te maken met een stroom van events. Het is niet vooraf te bepalen wanneer deze events zullen optreden. Omdat een stroom van events zo fundamenteel is voor reactieve systemen, biedt functioneel reactief programmeren primitieve ondersteuning aan om deze voor te kunnen stellen. Met behulp van primitieve ondersteuning voor eventstromen wordt het mogelijk deze op een natuurlijke manier te behandelen. Conceptueel kan in FRP dus alles voorgesteld worden als een stroom van asynchrone gebeurtenissen. Met primitieve ondersteuning bedoelen we dat we niet alleen alles in de vorm van een stroom kunnen voorstellen, maar er vervolgens ook operaties op kunnen uitvoeren om deze stroom te manipuleren. De inspiratie voor heel wat FRP-operaties komt duidelijk uit functionele programmeertalen. Denk daarbij aan operaties als map-, filter- en fold-operaties. Het verschil daarbij is dat in functionele programmeertalen deze operaties gedefinieerd zijn op collecties van data, terwijl ze bij functioneel reactief programmeren gedefinieerd zijn op stromen van data doorheen de tijd.

Om deze concepten te verduidelijken, stellen we in figuur 2.4 een initiële eventstroom voor in stroom a en enkele operaties die deze omvormen naar twee nieuwe stromen b en c. We zien in de initiële stroom dat events op eender welk tijdstip kunnen optreden. In de eerste eventstroom zien we dat er doorheen de tijd events optreden die een karakterwaarde bevatten. Deze eventstroom kan bijvoorbeeld afkomstig zijn van een gebruiker die een tekst typt en zo events genereert via het toetsenbord. Een map-operatie laat toe om deze stroom van karakters om te vormen naar bijvoorbeeld een stroom van posities van de karakters in het alfabet. De fold-operatie kan aangewend worden om verschillende events doorheen de tijd op eender welke manier te accumuleren. In het voorbeeld worden de posities van de karakters bij elkaar opgeteld. FRP biedt een model waarin eender welke voorstelling van data omgevormd kan worden tot een stroom van events en nadien kan die dan weer omgevormd worden naar een stroom van een andere vorm.

Men heeft zich voor de verschillende operaties in FRP rechtstreeks laten inspireren door analoge functies uit functionele talen als bijvoorbeeld Haskell. We kunnen in figuur 2.4 de stroom interpreteren als een lijst van karakters in geheugen in plaats van in tijd zoals op de figuur. Als we daarop dezelfde *map*- en *fold*-operatie uitvoeren zal het resultaat gelijk zijn aan 32 ofwel de som van de de posities van de karakters in het alfabet. Wanneer we dit opnieuw vergelijken met de waarden zoals in de FRP-interpretatie, zien we dat aan het einde van de laatste eventstroom dit resultaat gelijk is. Merk daarbij op dat we dit kunnen zien als een lijst die zich doorheen de tijd opbouwt en de som van de posities van de karakters dus steeds aangepast wordt als er een nieuw karakter in de vorm van een event toekomt in het systeem. Deze onderste stroom in de figuur stelt dit concept voor en we zullen dit aan de hand van FRP op een eenvoudige manier kunnen voorstellen.



Figuur 2.4: Stromen van asynchrone events en de operaties die ze omvormen

2.2.4 Doel

Het doel van FRP is om een model aan te bieden waarin het eenvoudiger wordt om event-gedreven applicaties te ontwerpen. De onnodige, low-level details die voor het systeem nodig zijn om de applicatie uit te kunnen voeren worden afgeschermd van de programmeur. Aan de hand van de aangeboden abstracties, kan de effectieve applicatielogica op een hoger niveau beschreven worden en kan men meer focussen op de basis *business*logica van de applicatie.

2.3 EDSL en Lightweight Modular Staging

Zoals in de vorige sectie aan bod kwam, moeten we de nodige programmeerconstructies voorzien die het mogelijk maakt om functioneel reactief te kunnen programmeren. Het ontwikkelen van een volledig nieuwe programmeertaal is echter verre van triviaal en vraagt naast de focus op de FRP-primitieven ook extra inspanningen voor de ontwikkeling van de volledige compiler toolchain.

Daarom lichten we eerst toe hoe een Embedded Domain Specific Language (*EDSL*) ons kan helpen om hetzelfde doel te bereiken en toch te kunnen focussen op functioneel reactief programmeren. Daarna gaan we iets meer in detail in op Lightweight Modular Staging, een concreet framework dat ons toelaat een EDSL te bouwen.

2.3.1 Embedded Domain Specific Language

Om te beginnen is er een verschil tussen een *general-purpose* programmeertaal (GPL) en een domein specifieke programmeertaal (*DSL*). Terwijl de eerste inzetbaar is als taal voor verschillende domeinen, is een DSL een programmeertaal die zich specialiseert in een welbepaald domein. Een DSL heeft duidelijk een aantal voordelen maar zeker ook enkele nadelen. We sommen deze kort even op:

Voordelen

- specifieke features/abstracties voor het specifieke domein

- validatie van logica op domein niveau (en niet meer alleen op taalniveau)
- meer focus op domeinlogica dan op programmeerlogica

Nadelen

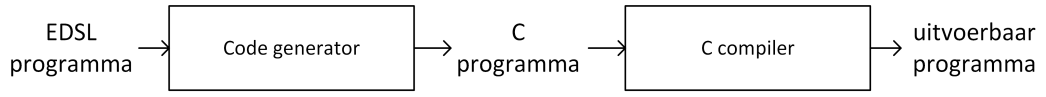
- hoge kost voor ontwikkeling en onderhoud van nieuwe programmeertaal
- hoge kost om voor elk domein een nieuwe taal te leren
- beperkte documentatie omwille van het beperkte domein
- mogelijk verlies van runtime efficiëntie in vergelijking met dezelfde code in GPL

Om een aantal van deze problemen op te lossen, kan men ook een domein specifieke taal integreren in een general-purpose programmeertaal. Men spreekt dan van een embedded domein specifieke taal (EDSL) omdat ze ingebed zit in een andere taal. Het voordeel is dat er geen volledig nieuwe compiler toolchain gebouwd moet worden. De ontwikkel- en onderhoudskost van de DSL kunnen sterk verminderen, terwijl we toch nog steeds specifieke features kunnen aanbieden voor een bepaald domein. Het is duidelijk dat we geïnteresseerd zijn in het bouwen van een DSL om de FRP-concepten uit de vorige sectie als primitieven aan te kunnen bieden. We richten ons zoals aangegeven in de inleiding op het domein van Internet of Things applicaties of meer algemeen ingebedde softwaresystemen. Dit is een logische keuze aangezien dit domein sterk kan profiteren van de FRP-concepten aangezien ze typisch ook event-gedreven zijn. In het vervolg van deze sectie introduceren we Lightweight Modular Staging, het framework dat ons instaat stelt om een EDSL te bouwen.

2.3.2 Lightweight Modular Staging

LMS is een framework dat een bibliotheek aan basiscomponenten aanbiedt om een codegenerator te implementeren in Scala. Op basis van een codegenerator wordt het eenvoudiger om een Embedded Domein Specific Language aan te bieden. Zoals aangegeven in de vorige sectie kunnen we dus de compiler toolchain van Scala hergebruiken in onze DSL. Het feit dat LMS aangeboden wordt in de vorm van een bibliotheek betekent dat het uitbreidbaar is. We kunnen dus eenvoudig nieuwe optimalisaties en constructies toevoegen die specifiek zijn voor ons domein. Daarnaast kunnen we ook nieuwe doeltalen toevoegen waarin de code gegenereerd zal worden.

LMS laat zoals gezegd toe om een codegenerator te implementeren. Een klassiek computerprogramma ontvangt een aantal invoerargumenten en voert daar berekeningen op uit. De uitvoer van het programma is dan het resultaat van deze berekening. Een codegenerator daarentegen heeft als uitvoer een nieuw computerprogramma, mogelijks in een andere programmeertaal. De taal van het uitvoerprogramma noemen we de doeltaal (*target language*) terwijl de taal waarin de DSL in ingebed zit de hosttaal genoemd wordt, in ons geval Scala. Het meest bekende voorbeeld van een codegenerator is een compiler. Deze vertaalt een computerprogramma, typisch in een



Figuur 2.5: Compilatieproces

hoger-niveau programmeertaal, naar een programma in een lager-niveau programmeertaal. De codegenerator die we samenstellen aan de hand van LMS zal als invoer een programma in de domein specifieke taal hebben. De uitvoer zal een programma zijn in een taal waarvoor het ingebed systeem een compiler aanbiedt. Figuur 2.5 geeft aan hoe dit proces eruit ziet. De codegenerator stelt ons dus in staat om in de EDSL de nodige FRP-concepten aan te bieden. De uitvoer van de codegenerator is dan bijvoorbeeld een standaard C-programma dat opnieuw gecompileerd kan worden om uiteindelijk het uitvoerbare programma te bekomen.
LMS maakt gebruikt van het multi-stage programmeerparadigma. Multi-stage programmeren geeft de ontwikkelaar de nodige bouwstenen om een codegenerator samen te stellen. Bij multi-stage programmeren betekent dit dat men kan aangeven of een expressie tijdens compilatie uitgevoerd zal worden of dat deze uitgesteld (*staged*) moet worden. Talen zoals bijvoorbeeld MetaOcaml hebben speciale syntax om dit onderscheid te maken. LMS daarentegen maakt het onderscheid op basis van types, zoals te zien in tabel 2.1. Dit wil zeggen dat het type van de expressie bepaalt of een expressie *nu* wordt uitgevoerd of pas *later*. Met *later* bedoelen we in dit geval dat de expressies van type $Rep[T]$ aanwezig zullen zijn in de gegenereerde code.

Type van de expressie	Moment van evaluatie
T	Nu - compilation time
$Rep[T]$	Later - staging time

Tabel 2.1: Staging op basis van type

Intern zal LMS een tussentijdse voorstelling (*Intermediate Representation* of *IR*) opbouwen. Deze laat toe om tijdens het compilatieproces de voorstelling van het te genereren programma bij te houden en hierop ook de nodige optimalisaties toe te passen. De laatste stap in dit proces is de vertaling van deze voorstelling naar programmacode in de doeltaal. Omdat LMS geïmplementeerd is als bibliotheek is ook deze laatste vertaalstap uitbreidbaar en kunnen nieuwe doeltalen toegevoegd worden. Dit zal ons in staat stellen om code te genereren voor specifieke talen voor ingebedde systemen. Alvorens in de volgende sectie zo'n specifiek systeem, namelijk Sancus, te tonen waarop we ons verder in de tekst op zullen richten geven we nog een voorbeeldje van *staging* in LMS.

Om deze sectie wat kracht bij te zetten vergelijken we 3 codefragmenten en hoe deze vertaald worden naar een gegenereerd programma. Deze eenvoudige voorbeeldjes geven zo hopelijk een goede intuïtie van de manier waarop expressies uitgesteld kunnen worden naar een volgende *stage*. De inspiratie voor deze voorbeelden komt

van de documentatie [2] van het LMS-framework en daar kunnen de soortgelijke voorbeelden dan ook in zijn volledigheid teruggevonden worden. Om te beginnen bekijken we twee eenvoudige expressies die *nu* ofwel tijdens de uitvoering van de codegenerator geëvalueerd zullen worden:

```
val x: Int = 5
val y = x + 1
```

Zoals aangegeven zijn de types van deze expressies integers en worden deze expressies *nu* geëvalueerd. Ze zullen dus niet aanwezig zijn in de gegenereerde code. Een tweede fragment gebruikt een Rep-type om aan te geven dat we code willen genereren:

```
def compute(b: Boolean): Rep[Int] {
  if(b) 1
  else 0
}
compute(true) + compute(1==1)
```

Het resultaat van de codegenerator in dit geval is de expressie $\ll 1 + 1 \gg$. Dit is misschien verrassend maar aangezien de functie *compute* een argument heeft dat geen Rep-type is zal de codegenerator deze gewoon kunnen uitvoeren. Het resultaat van de functie daarentegen is van type Rep[Int]. Dat betekent dat we optelling van twee Rep-types niet *nu* evalueren maar pas *later*. Concreet betekent dit dat deze expressie aanwezig zal zijn in de gegenereerde code. Pas wanneer de gegenereerde code uitgevoerd wordt zal deze operatie uitgevoerd worden en dit is de moment die we voordien bestempeld hebben als *later*. Om dit voorbeeld door te trekken tonen we nog een laatste codefragment:

```
def compute(b: Rep[Boolean]): Rep[Int] {
  if(b) 1
  else 0
}
val y = compute(true)
```

Omdat we nu een functie *compute* gedefinieerd hebben van type Rep[Boolean] $\rightarrow$ Rep[Int] kan deze niet meer geëvalueerd worden in de codegenerator. De logica van de functie zal daarom *gestaged* moeten worden. Het bijhorende gegenereerde programma zal er als volgt uitzien:

```
bool x = true;
int y;
if(x) y = 1;
else y = 0;
```

Als we willen dat deze functie zelf aanwezig is in de gegenereerde code moeten we ervoor zorgen dat ze van het type Rep[Boolean $\rightarrow$ Int] is.

2.4 Sancus

In de vorige sectie hebben we kort toegelicht hoe we op basis van het LMS-framework een codegenerator kunnen bouwen. We lieten daarbij zien hoe een programma in de domein specifieke taal omgezet kan worden naar een taal waarvoor het ingebed systeem een compiler aanbiedt. Zoals we reeds hebben aangegeven is het relatief eenvoudig in het LMS-framework om een doeltaal toe te voegen. Dit stelt ons in staat om programma's te genereren in verschillende talen maar ook varianten van talen die bepaalde speciale constructies bevatten. In dit werk focussen we op het Sancus [10, 11], waarbij een applicatie geschreven wordt in C met een toevoeging van specifieke macro's om het doel van het framework te bereiken.

2.4.1 Definitie

Sancus is een beveiligd module architectuur voor netwerkgeconnecteerde systemen. Dit wil zeggen dat het toelaat om software toch veilig uit te kunnen voeren op nodes[1] waarvan niet noodzakelijk de gehele softwarestack vertrouwd kan worden. Sancus slaagt erin om met behulp van afgeschermde software modules op een node de lokale toestand op een veilige manier te bewaren en te communiceren met andere modules waarvoor men gekozen heeft deze te vertrouwen ten tijde van installatie van de software. Deze architectuur biedt garanties op gebied van veiligheid bij de executie van de applicatie die geschreven is aan de hand van specifieke macro's om modules te creëren. Voor een volledig overzicht van de garanties die geboden worden verwijzen we naar [10, 11]. Soortgelijke frameworks zoals SGX van Intel [8] bieden gelijkaardige concepten aan en vereisen ook een eigen syntax om afgeschermde modules te creëren en te hanteren. In [14] wordt een overzicht gegeven van een aantal beveiligde software module architecturen.

In dit werk focussen we minder op de veiligheidsgaranties die aangeboden worden maar zijn we eerder geïnteresseerd in de specifieke manier waarop men voor Sancus een programma correct kan opbouwen. Aangezien de focus bij Sancus op netwerkgeconnecteerde ingebedde systemen ligt, zijn de programma's die gebruik maken van de afgeschermde modules ook event-gedreven. Dat betekent dat dit een zeer geschikte kandidaat is als toepassing voor het FRP-framework dat we zullen voorstellen. We zullen dit deel afsluiten met een voorbeeld van een programma voor Sancus vooraleer we een overzicht presenteren van de verschillende concepten die we hebben geïntroduceerd in dit hoofdstuk.

2.4.2 Voorbeeld

Bij wijze van voorbeeld stellen we een kort programma voor geschreven voor het Sancus framework. Wat vooral belangrijk is bij dit voorbeeld is het concept van beveiligde modules enerzijds en de taalconstructies anderzijds om dit concreet uit te

[1] Een node is een klein systeem in een netwerk van ingebedde systemen. Deze nodes zijn vaak beperkt qua rekenkracht waardoor de kracht van zo een netwerk ligt in het aantal nodes.

drukken voor Sancus. Aangezien we uiteindelijk zelf programma's willen genereren voor het Sancus framework moeten we ons daarin verdiepen.

Veronderstel een IoT-applicatie voor een parkeergarage zoals voorgesteld in [10, p.3]. We tonen eerst pseudocode in codefragment 2.1 om het concept van de applicatie uit de doeken te doen. In totaal zien we in het fragment dat er drie modules gecreëerd worden, namelijk *A*, *B* en *C*. Modules kunnen zowel op dezelfde alsook op verschillende nodes uitgevoerd worden. Module A reageert op twee soorten inputevents, *Counter1* en *Sensor1* en genereert zelf ook een event als uitvoer, namelijk *O1* die een overtreding kan aangeven. We zien dat er op een erg procedurele manier vervolgens twee callback-routines geregistreerd worden voor zowel *Sensor1* als voor *Counter1*. De details van de interne logica van de applicatie zijn van ondergeschikt belang. Kort gezegd zal elke parkeerplaats in de garage van een dergelijke node voorzien zijn. De sensor registreert of er al dan niet een auto op de plaats geparkeerd staat. Vanaf dat dit het geval is, wordt er geregistreerd hoelang de auto al aanwezig is. Wanneer de maximale tijd dat de auto in de garage geparkeerd staat overschreden is, zal de node een event versturen op uitvoer *O1*. Module *B* biedt precies dezelfde functionaliteit als de eerste module maar zal uiteraard op een andere node uitgevoerd worden bij een volgende parkeerplaats. Uiteindelijk is er ook een node nodig die alle parkeerovertredingen vaststelt en kan tonen. Daarvoor gebruiken we module *C*. Deze module reageert op twee soorten inputevents *PO1* en *PO2* ofwel de parkeerovertredingen per parkeerplaats. Deze stellen een overtreding voor van elke parkeerplaats in de parkeergarage. Telkens er een overtreding optreedt, zal uitvoer *S* gebruikt worden om de overtreding naar een module te sturen die dit kan afbeelden op een scherm.

Listing 2.1: Sancus applicatie

```
1  module A(Counter1,Sensor1;O1);
2  on Sensor1(x) {
3    if x then taken = 1;
4    else taken = 0; count = 0; O1(0);
5  }
6
7  on Counter1(x) {
8    if taken then count = count + 1;
9    if count > MAX then O1(1);
10 }
11
12 module B(Counter2,Sensor2;O2);
13 ... similar
14
15 module C(PO1,PO2; S);
16 on PO1(x) {
17   if x then c1 = 1 else c1 = 0;
18   if c1 then S(1)
19   if c2 then S(2)
```

```
20 }
21 on PO2(x) ... similar
```

Een laatste belangrijke stap is het koppelen van de modules:

$$O1 \rightarrow PO1$$

$$O2 \rightarrow PO2$$

Zo zien we dat de uitvoer van modules A en B gekoppeld worden aan de input van module C. Een belangrijke opmerking bij deze koppeling is echter dat er geen *type safety* afgedwongen wordt tussen de uitvoer en de invoer. We zien dat zowel de uitvoer als de invoer niet getypeerd is en het is dus de verantwoordelijkheid van de programmeur om ervoor te zorgen dat dit correct is. Dit biedt opnieuw mogelijkheden tot verbetering via onze EDSL. Door specifieke constructies in de EDSL aan te bieden om modules te creëren en ze vervolgens ook aan elkaar te koppelen en daarbij de types wel in rekening te brengen, zullen we type safety kunnen afdwingen voor de koppeling van modules.

Om de lezer een idee te geven hoe de echte code eruit ziet, tonen we in codefragment 2.2 hoe module C opgebouwd kan worden. Daarin zien we alle elementen die belangrijk zijn om met behulp van de nieuwe C-macro's modules op te bouwen. We zien dat elke macro de naam van de module bevat. Op deze manier kan het framework bepalen welke functies en data bij welke module horen. Dit is belangrijk omdat deze functies en data enkel en alleen toegankelijk zijn voor de functies in de module zelf. De enige functie die toegang biedt tot de module vanuit de buitenwereld zijn de zogenaamde *entry points*. Deze toegangspunten in de vorm van functies worden aangegeven door de macro *SM_INPUT*. Alle andere functies en de data die zich in de module bevinden, krijgen de macro *SM_FUNC*, respectievelijk *SM_DATA*. Als laatste kan een uitvoer voor de module gedefiniëerd worden met de macro *SM_OUTPUT*. We zien dat uiteindelijk de verschillende overtredingen via uitvoer s de module verlaten naar een andere module die ze op een scherm zal kunnen tonen. In bijlage D kan de geïnteresseerde lezer de code terugvinden voor zowel module A, B als C. Ook in hoofdstuk 5 zullen we opnieuw aandacht besteden aan Sancus code en voorstellen hoe een FRP-programma opgesteld in onze EDSL vertaald wordt naar een dergelijk Sancus programma.

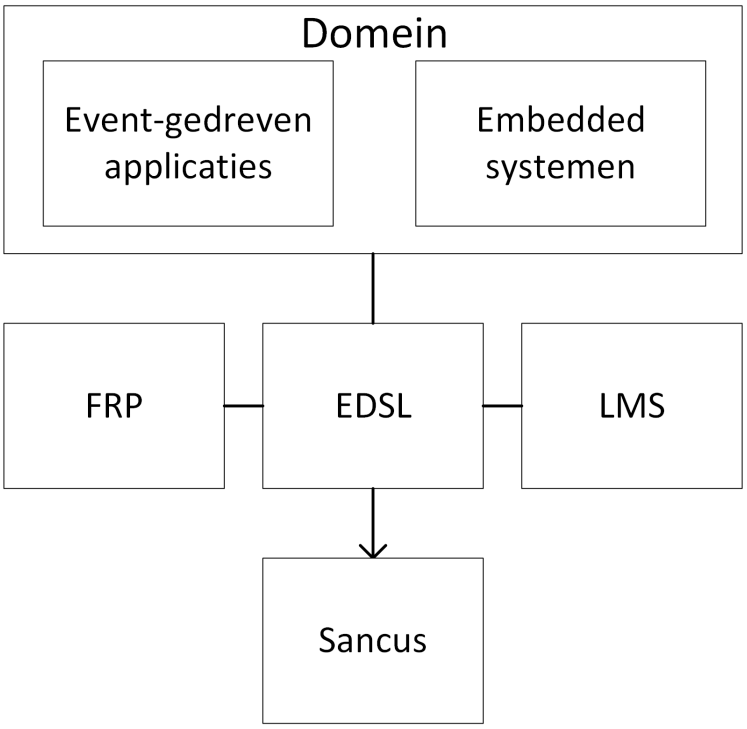
Listing 2.2: Module C

```
1 SM_DATA(modC) uint8_t c1;
2 SM_DATA(modC) uint8_t c2;
3
4 SM_OUTPUT(modC, s);
5
6 SM_FUNC(modC) void printViolations()
7 {
```

```
 8     if(c1) s(1)
 9     if(c2) s(2)
10   }
11
12   SM_INPUT(modC, po1, data, len)
13   {
14     uint8_t violation = data[0];
15     if(violation) c1 = 1; else c1 = 0;
16     printViolations();
17   }
18
19   SM_INPUT(modC, po2, data, len)
20   {
21     uint8_t violation = data[0];
22     if(violation) c2 = 1; else c2 = 0;
23     printViolations();
24   }
```

2.5 Overzicht

Omdat we in dit hoofdstuk een brede variëteit aan concepten hebben toegelicht, stellen we in deze laatste sectie een overzicht voor om de relatie tussen alle concepten nog even opnieuw te duiden. Dit kan de lezer helpen het overzicht te bewaren en de verschillende concepten in de juiste context te plaatsen met betrekking tot dit werk. In figuur 2.6 zien we de verbanden die ze met elkaar hebben. We introduceren in dit werk een *EDSL* die een aantal nieuwe primitieve operaties toevoegt aan de hosttaal waarin deze geïmplementeerd is. De primitieven die we introduceren richten zich op het domein van event-gedreven applicaties. Deze zijn niet alleen typisch voor grafische gebruikersinterfaces maar komen vaak ook terug in Internet of Things applicaties. Dit zijn applicaties waarbij de software uitgevoerd wordt op een netwerk van ingebedde systemen die met elkaar communiceren. Het domein van event-gedreven systemen wordt dus verder gespecialiseerd naar ingebedde systemen. Functioneel reactief programmeren introduceert een aantal abstracties die het natuurlijker maakt om event-gedreven applicaties te ontwikkelen en vormt de basis voor wat er aangeboden wordt in de EDSL. De EDSL die we voorstellen in dit werk is opgebouwd met het Lightweight Modular Staging framework. De codegenerator die het invoerprogramma, geschreven in de domein specifieke taal, omzet naar de doeltaal is eenvoudig uitbreidbaar. Dit stelt ons in staat om aan de codegenerator een nieuwe doeltaal toe te voegen en op die manier vanuit eenzelfde programma met FRP-abstracties verschillende doelprogramma's te genereren. We richten ons in dit werk dan ook tot het genereren van programma's voor het Sancus framework, een beveiligde architectuur waarin software met een aantal veiligheidsgaranties uitgevoerd kan worden in een netwerk van intergeconnecteerde nodes. In dit framework



Figuur 2.6: Overzicht van de geïntroduceerde concepten

kunnen verschillende software modules opgebouwd worden die nadien ook aan elkaar gekoppeld kunnen worden. In de originele constructies waarmee deze modules en koppelingen aangegeven kunnen worden is er echter geen garantie dat de koppeling type safe is. Door onze EDSL uit te breiden en extra primitieve operaties te voorzien voor het aanmaken en koppelen van modules, kunnen we alsnog de modules type safe koppelen. Naast dit voordeel zullen we uiteraard ook gebruik kunnen maken van de FRP-abstracties om programma's op te bouwen voor het Sancus framework.

Hoofdstuk 3

FRP API

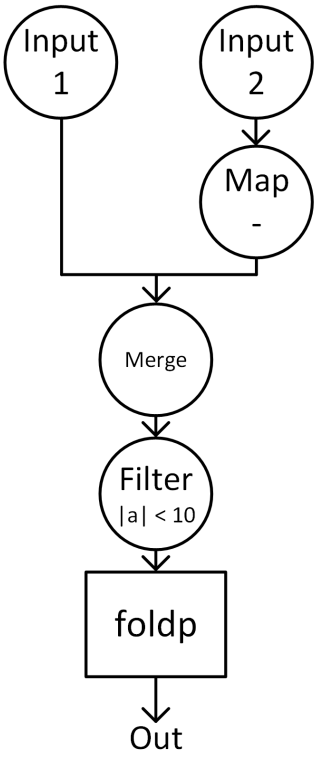
Aan het einde van het vorige hoofdstuk gaven we met een overzicht aan hoe de belangrijkste concepten voor dit werk met elkaar in verband staan. Vooraleer we het hebben over de implementatie van de EDSL, introduceren we in dit hoofdstuk hoe deze gebruikt kan worden. Een voorbeeldapplicatie wordt voorgesteld om te tonen hoe de FRP API die we aanbieden er concreet uitziet. We stellen daarbij ook een grafische manier voor om een FRP-programma voor te stellen aan de hand van een graaf. Dit hoofdstuk focust dus op het gebruik van de EDSL vanuit het perspectief van de gebruiker ervan. Pas in het volgende hoofdstuk zullen we ons richten op de implementatie van de EDSL.

3.1 Een voorbeeld

Om te beginnen introduceren we een eenvoudige event-gedreven applicatie die we zullen vertalen naar een programma dat gebruik maakt van de EDSL die we voorstellen. Op deze manier kunnen we voor een eerste keer kennis maken met de FRP API vanuit het standpunt van een gebruiker ervan. We introduceren de specifieke operaties om events en behaviors aan te maken en om te vormen.

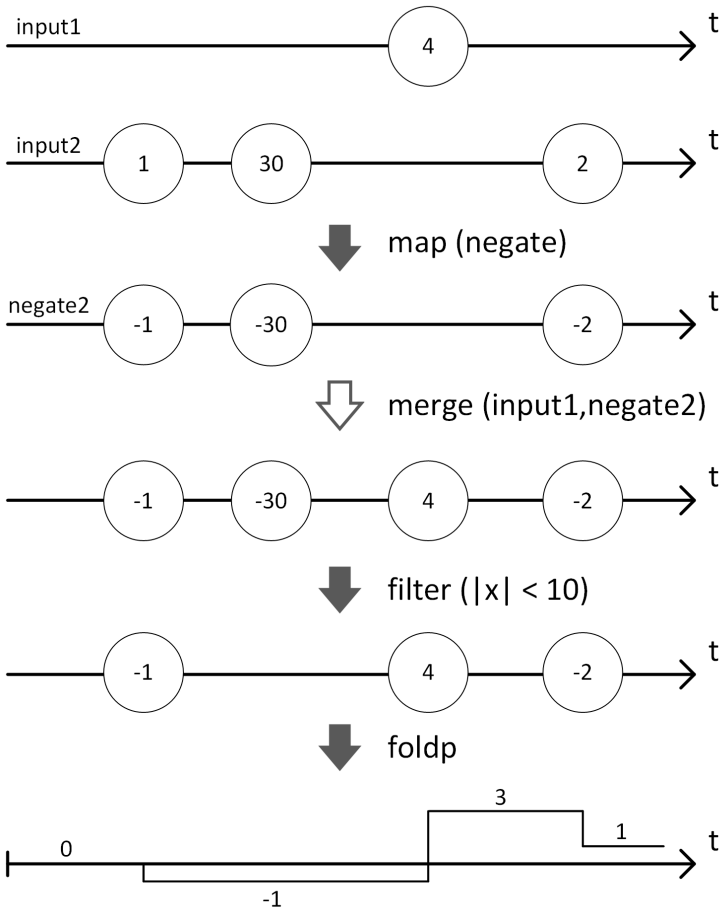
Veronderstel een event-gedreven programma dat reageert op twee verschillende *input*events. Beide events hebben een waarde van type *integer*. Wanneer een event ontvangen wordt op *input1* willen we deze waarde optellen bij een interne teller. Wanneer op *input2* een event ontvangen wordt, trekken we deze waarde af van de teller. Als extra voorwaarde willen we afdwingen dat een waarde groter dan of gelijk aan 10 geen effect mag hebben op de teller. Om dit programma te definiëren zullen we gebruik maken van de FRP API om events en behaviors voor te stellen en te transformeren. Het programma zelf kunnen we eenvoudig voorstellen in de vorm van een graaf zoals in figuur 3.1. Op basis van deze voorstelling leggen we de verschillende operaties uit.

Om te beginnen definiëren we de twee inputevents, *input1* en *input2*. We gebruiken de map-operatie op *input2* om elke waarde in de eventstroom te transformeren



Figuur 3.1: Voorbeeldapplicatie

naar een andere waarde. Omdat we de waarde van *input2* willen aftrekken van de teller transformeren we de originele waarde naar diens negatie. Om vervolgens beide stromen te combineren tot één enkele stroom gebruiken we de merge-operatie. Dit wil zeggen dat de resulterende stroom zowel waarden afkomstig van *input1* als *input2* kan bevatten. Om er vervolgens voor te zorgen dat de waarde die we zullen optellen of aftrekken bij de interne teller kleiner is dan 10, gebruiken we de filter-operatie. Deze operatie resulteert in een eventstroom die alleen waarden zal bevatten die voldoen aan het predicaat dat bij deze filter-operatie hoort. Tot op dit moment hebben we de twee invoerstromen omgevormd naar een stroom van positieve en negatieve waarden waarvan de absolute waarde steeds kleiner is dan 10. Zoals we bij de definitie van events in sectie 2.2.1 hebben aangegeven draagt de eventstroom slechts op een welbepaald moment een waarde en kan deze niet gebruikt worden om toestand op te slaan. Om de interne teller te construeren hebben we nood aan een behavior. Met behulp van de foldp-operatie kunnen we een eventstroom transformeren naar een behaviorwaarde. Merk op dat de voorstelling van een behavior in de grafische voorstelling aangeduid wordt met een rechthoek, tegenover een cirkel voor de events. Bij de definitie van een behavior in sectie 2.2.1 zagen we al dat een behavior constant gedefinieerd is en op die manier kan men daarmee een toestand opbouwen. De foldp-operatie staat voor *fold from the past*. Dit wil zeggen dat we de huidige waarde opbouwen op basis van waarden die zich in het verleden hebben voorgedaan. Concreet



Figuur 3.2: Visualisatie doorheen de tijd

wil dit zeggen dat telkens er een nieuwe waarde ontvangen wordt in de foldp-node, de huidige waarde van de behavior aangepast wordt. Deze aanpassing gebeurt op basis van de functie die bij de foldp-operatie hoort en we zien zo dadelijk hoe deze eruit ziet. Uiteindelijk zal deze waarde ten allen tijden de huidige waarde voorstellen van onze teller. Om dit voorbeeld nog iets te verduidelijken tonen we in figuur 3.2 een mogelijke uitvoering van het programma doorheen de tijd. Dit is slechts één mogelijk uitvoeringspad omdat het pad bij een event-gedreven programma bepaald wordt door de events die optreden.

Om het voorbeeld te vervolledigen tonen we het programma om deze applicatie voor te stellen in de EDSL. We zien in codefragment 3.1 hoe de eventstromen in elkaar en uiteindelijk in de behaviorwaarde worden omgezet. Zoals besproken definiëren we aan het begin twee inputstromen, waarvan de waardes van *input2* naar de tegengestelde waarde gemapt worden. Merk op dat bij de merge-operatie er nog een extra functieparameter vereist is. Deze functie wordt gebruikt als beide stromen op hetzelfde moment een waarde propageren. Deze functie wordt dan gebruikt om te

bepalen welke nieuwe waarde aanwezig zal zijn in de resulterende eventstroom. Zoals vermeld wordt de functie in de foldp-operatie gebruikt om de huidige waarde aan te passen op basis van de eventwaarde die ontvangen wordt. Aangezien we ervoor moeten zorgen dat de behaviorwaarde *counter* steeds gedefinieerd is, is een tweede argument nodig om de beginwaarde te bepalen.

Merk op aan de code van dit FRP-programma dat er dus geen expliciete toestand aangepast moet worden. We kunnen op een hoog-niveau aangeven wat we bedoelen met onze applicatie en dit komt de leesbaarheid ten goede.

Listing 3.1: Voorbeeldapplicatie

```
1 val input1   = InputEvent[Int]
2 val input2   = InputEvent[Int]
3 val negate2  = input2.map( (i) => -i)
4 val merged   = input1.merge(negate2, (x,y) => x + y)
5 val filtered = merged.filter( x => abs(x) < 10)
6 val counter  = filtered.foldp((x,state) => state + x, 0)
```

3.2 Overzicht van de API

Deze sectie bevat een overzicht van de totale FRP API en kan gebruikt worden als referentie bij het opstellen van programma's met de ontwikkelde EDSL. Programma's die deze FRP API gebruiken noemen we FRP-programma's. Deze vormen de invoer voor de codegenerator die we in het volgende hoofdstuk zullen introduceren. Omdat deze codegenerator geïmplementeerd is in LMS zal dit ook invloed hebben op de vorm van deze API, meer bepaald op de types die daarin voorkomen. In het volgende hoofdstuk wordt dit ruim toegelicht. De gebruiker van de API hoeft dit uiteindelijk niet te weten dus vandaar dat voorstelling van de API zoals hier gepresenteerd nog steeds een goede referentie vormt. Het is alleen al goed om in het achterhoofd te houden dat de eigenlijke FRP API er in werkelijkheid uitziet zoals weergeven in bijlage C.

3.2.1 Event

In codefragment 3.2 zien we een voorstelling van de basis API voor events. Operaties waar we nog geen aandacht aan hebben besteed zijn *constant*, en *startsWith*. De eerste operatie spreekt voor zich en er zal steeds een event doorgegeven worden met een constante waarde. De *startsWith*-operatie produceert een nieuw behavior dat steeds de waarde aanneemt van het event dat het ontvangt. Dit is dus in principe de meest eenvoudige manier om een toestand op te bouwen. De overige operaties zijn reeds in het voorbeeld aan bod gekomen.

Listing 3.2: Event API

```
trait Event[A] {
  def constant[B] (c: B): Event[B]
  def map[B](f: A => B): Event[B]
  def filter(f: A => Boolean): Event[A]
  def merge(e:Event[A], f:(A,A)=>A): Event[A]
  def startsWith(i: A): Behavior[A]
  def foldp[B](f:(A,B) => B, init:B): Behavior[B]
}
```

3.2.2 Behavior

In codefragment 3.3 wordt de API voor behaviors voorgesteld. De enige operatie die we nog niet hebben toegelicht is *changes*. Deze kan men gebruiken om een behavior om te vormen naar een eventstroom. Telkens de behaviorwaarde geüpdatet wordt zal er een event met deze waarde aan de eventstroom worden toegevoegd.

Listing 3.3: Behavior API

```
trait Behavior[A] {
  def map2[B,C](b: Behavior[B], f:(A, B) => C): Behavior[C]
  def snapshot[B](e: Event[B]): Event[A]
  def changes(): Event[A]
}

def ConstantB[A](value: A): Behavior[A]
```

Hoofdstuk 4

Basisimplementatie

In het vorige hoofdstuk kwam aan bod hoe de EDSL gebruikt kan worden om een FRP-programma op te stellen. In dit hoofdstuk bekijken we de implementatie ervan en hoe het compilatieproces intern is opgebouwd. De focus ligt hier dus voornamelijk op de technische realisaties van het werk.

We bekijken eerst op een conceptuele manier hoe een invoerprogramma wordt omgevormd naar een interne representatie van het programma. Daarbij worden enkele belangrijke aspecten verduidelijkt waarop we moeten letten bij de FRP-implementatie om een correcte uitvoering van het programma te waarborgen. In sectie 4.2 lichten we toe hoe we deze interne representatie kunnen omvormen naar een tussentijdse voorstelling in het LMS-framework zoals ingeleid in sectie 2.3.2. Nadien lichten we toe hoe LMS vanuit deze voorstelling een programma genereert in de doeltaal. Om dit hoofdstuk af te sluiten presenteren we een overzicht van het gehele codegeneratieproces dat aan bod komt in dit hoofdstuk.

4.1 Van FRP-programma naar interne representatie

In sectie 3.1 gaven we een eenvoudig voorbeeld van een event-gedreven programma dat gebruik maakt van de EDSL. In deze sectie stellen we nu een manier voor om hiervan een interne representatie te maken. Met deze representatie zullen we de relaties tussen de verschillende eventstromen en behaviors vastleggen. We lichten eerst toe hoe deze voorstelling eruit zal zien. Daarna focussen we op de belangrijkste elementen in deze voorstelling en hoe deze van belang zijn om uiteindelijk een correct werkend programma te genereren.

4.1.1 Voorstelling met een graaf

De grafische voorstelling die we introduceerden in hoofdstuk 3 geeft een goede voorstelling van de manier waarop we de relaties tussen events en behaviors kunnen vastleggen. De eerste stap is dan ook om de het FRP-programma, de invoer van de codegenerator, om te vormen naar een graaf. Elke methode in de FRP API kan

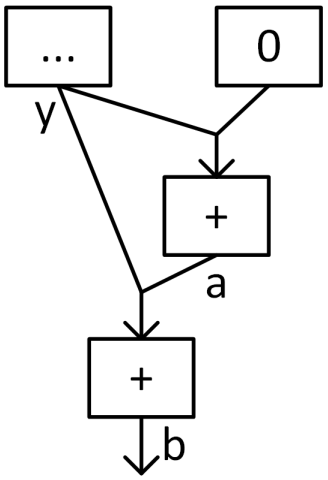
gebruikt worden om ofwel een nieuwe eventstroom of behavior te creëren of om deze om te vormen. Het idee is om elke operatie die voorkomt in het programma voor te stellen als een node in een graaf. Aangezien elke operatie een nieuwe eventstroom of een behavior voortbrengt, zullen we in het verdere verloop van de tekst een knoop dan ook associëren met zowel de operatie als de stroom of de behaviorwaarde die uit die operatie ontstaat. Zoals we al zagen in figuur 3.1 wordt een nieuwe eventstroom voorgesteld met een cirkel en een nieuwe behavior met een rechthoek. Elke node in onze graaf is specifiek voor een operatie en houdt intern alle nodige informatie bij om zijn functie te vervullen. Als voorbeeld bekijken we de foldp-operatie uit codevoorbeeld 3.1:

```
val counter  = filtered.foldp((state,x) => state + x, 0)
```

Bij het aanmaken van een node voor deze operatie zal zowel de initiële waarde van de behavior alsook de functie opgeslagen worden. Herinner u dat de functie gebruikt zal worden om de behaviorwaarde gepast aan te passen wanneer er van eventstroom *filtered* een event ontvangen wordt. Naast deze specifieke informatie per operatie, zal elke node informatie bevatten om de context van deze node vast te leggen. Elke node moet een lijst van ouder- en kindernodes bevatten. Verder bevat elke node een indicatie van het niveau (*level*) waarop de node zich bevindt in de graaf. Alle nodes in de graaf zullen dus verdeeld zijn over verschillende niveaus. We zullen beginnen met uit te leggen waarom dit belangrijk is om pas daarna te verduidelijken hoe dit niveau bepaald wordt.

4.1.2 Volgorde van propagatie en glitchpreventie

Om te begrijpen waarom het belangrijk is dat de nodes zich op een bepaald niveau in de graaf bevinden, moeten we even in detail bekijken hoe het ontvangen van een nieuw event aan het begin van de graaf een reeks bewerkingen zal teweeg brengen. Een event komt toe in het systeem via één van de bovenste nodes, ofwel de inputnodes. Conceptueel zal de waarde dan doorheen de graaf propageren en telkens wanneer het een node op zijn pad vindt zal de operatie die bij deze node hoort worden toegepast op deze waarde. Een event sijpelt van boven naar beneden doorheen de graaf en de waarde van dit event wordt mogelijks op zijn weg naar beneden aangepast door de nodes. We kunnen de nodes ook zien als een schakel tussen de verschillende eventstromen en behaviors die het met elkaar verbindt. Telkens wanneer een node een event ontvangt, zal het de bijhorende operatie uitvoeren op het event en de resulterende waarde zal als een nieuw event aan de stroom worden toegevoegd die deze node heeft gecreëerd. De volgorde waarin een event door de graaf propageert is echter van groot belang. De beste manier om dit aan te tonen is aan de hand van een voorbeeld.



Figuur 4.1: Visualisatie van de relaties

Listing 4.1: Voorbeeld propagatie

```
1 val y : Behavior[Int] = ...
2 val a = y + 0;
3 val b = y + a;
```

Beschouw het eenvoudige voorbeeldprogramma[1] in codefragment 4.1, geïnspireerd op een voorbeeld uit [9, p.12]. We veronderstellen daarin een numerieke behaviorwaarde y waarvan we niet hoeven te weten hoe deze werd aangemaakt. Wanneer we hiervan een graafvoorstelling maken zoals in figuur 4.1 zien we dat behaviorwaarde a afhankelijk is van behaviorwaarde y en dat behaviorwaarde b afhankelijk is van zowel y als a. Conceptueel zal het FRP-framework ervoor moeten zorgen dat de waarden van de behaviors op een consistente manier geüpdatet worden. Daarmee bedoelen we dat de operaties die bij de verschillende nodes horen in de juiste volgorde uitgevoerd moeten worden om ervoor te zorgen dat de waarden van de behaviors steeds correct aangepast worden. We verwachten immers dat a steeds dezelfde waarde heeft als y en dat b steeds de dubbele waarde heeft van y.

Het propageren van een event wil in principe niets anders zeggen dan het introduceren van een nieuw event op alle eventstromen of het aanpassen van de behaviorwaarde die afhankelijk zijn van dit nieuwe event. Voor ons FRP-framework betekent dit concreet dat we de node die bij een event of behavior hoort steeds moeten herevalueren als een van de bovenliggende nodes geherevalueerd is. Wanneer er een event toekomt in het FRP-programma zal er dus een waterval-effect ontstaan waarbij alle relevante nodes opnieuw geëvalueerd worden waardoor op elke eventstroom of behaviorwaarde

[1] Merk op dat in dit voorbeeld de + operatie is gedefinieerd op behaviors. Deze operatie kan geïmplementeerd worden aan de hand van de map2-operatie.

een nieuwe waarde geïntroduceerd wordt. Dit is exact waar functioneel reactief programmeren om draait en wat de basis van ons framework vormt. Wanneer de waarde van behavior y wijzigt van waarde 2 naar waarde 3, zal dus zowel a als b aangepast moeten worden door het FRP-framework. Er zijn nu twee scenario's die kunnen voorvallen. In het eerste scenario zal eerst eerst node b geëvalueerd worden en daarna node a. Merk op dat na de evaluatie van node a, node b ook opnieuw geëvalueerd zal worden aangezien deze ook afhankelijk is van node a en bijgevolg geüpdatet zal worden als deze waarde wijzigt.

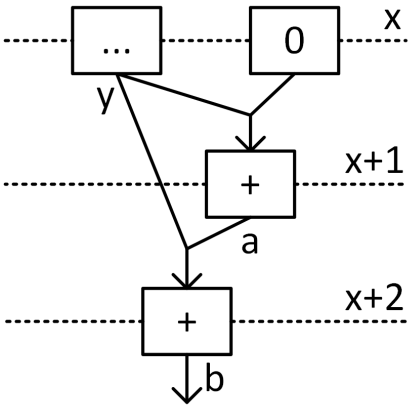
Initiële toestand	y = 2	a = 2	b = 4
wijziging y	**y = 3**	a = 2	b = 4
Evaluatie node b	y = 3	a = 2	**b = *5***
Evaluatie node a	y = 3	**a = 3**	b = *5*
Evaluatie node b	y = 3	a = 3	**b = 6**

In het tweede scenario zal eerst node a en pas daarna node b geëvalueerd worden.

Initiële toestand	y = 2	a = 2	b = 4
wijziging y	**y = 3**	a = 2	b = 4
Evaluatie node a	y = 3	**a = 3**	b = 4
Evaluatie node b	y = 3	a = 3	**b = 6**

Het is logisch dat pas na uitvoering van alle operaties alle waarden terug consistent zijn met voorwaarden die we gesteld hebben. In het eerste scenario stellen we echter vast dat waarde b tijdelijk inconsistent is. We kunnen hier het verschil dus maken tussen waarden die nog niet zijn geüpdatet tegenover waarden die tijdelijk verkeerd zijn. Op het moment dat b voor de eerste keer geüpdatet werd, is er een zogenaamde *glitch* opgetreden. Net om dit soort gedrag te vermijden, moeten we ervoor zorgen dat een event in de correcte volgorde doorheen de graaf gepropageerd wordt. In het tweede scenario zien we dat deze glitch niet optreedt. Een belangrijk tweede aspect aan het belang van deze volgorde is dat in het tweede scenario het aantal evaluaties van nodes lager is. Een derde verschilpunt tussen beide scenario's is het aantal keer dat een specifieke node geëvalueerd wordt. In het eerste scenario wordt de node die bij behavior b hoort tweemaal uitgevoerd. Dit is belangrijk omdat een functie die bij de node hoort mogelijks de toestand van het systeem tweemaal verandert, zoals bijvoorbeeld de foldp-operatie. Het is daarom ongewenst om een node tweemaal te evalueren bij de propagatie van één en hetzelfde inputevent.

Het voorbeeld heeft dus aangetoond dat de volgorde van evaluatie belangrijk is. Dit is precies waarom we hierboven in sectie 4.1.1 verschillende niveaus introduceren in de graaf. In figuur 4.2 zien we hoe de nodes uit het voorbeeld geordend moeten worden om de correcte volgorde te respecteren. We kunnen bij het opbouwen van de graaf eenvoudig bepalen op welk niveau de node zich moet bevinden. De inputnodes krijgen uiteraard niveau 0 toegekend. De andere nodes moeten ervoor zorgen dat ze



Figuur 4.2: Visualisatie van de relaties met niveaus

zich op een lager niveau bevinden dan hun oudernode(s). In het geval dat er slechts één oudernode is met niveau x, dan zal het niveau van de huidige node x+1 moeten zijn. Wanneer er twee oudernodes zijn met niveaus x en y zoals bijvoorbeeld bij de mergenode, dan zal de mergenode zelf niveau max(x,y)+1 krijgen Voor nodes op hetzelfde niveau maakt het niet uit in welke volgorde ze geëvalueerd worden aangezien ze niet van elkaar afhankelijk zijn.

4.1.3 Wegcompileren van de graaf

Terwijl we in de deze sectie de nodige aandacht besteden aan de interne representatie van het programma in de vorm van een graaf moeten we ons goed voor ogen houden dat deze graaf wordt opgesteld om uiteindelijk een correct programma te genereren. Zoals aangegeven in de inleiding willen we er echter voor zorgen dat het gegenereerde programma zo efficiënt mogelijk is. Een belangrijke bijdrage van onze EDSL is dan ook dat we deze graaf in de mate van het mogelijke willen *wegcompileren*. We streven er in onze implementatie naar dat deze graaf zo min mogelijk aanwezig is in de gegenereerde code.

In deze sectie is duidelijk geworden dat we een graaf gebruiken om het FRP-programma intern voor te stellen. Aangezien we deze graaf echter willen wegcompileren moeten we een manier vinden om de relaties tussen de verschillende nodes toch vast te leggen in het gegenereerde programma. Dit kunnen we doen door de relaties als vast te beschouwen vanaf dat we het volledige FRP-programma intern hebben voorgesteld. Concreet betekent dit voor het uitvoerprogramma dat de relaties tussen de verschillende nodes verankerd zijn in de code. Dit betekent dat we een FRP-framework aanbieden voor een statische graaf, wat ook wel eens eerste-orde FRP genoemd wordt. Een statische FRP-graaf heeft het voordeel dat we gespecialiseerde code kunnen genereren voor deze graaf zonder dat het gegenereerde programma al de informatie van de graaf moet bevatten. Het zal met onze EDSL

niet mogelijk zijn om een programma te genereren voor een ingebed systeem dat tijdens executie de graaf nog kan veranderen. Omdat de graaf vast ligt, is er dus niet zoiets als een event-manager nodig die *at runtime* zal moeten bepalen welke nodes uitgevoerd moeten worden. Merk op dat dit een vereiste is als we de relaties tussen de verschillende nodes tijdens de uitvoering van het FRP-programma nog willen wijzigen. Net door deze relaties vast te leggen, spreken we van het wegcompileren van de graaf. Het is logisch en ook noodzakelijk dat de relaties die de graaf vastlegt tussen de verschillende nodes wel in de gegenereerde code aanwezig zijn. In de volgende sectie behandelen we de manier waarop we dit realiseren met de hulp van het LMS-framework.

4.2 Van interne representatie naar tussentijdse voorstelling in LMS

In de vorige sectie hebben we besproken hoe het FRP-programma omgezet kan worden naar een interne representatie ervan. Door de verschillende operaties in een graaf te gieten, slagen we erin om de relaties die ze tot elkaar hebben vast te leggen. Er zijn heel wat FRP-frameworks [3, 9] waarbij dit soort datastructuur wel volledig aanwezig is tijdens het uitvoeren van het FRP-programma. Op basis van deze graaf kan dan bij een nieuw event bepaald worden welke nodes er allemaal aangepast moeten worden bij een bepaald event. De aanwezigheid van deze graaf tijdens executie brengt echter ook een zekere overhead met zich mee. Deze datastructuur moet echter niet verplicht aanwezig zijn in de gegenereerde code wanneer we ons richten op eerste-orde FRP. Zoals in de vorige sectie al aangehaald werd, zorgen we er in onze implementatie voor dat deze graaf statisch is en zo kan worden weggecompileerd. Daarmee bedoelen we dat we slechts de hoogst nodige informatie in het uiteindelijk gegenereerde programma willen om de toekomende events toch correct af te handelen. Door ons in de gegenereerde code te specialiseren op één specifieke graaf trachten we aan efficiëntie te winnen.

In het vervolg van deze sectie bespreken we algemeen hoe we erin slagen om de FRP-graaf weg te compileren. De graaf wordt omgezet naar een functie per inputnode. Om deze functie op te bouwen werken we top-down. Het idee is dat deze functie alle nodige operaties uitvoert die nodig zijn om een nieuw event af te handelen dat in het systeem toekomt via de inputnode waar deze functie toe behoort. De top-down aanpak komt het meest natuurlijk overeen met de manier waarop we conceptueel hebben uitgelegd hoe een event door de graaf propageert. We noemen dit soort functie een *toplevelfunctie* en voor elke inputnode zullen we een toplevelfunctie opstellen. Er ontstaat een 1-op-1-mapping tussen een bepaald inputevent en de functie die deze volledig zal afhandelen.

Het is belangrijk om in te zien dat we in deze sectie twee stappen moeten zetten. Het uiteindelijke doel is om de toplevelfuncties waarvan we willen dat ze aanwezig zijn in de gegenereerde code te *stagen* in het LMS-framework. Dit betekent

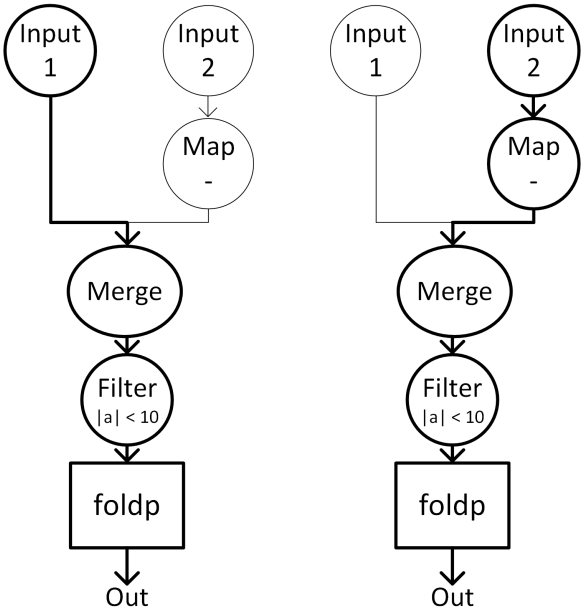
dat we deze functies uitstellen om pas later (op het ingebed systeem) uitgevoerd te worden. Vooraleer we bespreken hoe we erin slagen om deze functies uit te stellen moeten we dus eerst goed voor ogen hebben wat we precies willen *stagen*. Precies daarom gaan we in het begin van deze sectie eerst iets dieper in op de top-down aanpak en hoe we zo het programma opdelen in verschillende toplevelfuncties en hoe deze er concreet moeten uitzien. Om aan te geven dat we de functies die we opstellen uiteindelijk willen genereren, stellen we deze functies in C voor. Pas wanneer we een goed idee hebben wat we willen genereren, lichten we in sectie 4.2.5 toe hoe we deze functies kunnen *stagen* en zo de graaf kunnen omvormen naar een tussentijdse voorstelling in LMS.

4.2.1 Top-down opbouw

De omvorming van de graaf naar de verschillende toplevelfuncties gebeurt best van boven naar beneden of *top-down*. Dit is het meest natuurlijke werkwijze aangezien een event toekomt in een van de inputnodes en zijn weg baant doorheen de graaf. Om een toplevelfunctie op te stellen voor een bepaalde inputnode beginnen we bij het oplijsten van alle nodes die van deze input afhankelijk zijn. We gaven reeds aan in sectie 4.1.1 dat elke node een lijst bevat van alle kindnodes ofwel alle nodes die rechtstreeks afhankelijk zijn van deze node. Startende van de inputnode kunnen we een lijst van nodes opstellen die rechtstreeks afhankelijk zijn van de inputnode door de graaf te doorkruisen [2]. Wanneer we een event ontvangen op de inputnode zal elke node in deze lijst geëvalueerd moeten worden om het systeem volledig te updaten. Aan de hand van deze lijst kunnen we de topologische volgorde bepalen op basis van de niveaus waarop de nodes zich moeten bevinden. Het belang hiervan is reeds aangetoond in de sectie 4.1.2 en voor nodes van hetzelfde niveau is de volgorde van geen belang. In figuur 4.3 zien we hoe de voorbeeldapplicatie uit sectie 3.1 omgevormd wordt tot twee toplevelfuncties. De graaf bevat alle informatie om vanuit de inputnode alle nodes te vinden die de toplevelfunctie zal moeten bevatten.

Een belangrijke opmerking bij deze toplevelfuncties is dat we ervan uitgaan dat twee toplevelfuncties niet tegelijkertijd uitgevoerd zullen worden. De belangrijkste motivatie hiervoor is omdat in een ingebed systeem de events normaal gezien opgewekt worden via interrupts. Aangezien we ervan uitgaan dat een interrupt eerst volledig afgehandeld moet zijn alvorens de volgende afgewerkt kan worden, kunnen we deze lijn doortrekken voor inputevents. Dit heeft als gevolg voor onze specifieke voorbeeldapplicatie dat bijvoorbeeld de merge-operatie in onze applicatie nooit gebruik zal moeten maken van de extra functie, die we moeten opgeven bij de merge-operatie. Herinner u dat deze functie, de *mergefunctie*, gebruikt zal worden om eventuele waardes van de linker- als de rechtertak te combineren indien deze beide tegelijkertijd een waarde propageren. Omdat we veronderstellen dat verschillende inputnodes niet tegelijkertijd een event zullen propageren kunnen we een aantal veronderstellingen maken. Zo zal in ons voorbeeld de linker- en rechtertak niet tegelijkertijd een event kunnen propageren aangezien ze geen inputnode gemeen

[2] Om dit efficiënt en eenvoudig te houden, hebben we opgelegd dat een graaf acyclisch is.

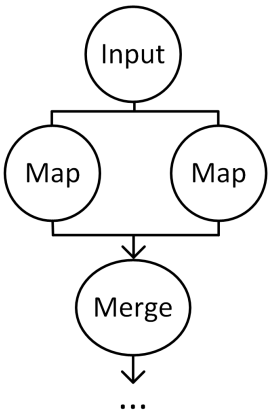


Figuur 4.3: 2 toplevelfuncties in de voorbeeldapplicatie

hebben. De mergefunctie is echter alleen overbodig in dit specifieke voorbeeld en zoals we kunnen zien in de grafische voorstelling van een willekeurig ander FRP-programma in figuur 4.4 zal in dit geval de mergefunctie wel nodig zijn. Meer nog, in dit geval zal de mergefunctie steeds nodig zijn aangezien beide takken die toekomen in de mergenode slechts afhankelijk zijn van één en dezelfde inputnode. Indien deze inputnode een event propageert, zal dus zowel links als rechts in de mergenode een event toekomen. In het geval dat beide takken van de mergenode zowel gemeenschappelijke als niet-gemeenschappelijke inputnodes hebben, zal de functie soms wel en soms niet gebruikt moeten worden. Ook de aanwezigheid van een filternode kan ervoor zorgen dat de functie al dan niet nodig zal zijn. Dit komt omdat de filternode ervoor kan zorgen dat een event al dan niet stopt met door de graaf te propageren.

4.2.2 Opbouw van een toplevelfunctie

Een FRP-programma wordt omgezet naar een reeks toplevelfuncties. Zoals in figuur 4.3 is het vaak het geval dat de operatie die bij een bepaalde node hoort in beide toplevelfuncties aanwezig zal moeten zijn door bijvoorbeeld de mergenode. Om codeduplicatie te voorkomen in de toplevelfuncties moeten we ervoor zorgen dat de operatie die bij een node hoort herbruikbaar is door verschillende toplevelfuncties. Een logische oplossing voor dit probleem is om per node een functie te genereren. Nadien kan elke toplevelfunctie samengesteld worden door simpelweg de correcte functies per node op te roepen in de juiste volgorde. In sectie 4.2.4 gaan we dieper in op het nut van de toplevelfuncties en waarom we niet simpelweg in de nodefuncties de



Figuur 4.4: Merge operatie met dezelfde inputstroom

volgende nodefunctie oproepen. We bekijken twee manieren om de toplevelfuncties op te bouwen. In de eerste aanpak gaan we heel pragmatisch te werk en zorgen we ervoor dat alle details voor correcte propagatie worden afgehandeld in de nodefuncties. Het voordeel is dat de toplevelfuncties nu zeer eenvoudig zijn. Daarna presenteren we een optimalere manier waardoor de toplevelfunctie iets complexer is.

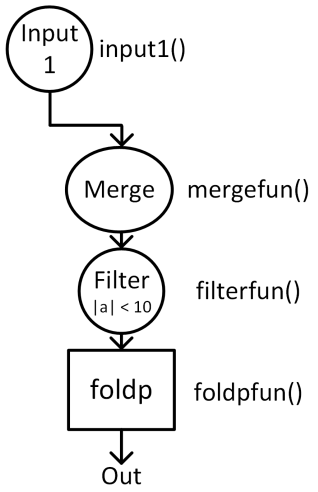
Eenvoudige opbouw

We zorgen ervoor dat elke functie alle logica bevat die nodig is om zijn bewerking uit te voeren. In figuur 4.5a hebben we de nodes afgezonderd die nodig zijn voor de eerste toplevelfunctie zoals te zien was in figuur 4.3. Op basis van dit voorbeeld kunnen we aantonen hoe we te werk gaan. Deze aanpak zorgt ervoor dat elke nodefunctie op zichzelf staat en de toplevelfunctie simpelweg de nodes in topologische volgorde moet oproepen. Hoewel we pas in de volgende sectie laten zien hoe de individuele nodefuncties zijn opgebouwd, is het voorlopig voldoende om in te zien dat deze nodefuncties globale data nodig hebben om gegevens uit te wisselen. Dit is logisch omdat nodes afhangen van de data die diens oudernode(s) produceren. Aangezien de nodefuncties geen argumenten ontvangen wil dit zeggen dat de data die bij de node hoort globaal toegankelijk moet zijn. Deze aanpak zorgt ervoor dat de opbouw van de toplevelfunctie eenvoudig blijft[3]. Vooraleer we de implementatie tonen van de nodefuncties, bekijken we een meer optimale implementatie van de toplevelfuncties en waarom dit relevant is.

Optimalisatie

Om aan te tonen waarom een optimalisatie op zijn plaats is, is het belangrijk om in te zien dat niet alle data van de nodes globaal hoeft te zijn. Herinnert u zich uit sectie 2.2.1 dat een eventstroom geen continu gedefinieerde waarde heeft zoals een behavior.

[3]Naast een eenvoudigere toplevelfunctie was ook een uitbreiding van het LMS-framework nodig. Later meer hierover.



(a) Inputnode en afhankelijke nodes

```
void topfunctie_input1()
{
  input1();
  mergefun();
  filterfun();
  foldpfun();
}

void input1() { ... }
void mergefun() { ... }
void filterfun() { ... }
void foldpfun() { ... }
```

(b) Toplevelfunctie en een aantal nodefuncties

Figuur 4.5: Eenvoudige opbouw van een toplevelfunctie

Dit wil zeggen dat de data die bij een event hoort niet meer beschikbaar mag zijn vanaf het moment dat dit event volledig doorheen de graaf gepropageerd is. Wanneer we dit vertalen naar onze implementatie is het voldoende dat een concrete waarde die een eventnode propageert, beschikbaar is tot dat de node(s) die hier rechtstreeks afhankelijk van zijn deze waarde hebben verwerkt. Dit wil dus in principe zeggen dat de data die door een eventnode geproduceerd wordt niet meer beschikbaar hoeft te zijn wanneer de toplevelfunctie volledig is uitgevoerd. Merk op dat behaviors wel continu gedefinieerd zijn en hiervoor zal nog wel globale data nodig zijn. Precies om deze reden is het nuttig om de implementatie te optimaliseren wat betreft geheugenruimte. Naast een optimalisatie in ruimte is er ook een winst in tijd aangezien eventdata na elke uitvoering van de toplevelfunctie opnieuw geïnitialiseerd moet worden. Dit is nodig bij de eenvoudige aanpak van de toplevelfuncties omdat deze eventdata anders nog beschikbaar is bij de volgende uitvoering van een toplevelfunctie. Bij de optimalisatie die we voorstellen is dit niet meer nodig aangezien de data niet meer beschikbaar is vanaf het moment dat de toplevelfunctie uitgevoerd is. Wanneer de data die door de events gegenereerd worden niet langer globaal toegankelijk is, zullen we ervoor moeten zorgen dat de toplevelfunctie instaat voor de overdracht van data tussen opeenvolgende nodes. In codefragment 4.2 tonen we voor de graaf zoals in figuur 4.5a hoe een meer optimale toplevelfunctie eruit ziet. We zien dat elke node een datapakket zal ontvangen met de nodige informatie van de vorige node. Dit is uiteraard nodig voor elke node omdat het een operatie zal uitvoeren op de resultaten van de vorige node(s).

Listing 4.2: Geoptimaliseerde toplevelfunctie

```
void topfunctie_input1()
{
  node_data data1;
  data1 = input1();
  node_data data2;
  data2 = mergefun(data1);
  node_data data3;
  data_node3 = filterfun(data2);
  foldpfun(data3);
}

node_data fun1()  { ... }
node_data fun2(node_data nd)  { ... }
...
```

De reden dat we niet meteen voor deze aanpak gekozen hebben is omdat het LMS-framework geen ondersteuning biedt voor pointers en structs zoals ze voorkomen in de C-taal. Om het mogelijk te maken om meerdere waardes terug te geven van een functie[4] hebben we nood aan structs. Een andere mogelijkheid is om de variabelen in de toplevelfunctie te definiëren en deze vervolgens als pointers door te geven naar de functie. We hebben gekozen voor deze tweede mogelijkheid en pas vanaf dat we LMS hadden uitgebreid om ook een tussentijdse voorstelling te kunnen maken voor pointers was het mogelijk de toplevelfuncties te optimaliseren.

4.2.3 Opbouw van een nodefunctie

Nu we hebben ingevuld hoe een toplevelfunctie eruit kan zien, zijn we klaar om de nodefuncties van een implementatie te voorzien. Hoewel de manier waarop we de toplevelfunctie opbouwen zal bepalen of de nodefunctie de benodigde data globaal ophaalt of via de functieargumenten, heeft dit verder geen impact op de functionaliteit van de nodefunctie. De nodefuncties zullen dus voor beide implementaties van de toplevelfuncties zeer gelijkaardig zijn. Voor de eenvoud zullen we hier bespreken hoe de nodefuncties geïmplementeerd kunnen worden in combinatie met de eenvoudig opgebouwde toplevelfuncties. Dat wil dus zeggen dat de data van de verschillende nodes globaal toegankelijk is. Om te beginnen maken we voor de nodefuncties een onderscheid tussen eventnodes en behaviornodes. Voor elke eventnode genereren we naast de nodefunctie twee globale variabelen. Deze stellen de status van de eventnode voor. De ene variabele geeft aan of de eventnode wel degelijk een waarde propageert[5]. Indien dit het geval is zal de tweede variabele de concrete waarde bevatten die gepropageerd wordt. Indien er geen event gepropageerd wordt, zal de

[4] In de volgende sectie zal blijken dat we inderdaad meerdere waardes moeten teruggeven.

[5] Zoals we zullen zien kan de filter-operatie ervoor zorgen dat een node geen waarde propageert. Daarom moeten we voor elke eventnode aangeven of er al dan niet een waarde zal propageren.

tweede variabele niet gedefiniëerd zijn. Voor elke behaviornode worden er ook twee globale variabele gedefiniëerd. De eerste variabele geeft aan of de node een event propageert waardoor afhankelijke nodes opnieuw kunnen bepalen of ze geëvalueerd moeten worden of niet. Wat verschillend is ten opzichte van de eventnode, is dat de tweede variabele die de behaviorwaarde bevat, wel steeds gedefinieerd zal zijn.

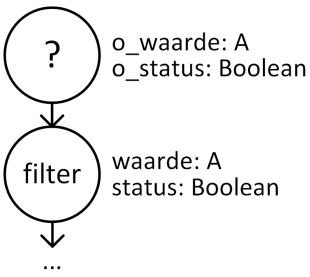
Om dit te verduidelijken zullen we een aantal verschillende types van nodes in detail bekijken. Wat betreft de eventnodes bespreken we de filter-, map- en merge-operatie. Voor de behaviornodes zijn de foldp-, map2- en snapshot-operatie de meest interessante.

Filter-operatie

In codefragment 4.3 is de signatuur van de filter-operatie te zien. In figuur 4.6 zien we hoe deze operatie zich vertaald naar een grafische voorstelling.

Listing 4.3: API filterfunctie

```
trait Event[A] {
  def filter(f: A => Boolean): Event[A]
}
```



Figuur 4.6: Filternode

Bij de filter-operatie moeten we eerst nagaan of de oudernode een event propageert. We kunnen dit veralgemenen voor elke operatie die afhankelijk is van een eventnode. Dat betekent dat al deze operaties eerst zullen moeten nagaan of de eventnode waarvan ze afhankelijk zijn wel degelijk een waarde propageren. We doen dit steeds aan de hand van de *status*variabele van de oudernode. Indien er wel een waarde gepropageerd wordt, zal de predikaatfunctie f worden gebruikt om te bepalen of het event doorgegeven zal worden of niet. Indien de waarde van de oudernode voldoet aan het predikaat, zal de node deze waarde als zijn eigen waarde instellen zodat ook de volgende node deze kan opvragen. Op deze manier propageert de eventwaarde doorheen de graaf. Codefragment 4.4 toont hoe deze functie conceptueel is opgebouwd. We bekijken de filter-operatie als eerste aangezien we op deze manier kunnen aantonen waarom de status-variabele in de eerste plaats nodig is. De filteroperatie kan er immers voor zorgen dat een event tegengehouden wordt. Net

omdat we in de eenvoudig opgebouwde toplevelfunctie alle nodes op het pad evalueren kunnen we er op deze manier voor zorgen dat een node alleen effect heeft als er wel degelijk een event gepropageerd wordt door de oudernode. In de filterfunctie zien we dat deze node geen event zal propageren in het geval dat de oudernode geen event propageert en als de eventwaarde niet voldoet aan het predikaat. Wanneer dit wel het geval is, wordt de status van de node op *waar* gezet en zal de waarde die de node propageert dezelfde zijn als die van de oudernode.

Listing 4.4: Filterfunctie

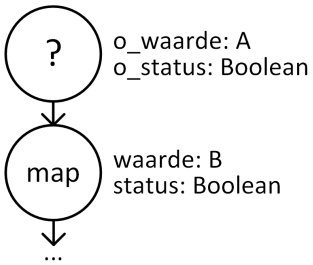
```
void filterfun()
{
  if(o_status){
    if(f(o_waarde)) {
        status = true;
        waarde = o_waarde;
    }
    else { status = false; }
  }
  else{ status = false; }
}
```

Map-operatie

In codefragment 4.5 zien we dat de signatuur van de map-operatie er iets anders uitziet. In figuur 4.7 zien we hoe dit er grafisch uitziet.

Listing 4.5: API mapfunctie

```
trait Event[A] {
  def map[B](f: A => B): Event[B]
}
```



Figuur 4.7: Mapnode

Bij de map-operatie moeten we opnieuw nagaan of de oudernode een event propageert. Zoals gezegd zal dit voor elke node nodig zijn die direct afhankelijk is van een

eventnode. Indien er een waarde gepropageerd wordt, zal de functie f worden gebruikt om de waarde van de ouder om te vormen. Codefragment 4.6 toont hoe deze functie conceptueel is opgebouwd. We zien in dit geval dat de mapnode steeds een waarde zal propageren indien de oudernode dit ook doet in tegenstelling tot de filternode waarbij dit niet steeds het geval is.

Listing 4.6: Mapfunctie

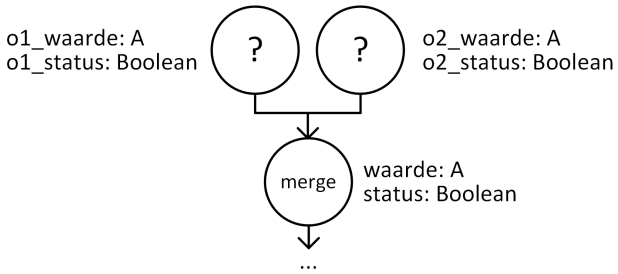
```
void mapfun()
{
  if(o_status){
    status = true;
    waarde = f(o_waarde);
  }
  else{ status = false; }
}
```

Merge-operatie

In codefragment 4.7 zien we de signatuur van de merge-operatie. In figuur 4.8 zien we opnieuw de grafische voorstelling.

Listing 4.7: API mergefunctie

```
trait Event[A] {
  def merge(e:Event[A], f:(A,A)=>A): Event[A]
}
```



Figuur 4.8: Mergenode

Bij de merge-operatie zijn er nu twee oudernodes die al dan niet een waarde kunnen propageren. We hebben in de nodefunctie dus de nodige conditionele logica nodig om uit te maken of er al dan niet eenvoudigweg een waarde doorgegeven kan worden of dat de mergefunctie f gebruikt moet worden om beide waarden te combineren.

Listing 4.8: Mergefunctie

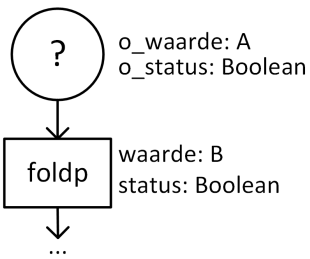
```
void mergefun()
{
  if(o_status){
    status = true;
    if(o1_status && o2_status)
    {
      waarde = f(o1_waarde, o2_waarde);
    }
    else if(o1_status) { waarde = o1_waarde; }
    else if(o2_status) { waarde = o2_waarde; }
    else { status = false; }
  }
  else{ status = false; }
}
```

Foldp-operatie

In codefragment 4.9 en figuur 4.9 zien we voorstelling van de foldp-operatie.

Listing 4.9: API foldpfunctie

```
trait Event[A] {
  def foldp[B](f:(A,B) => B, init:B): Behavior[B]
}
```



Figuur 4.9: Foldpnode

De foldp-operatie is de eerste die geen nieuwe eventstroom zal genereren maar een behavior. Als er een waarde toekomt in de node, zal deze gebruikt worden om de behaviorwaarde te updaten. Dit gebeurt aan de hand van de foldpfunctie f die de eventwaarde en de huidige behaviorwaarde combineert. De status van de node zal ook aangeven of de node al dan niet opnieuw geëvalueerd is.

Listing 4.10: Foldpfunctie

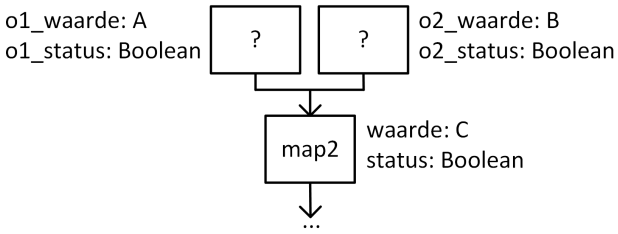
```
void foldpfun()
{
  if(o_status){
    status = true;
    waarde = f(o_waarde, waarde)
  }
}
```

Map2-operatie

De signatuur van de map2-operatie is te zien in codefragment 4.11. In figuur 4.10 stellen we dit opnieuw grafisch voor.

Listing 4.11: API map2functie

```
trait Behavior[A] {
  def map2[B,C](b: Behavior[B], f:(A, B) => C): Behavior[C]
}
```



Figuur 4.10: Map2node

De map2-operatie kan gebruikt worden om twee behaviors om te zetten in een nieuwe behavior. We zullen in de nodefunctie zoals beschreven in fragment 4.12 de map2-operatie uitvoeren wanneer ten minste één van beide oudernodes geëvalueerd werd. Dit wordt aangegeven op basis van de status van de oudernodes.

Listing 4.12: Map2functie

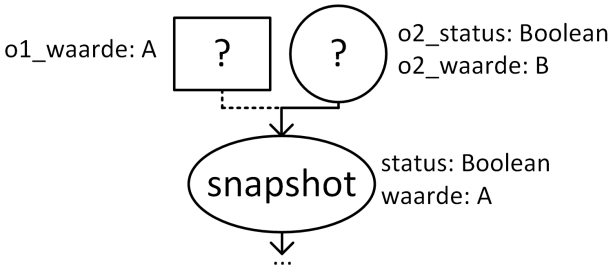
```
void map2fun()
{
  if(o1_status || o2_status) {
    status = true;
    waarde = f(o1_waarde, o2_waarde);
  }
}
```

Snapshot-operatie

Als laatste bekijken we dat de snapshot-operatie zoals in codefragment 4.13. In figuur 4.11 geven we dit grafisch weer.

Listing 4.13: API snapshotfunctie

```
trait Behavior[A] {
  def snapshot[B](e: Event[B]): Event[A]
}
```



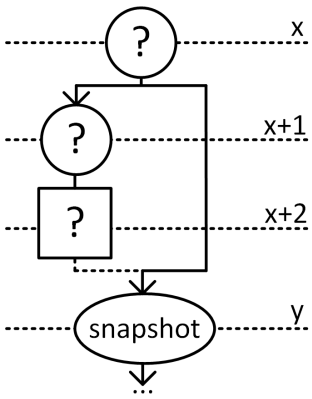
Figuur 4.11: Snapshotnode

De snapshot-operatie kan gebruikt worden om een behavior te *samplen*. Telkens wanneer de eventnode *o2* een event propageert, zal de snapshotnode de waarde van behaviornode *o1* propageren. We zullen in de nodefunctie zoals beschreven in codefragment 4.14 alleen een waarde propageren als er een event ontvangen wordt van *o2*. Dit maakt de snapshotnode de opmerkelijkste nodefunctie en ook daarom is in figuur 4.11 de behaviornode verbonden in met een stippellijn. Het is niet de bedoeling dat deze snapshotnode een event propageert indien de behaviorwaarde wijzigt. Alleen bij een event via *o2* wordt de behaviorwaarde *gesampled*. Dit is opnieuw mogelijk aangezien de behaviorwaarde constant gedefiniëerd is.

Listing 4.14: snapshotfunctie

```
void snapshotfun()
{
  if(o2_status){
    status = true;
    waarde = o1_waarde;
  }
  else { status = false; }
}
```

Voor de snapshotnode willen we dan ook alleen dat deze node aanwezig is in de toplevelfunctie waarvan node *o2* deel van uitmaakt. Op deze manier zorgen we ervoor dat ze alleen reageert op events van *o2*. Wat betreft het niveau van deze node moeten we echter wel opletten. Het is niet zo omdat de snapshotnode alleen

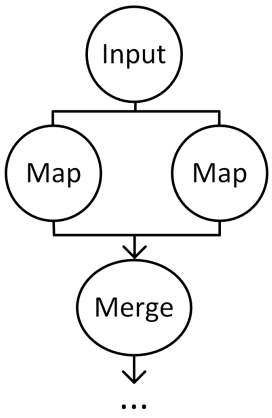


Figuur 4.12: Niveaubepaling snapshotnode

aanwezig is in de toplevelfunctie waarin node *o2* wordt opgeroepen, dat we het niveau simpelweg op deze node kunnen baseren. We bekijken in figuur 4.12 een scenario waarin dit tot een vervelende misinterpretatie kan leiden. Wanneer we het niveau alleen baseren op het toekomende event, zou niveau y gelijk moeten zijn aan x+1. Dit zou willen zeggen dat de behaviorwaarde nog niet vernieuwd is aangezien deze node pas op niveau x+2, en dus later, geëvalueerd zal worden. De snapshotnode zou dus een oude behaviorwaarde samplen en propageren. Omdat de snapshotnode van beide nodes afhankelijk is, geldt de standaardregel dat het niveau y gelijk moet zijn aan $max(o1\_niveau, o2\_niveau)$. Op die manier garanderen we dat steeds de meest recente behaviorwaarde zal propageren.

4.2.4 Nut van toplevelfuncties

Zoals we aan het begin van deze sectie al hebben aangegeven gaan we hier dieper in op het nut van de toplevelfuncties. De lezer kan zich al hebben afgevraagd waarom de verschillende nodefuncties elkaar niet simpelweg oproepen. De nodefunctie van de inputnode zou dan de nodefuncties kunnen oproepen van diens kindernodes. Wanneer er geen vertakkingen zijn zoals in de graaf in figuur 4.5a is dit perfect mogelijk. Er zijn echter situaties waarbij dit minder triviaal is. Wanneer er verschillende vertakkingen zijn in de graaf, zoals in figuur 4.13, die afkomstig zijn van dezelfde inputnode kunnen de nodefuncties elkaar niet meer zo eenvoudigweg oproepen zonder extra logica in te bouwen. We zien in de figuur dat we bij de mergefunctie gebruik zullen moeten maken van de functie f die we bij de merge-operatie gedefinieerd hebben om de toekomende events te combineren. Wanneer de nodefuncties simpelweg de volgende nodes zouden aanroepen, is het niet mogelijk om de eventwaardes te combineren omdat ze niet tegelijkertijd arriveren in de node. Een oplossing daarvoor kan zijn om in de mergenode extra logica in te bouwen die het event niet verder zal propageren tot ook de waarde van de tweede tak is toegekomen. Deze aanpak wordt onder andere gebruikt in Elm [3]. Zoals besproken lossen we in ons FRP-framework dit probleem op door het gebruik van toplevelfuncties. Deze functie geeft ons de



Figuur 4.13: Probleemsituatie zonder toplevelfunctie

flexibiliteit om te bepalen welke we node we op welk moment willen evalueren. Dit maakt het mogelijk om de mergefunctie pas te evalueren vanaf het moment dat beide takken die in deze node samenkomen reeds geëvalueerd zijn. Merk op dat we in onze implementatie de nodes in de graaf, vertrekkende van de inputnode, in *broad-first* volgorde evalueren. Dit leunt conceptueel het dichtste aan bij de manier waarop we ons kunnen voorstellen dat een event doorheen de graaf zal propageren.

4.2.5 Overzicht en uitdrukking in LMS

In deze sectie gaven we voorlopig al een overzicht van de functies die we uiteindelijk in de gegenereerde code willen zien terugkomen. De bedoeling daarvan is om de FRP-graaf zo veel mogelijk weg te compileren. We doen dit aan de hand van toplevelfuncties. We stellen één toplevelfunctie op per inputnode waarin we alle operaties verpakken die nodig zijn om het systeem te laten reageren op een event van die inputnode. In een toplevelfunctie worden verschillende nodefuncties opgeroepen om het event af te handelen en eventueel de toestand van het systeem aan te passen.

Nu we weten wat we precies willen genereren, moeten we deze functies ook registreren in het LMS-framework als tussentijdse voorstelling. Om dit te bewerkstelligen gebruiken we de technieken die aangereikt worden door LMS zoals we ze hebben ingeleid in 2.3.2. Daarin zagen we hoe we gebruik kunnen maken van *staging* om een onderscheid te maken tussen code die *nu* en code die *later* geëvalueerd moet worden. Het LMS-framework maakt dit onderscheid op basis van type waardoor een willekeurig codefragment relatief eenvoudig *gestaged* kan worden door de types aan te passen. Omdat we in de vorige secties de toplevelfuncties en nodefuncties hebben opgesteld kunnen we deze gebruiken om ze om te vormen naar *staged* of uitgestelde functies. De functies worden uitgesteld om uiteindelijk pas op het ingebed systeem uitgevoerd te worden via de gegenereerde code.

Aangezien de hosttaal van LMS Scala is, zullen deze functies uiteraard nog steeds op-

gesteld worden in Scala. Pas in de volgende sectie tonen we aan hoe we de tussentijdse voorstelling dan ook kunnen omvormen naar bijvoorbeeld C-code. Om dit allemaal wat concreter te maken zullen we laten zien hoe de eenvoudige toplevelfuncties en de nodefunctie van de map-operatie omgevormd kan worden naar zijn uitgestelde variant. In codefragment 4.15 zien we hoe de eenvoudige toplevelfunctie voor *input1* uit figuur 4.3 eruit ziet. Daarnaast toont dit fragment ook hoe de filterfunctie eruit ziet in zijn uitgestelde voorstelling die alleen eventwaardes propageert die kleiner zijn dan 10. We zien dat beide functies zeer sterk gelijken op de manier waarop we ze in de vorige secties hebben voorgesteld. Het enige verschil is uiteraard dat ze vertaald zijn naar Scala en dat we gebruik hebben gemaakt van het Rep-type om de functies te *stagen*.

Listing 4.15: Registeren van de functies in het LMS-framework

```
1  def topfunctie_input1: Rep[(Unit)=>Unit] = {
2    input1()
3    mapfun()
4    mergefun()
5    filterfun()
6    foldpfun()
7  }
8
9  def filterfun(): Rep[(Unit)=>Unit] = {
10   val ouderwaarde: Rep[Int] = o_waarde
11   val ouderstatus: Rep[Boolean] = o_status
12
13   if(ouderstatus)
14   {
15     if( f(ouderwaarde) ) {
16       status = true;
17       waarde = o_waarde;
18     }
19     else { status = false; }
20   }
21   else{ status = false; }
22 }
```

Om te beginnen zien we dat de toplevelfunctie van het type Rep[(Unit)=>Unit] is. Dit betekent dat het een representatie van een functie zonder argumenten voorstelt die ook geen resultaat teruggeeft. Ook de filterfunctie is van dit type. Verder zien we in de filterfunctie dat zowel de statusvariabele als de variabele die de waarde aangeeft niet meer van type Int, respectievelijk Boolean zijn maar van type Rep[Int] en Rep[Boolean]. De variabele die de waarde aangeeft is ook niet langer van type Int maar Rep[Int]. Een belangrijk detail in deze code is de functie *f*. In codefragment 4.3 hebben we gezien dat deze functie een argument van type *A* verwacht in plaats van type Rep[A]. De volledige API zoals we hem hebben voorgesteld in sectie 3.2 ziet

er in werkelijkheid anders uit. In bijlage C kan u de volledige FRP API terugvinden zoals deze er werkelijk uitziet. De eindegebruiker van de EDSL hoeft echter geen rekening te houden met de interne werking van de codegenerator en kan de API simpelweg interpreteren zoals is weergegeven in 3.2. Het LMS-framework biedt de nodige impliciete conversies aan die ervoor zorgen dat alle expressies die de gebruiker van de EDSL gebruikt omgezet worden naar dezelfde expressie van het Rep-type indien nodig.

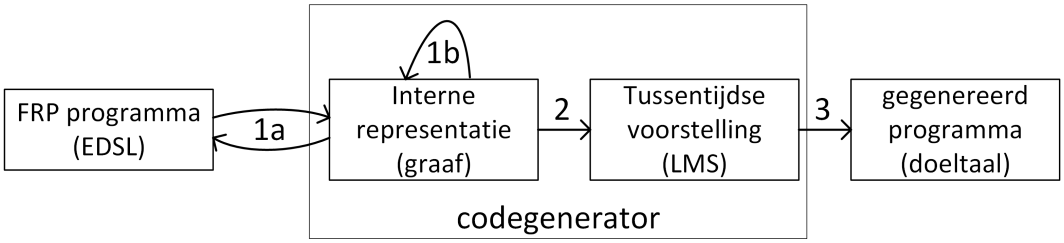
4.3 Van tussentijdse voorstelling naar gegenereerde code

Aan het einde van de vorige sectie zagen we dat de functies die uitgesteld worden naar de volgende stage in principe in de codegenerator opgesteld worden in Scala. Door het gebruik van de Rep-types die aangeboden worden door het LMS-framework genereren we in de vorige stap nog geen code maar wordt de code die we willen genereren voorgesteld in een tussentijdse voorstelling. In principe is dit zeer gelijkend aan een compiler die een *Abstract Syntax Tree* of AST opstelt van het programma dat gegenereerd zal worden. In LMS wordt dus ook zo een tussentijdse voorstelling opgebouwd. Elke expressie wordt vastgelegd in de vorm van een definitie van het type *Def[T]*. Zonder al te veel in detail te treden ziet bijvoorbeeld de definitie van de declaratie van een variabele er als volgt uit:

```
case class NewVarDecl[T]() extends Def[Variable[T]]
```

De tussentijdse voorstelling of AST van het programma bestaat uit een graaf van dit soort definities. Op basis van deze voorstelling zal het LMS-framework een aantal optimalisaties kunnen uitvoeren zoals eliminatie van onbereikbare code en minimalisatie van codeduplicatie. Uiteindelijk zal in de laatste stap van het proces van codegeneratie deze tussentijdse voorstelling omgevormd worden naar code die het resultaat vormt van onze codegenerator. Mede door gebruik te maken van de tussentijdse voorstelling was het mogelijk om de keuze van de doeltaal uit te stellen tot in deze stap. Dit maakt de codegenerator heel erg flexibel om nieuwe doeltalen toe te voegen.

Het LMS-framework neemt deze stap voor zijn rekening. Zoals aangegeven hebben we alleen de codegeneratie voor pointers moeten toevoegen om ook de geoptimaliseerde toplevelfuncties uit te kunnen drukken. Het moet echter wel gezegd worden dat de gehele generatiestap voor C-code herzien moest worden omdat de originele generator in LMS *C++*-code produceerde. Belangrijk om te onthouden van deze stap is dat er nu pas bepaald wordt in welke taal de code gegenereerd zal worden. Dit zorgt ervoor dat het genereren van specifieke code volledig onafhankelijk is van de voorgaande stappen. Wanneer we in het volgende hoofdstuk aangeven hoe een dialect van C ondersteund kan worden zal ook alleen deze stap aangepast moeten worden. Deze aanpak zorgt er dus voor dat het framework erg flexibel is om nieuwe doeltalen toe te voegen.



Figuur 4.14: Overzicht van de codegenerator

4.4 Overzicht

In dit hoofdstuk hebben we tot in detail gezien hoe de verschillende stappen van het codegeneratieproces eruit zien. Het lijkt ons nu een goed moment om even een stap terug te nemen en het totaal nog even in de vorm van een overzicht te presenteren. In figuur 4.14 zien we de verschillende stappen die we in dit hoofdstuk hebben toegelicht.

Het proces start met de invoer van het FRP-programma zoals voorgesteld in hoofdstuk 3. De eerste stap die we ondernemen is het omvormen van dit FRP-programma naar een interne representatie ervan. Het programma zal er dan uitzien als een graaf waarin de relaties tussen de verschillende eventstromen en behaviors worden vastgelegd. Zoals aangegeven in figuur 4.14 is deze stap opgedeeld in twee delen. In het eerste deel (1a) analyseren we het FRP-programma lijn per lijn. Na elke lijn wordt de graaf verder opgebouwd. Bij het opstellen van deze graaf is het belangrijk om de nodes in te delen in verschillende niveaus. Dit is nodig om uiteindelijk een topologische volgorde op te kunnen stellen voor de nodes om *glitches* te vermijden in het uiteindelijke programma. Vooraleer we de volgende stap kunnen aanvatten moet de graaf vervolledigd worden. Aan het einde van stap 1a bevatten de verschillende nodes in de graaf immers alleen informatie over de nodes die zich boven de node in kwestie bevinden. In stap 1b wordt de graaf doorlopen van onder naar boven om zo alle nodes te voorzien van aanvullende informatie over de kindnodes.

De tweede stap zal de graaf die alle informatie over het FRP-programma bevat omvormen naar een interne voorstelling in LMS. We trachten zoveel mogelijk van de FRP-graaf weg te compileren om het programma dat we uiteindelijk zullen genereren zo optimaal mogelijk te maken. Door de FRP-graaf als statisch te beschouwen slagen we erin de graaf weg te compileren door alle relaties tussen events en behaviors permanent vast te leggen. We stellen daarvoor per inputnode een toplevelfunctie op die alle operaties bevat om een event van deze inputstroom volledig af te handelen. In de FRP-graaf stellen de nodes de verschillende FRP-operaties voor. Deze operaties worden vastgelegd in nodefuncties. In de toplevelfunctie worden de gepaste nodefuncties opgeroepen die nodig zijn om een bepaald inputevent te verwerken. Alle functies die we opstellen om de verschillende soorten events die opgevangen worden door het programma af te handelen worden in het LMS-framework geregistreerd in een tussentijdse voorstelling.

In de derde en laatste stap wordt de tussentijdse voorstelling omgevormd naar het programma in de doeltaal. Het voordeel van de tussentijdse voorstelling is enerzijds dat er nog heel wat optimalisatie op kan worden uitgevoerd nadat deze voorstelling volledig is voor het te genereren programma. Anderzijds wordt hiermee uitgesteld welke doeltaal we willen gebruiken voor het gegenereerde programma. Dit betekent dat alleen deze stap aangepast moet worden in het geval we een nieuwe doeltaal willen ondersteunen.

In dit hoofdstuk hebben we de basisimplementatie van de EDSL voorgesteld om het toe te laten FRP-programma's op te bouwen. Zoals we in de inleiding en in sectie 2.4 hebben aangegeven zou het opstellen van programma's voor het Sancus framework ook eenvoudiger worden moesten er FRP-abstracties voor handen zijn. In het volgende hoofdstuk stellen we voor hoe we de EDSL kunnen uitgebreid om het Sancus framework te ondersteunen.

Hoofdstuk 5

Uitgebreide implementatie: Sancus

In het vorige hoofdstuk werd de basisimplementatie van het FRP-framework voorgesteld. Om nu concreet een werkend programma te kunnen genereren voor een ingebed systeem zijn er echter nog een aantal uitbreidingen nodig. Daarom introduceren we in dit hoofdstuk een aantal interfaces voor de hardware van het ingebed systeem waardoor er eventstromen aangemaakt kunnen worden. Daar stopt het echter niet bij aangezien we ons richten op het Sancus framework. We introduceren ook nog een aantal andere constructies in onze EDSL om ook daarvoor specifieke ondersteuning te bieden. Daarbij zal het creëren en koppelen van modules centraal staan. In het tweede deel van dit hoofdstuk gaan we dieper in op de uitbreiding van de concrete codegeneratie ofwel stap drie uit figuur 4.14. Aan het einde van dit hoofdstuk zullen alle puzzelstukjes aanwezig zijn in de EDSL om een werkend programma te kunnen samenstellen dat gebruik maakt van het Sancus framework en dat ook uitgevoerd kan worden op een ingebed systeem.

5.1 EDSL uitbreiding

Om te beginnen stellen we de verschillende uitbreidingen in de EDSL voor om een programma op te bouwen voor Sancus. Herinner u uit sectie 2.4 dat een programma verschillende modules kan bevatten. Ondersteuning voor modules bij het opstellen van het FRP-programma zal dan ook de belangrijkste toevoeging zijn. Daarnaast zijn ook een aantal concrete hardwarebronnen voor events nodig om een concrete applicatie te kunnen bouwen.

5.1.1 Modules

De EDSL kan uitgebreid worden met modules. We stellen eerst voor hoe deze uitbreidingen eruitzien voor de gebruiker van de EDSL om een applicatie op te bouwen met modules. We beginnen met de manier waarop een applicatie er op het hoogste niveau uit zal zien in codefragment 5.1. In de methode *createApplication*

zal de volledige applicatie samengesteld kunnen worden. De codegenerator zal het resultaat van deze methode gebruiken om de code te genereren. Zoals we zien verwacht de codegenerator van deze methode een lijst van modules die voorkomen in de applicatie. Het is belangrijk om op te merken dat deze modules van elkaar en van de buitenwereld zijn afgeschermd. Daarmee bedoelen we dat variabelen en functies die binnen een module gedefiniëerd worden, alleen van daarbinnen toegankelijk zijn.

Listing 5.1: Creatie van een applicatie

```
def createApplication: List[Module[_]] = {
  ...
}
```

De volgende stap is het opbouwen van de verschillende modules. Om een module te kunnen samenstellen bieden we in onze EDSL de methode createModule aan met de volgende signatuur:

```
def createModule[A]
  (graafFunctie: (ModuleName)=>Option[OutputEvent[A]] )
  : Module[A]
```

Uit de signatuur kunnen we afleiden dat deze methode een functie verwacht waarin we de FRP-graaf voor deze bepaalde module definiëren. De functie neemt als invoerargument de modulenaam die aangeleverd zal worden door het framework. In de *body* van de functie kunnen we vervolgens een FRP-graaf opstellen via de operaties die we hebben gezien in hoofdstuk 3. Op deze manier integreren we alle concepten uit hoofdstuk 4 in de applicatie. Aan het einde van de functie verwacht de codegeneratie een specificatie van wat de module uitvoert. In voorbeeld 2.1 hebben we gezien dat een module ook een uitgang kan definiëren, zoals bijvoorbeeld *O1*. We zien aan het resultaat type van de graaffunctie dat deze uitvoer echter optioneel is. We ondersteunen momenteel slechts één uitgang voor een module en lichten een uitbreiding toe in sectie 6.3. In codevoorbeeld 5.2 zien we concreet hoe de opbouw van een module er kan uitzien.

Listing 5.2: Opstellen van een modules in de EDSL

```
val mod1: Module[Int] =
  createModule { implicit n: ModuleName =>
    val input1 = InputEvent()
    val const = input1.constant(1)
    val counter: Behavior[Int]
      = const.foldp( (x, state) => state + x, 0)
    val changing = counter.changes()
    val module_output: OutputEvent[Int] = out(changing)
    Some(module_output)
  }
```

Op basis van type-inferentie zal de codegenerator voor dit voorbeeld kunnen bepalen dat de module *mod1* een eventstroom van integers genereert. Dit is een belangrijke vaststelling die we zullen gebruiken om verschillende modules op een type-veilige manier aan elkaar te koppelen. In sectie 2.4 gaven we al aan dat modules niet getypeerd aan elkaar gekoppeld worden in het Sancus framework. Hierdoor is het de verantwoordelijkheid van de programmeur om ervoor te zorgen dat de eventstroom die een module exporteert van hetzelfde type is dan de invoer die een gekoppelde module verwacht. Om dit te verbeteren bieden we in onze EDSL een manier aan om de modules toch *type safe* te verbinden.

5.1.2 Type safety tussen modules

Om ervoor te zorgen dat modules op een type-veilige manier gekoppeld worden, bieden we een manier aan om deze met elkaar te verbinden. Zoals we in codevoorbeeld 5.2 al hebben gezien bij het creëren van een module is het mogelijk om aan te geven welke eventstroom geëxporteerd zal worden. Hiermee bedoelen we dat de events van deze uitvoerstroom de module kunnen verlaten en toegankelijk zijn buiten de module in tegenstelling tot events binnen de module. Dit is nodig als we deze events willen gebruiken als invoer voor een andere module. De module als resultaat van de methode *createModule* bevat het type van de uitvoerstroom die voor deze module gedefiniëerd is. Dit type zal gebruikt kunnen worden in een andere module om het type van de gekoppelde eventstroom te bepalen.

Om dit te bewerkstelligen definiëren we de volgende nieuwe inputnode:

```
def ExternalEvent[A](oe: OutputEvent[A]): Event[A]
```

Deze inputnode registreert niet alleen welke modules er met elkaar gekoppeld moeten worden door het framework, maar het genereert daarnaast ook een eventstroom van het correcte type. In codefragment 5.3 geven we aan hoe we een tweede module kunnen samenstellen die gekoppeld wordt met de module uit fragment 5.2. Het is belangrijk om op te merken dat type-inferentie kan plaatsvinden door het type van de uitvoerstroom van *module1*. Op deze manier zijn we erin geslaagd om in onze EDSL modules op een type-veilige manier te koppelen.

Listing 5.3: Koppeling van modules op een type veilige manier

```
val mod2 =
  createModule { implicit n: ModuleName =>
    val input: Event[Int] = ExternalEvent(mod1.output)
    ...
  }
```

5.1.3 Interface met hardware van het ingebed systeem

Nu we weten hoe we verschillende modules met elkaar kunnen koppelen, moeten we nog toelichten hoe deze verschillende modules gekoppeld worden met de hardware van het ingebed systeem. We hebben tenslotte een bron nodig die events kan genereren en in hoofdstuk 3 spraken we steeds van de abstracte eventstroom *InputEvent*. We definiëren daarom zowel een inputnode voor hardwareknoppen als voor een interne timer die nieuwe inputevents zullen introduceren in het systeem. Daarnaast stellen we ook een nieuwe module voor die toelaat om resultaten op een scherm te tonen. We hebben ervoor gekozen om dit aan te bieden in de vorm van een module en niet in de vorm van een nieuwe FRP-operatie. Zo worden neveneffecten buiten het FRP-framework gehouden en blijven de FRP-abstracties pure expressies. Zoals we verder in dit hoofdstuk zullen zien zijn deze hardwareprimitieven in de EDSL voorlopig voldoende om een demoapplicatie te bouwen voor een ingebed systeem.

Om te beginnen introduceren we een ButtonEvent dat gebruikt kan worden om inputevents te genereren afkomstig van een hardwareknop. Deze operatie produceert een eventstroom die integers zal teruggeven die aangeven of de knop al dan niet net is ingedrukt of losgelaten. De operatie TimerEvent genereert een eventstroom waarop in een vast interval steeds een nieuw event toegevoegd wordt aan de eventstroom.

```
def ButtonEvent(b: Button): Event[Int]
def TimerEvent(): Event[Unit]
```

Interessanter is het toevoegen van een nieuwe module die gebruikt kan worden om iets te tonen op een scherm dat is aangesloten op het ingebed systeem. Deze module genereert dus geen uitvoer maar kan gebruikt worden als eindmodule.

```
def createLCDModule
 (graafFun:(ModuleName) => List[(Behavior[Int], Int, Int)])
 : Module[Nothing]
```

We zien bij deze module dat we opnieuw een functie kunnen specifiëren waarin we een willekeurige FRP-graaf in opbouwen. Het verschil met de module van voordien is dat deze module geen uitvoerstroom heeft maar bepaalt hoe het scherm van het systeem resultaten kan tonen. We moeten bij deze module bepalen welke behaviorwaarden op het scherm getoond worden. Voor het prototype beperken we ons tot behaviorwaardes van type integer. We gaan nog even in op deze beslissing in sectie 6.3. We kunnen dit doen door een lijst op te geven van behaviorwaardes dat we willen tonen. Bij elke behaviorwaarde kunnen we verder nog de rij en kolom mee, om aan te geven op welke positie de waarde getoond moet worden.

Vanuit een gebruikersstandpunt zorgen deze uitbreidingen ervoor dat we via de EDSL nuttige FRP-programma's kunnen opbouwen. Om dit nu ook correct te vertalen naar een programma in de doeltaal zijn er in de implementatie van het framework ook nog een aantal uitbreidingen nodig. Herinner u ook dat er een specifieke syntax

nodig is om een Sancus programma's te schrijven. In de volgende sectie gaan we daar in detail op in.

5.2 Uitbreiding codegeneratie

In het vorige hoofdstuk is gebleken in sectie 4.3 dat er slechts één stap moet uitgebreid worden van het LMS-framework om een nieuwe doeltaal te ondersteunen. Dit betekent dat we LMS uitbreiden om de tussentijdse voorstelling op een andere manier om te zetten naar code. Om dit te verduidelijken geven we een concreet voorbeeld voor de codegeneratie voor de declaratie van een variabele. In de tussentijdse voorstelling zal zo een declaratie voorgesteld worden door de volgende node:

```
case class NewVarDecl[T]() extends Def[Variable[T]]
```

Elke node in de tussentijdse voorstelling is van het type Def[T] wat staat voor de definitie van een expressie. Het LMS-framework zal ervoor zorgen dat wanneer de volledige tussentijdse voorstelling van het programma is opgesteld, elke definitie volgens een bepaalde regel wordt omgevormd naar gegenereerde code. In de originele C-codegenerator wordt in stap 3 uit figuur 4.14 een *NewVarDecl*-node bijvoorbeeld naar het volgende stukje code omgezet:

```
int x42;
```

Door de definitie van deze vertaling aan te passen slagen we erin om een declaratie om te vormen naar de syntax die nodig is in het Sancus framework:

```
SM_DATA(moduleName) int x42;
```

Merk daarbij op dat dit niets verandert aan het programma van de gebruiker. De modulenaam zal aangeleverd worden door het framework en wordt bepaald aan de hand van de context van deze expressie, *i.e.* de module waarin deze voorkomt. Deze aanpak geeft de flexibiliteit van het LMS-framework weer. Door ook de definitie voor functies aan te passen kunnen we bijvoorbeeld de macro *SM_FUNC* genereren. Merk op dat we ook een aantal nieuwe definities hebben toegevoegd aan de interne voorstelling van LMS om een aantal specifieke expressies voor Sancus te ondersteunen. Zo hebben we naast gewone functies ook inputfuncties die fungeren als *entry points* voor een module en daarom een specifieke syntax vereisen, namelijk *SM_INPUT*.

5.2.1 Interface tussen hardware en FRP-modules

Zoals aangegeven in de vorige sectie hebben we een aantal invoerstromen gedefiniëerd die de hardware van het systeem kunnen gebruiken om events te genereren. Deze events worden door het ingebed systeem gegenereerd en zitten dus van nature buiten alle modules. Wat ons nu nog rest is om ervoor te zorgen dat de eventdispatching correct gebeurt. Dit betekent dat de eventhandler voor de verschillende hardware events verwijst naar de correcte modules die deze inputevents wensen te ontvangen.

De eventhandler heeft enkel toegang tot de *entry points* die aangeboden worden door de module. De keuze voor toplevelfuncties in sectie 4.2 toont een mooie interface naar het FRP-programma in de module en vormt een natuurlijke keuze voor de toegangspunten voor de module. Zoals we daar besproken hebben genereren we een toplevelfunctie per inputnode. Dat wil zeggen dat er een mapping bestaat tussen een specifieke inputevent en de toplevelfunctie die dit inputevent zal afhandelen. We zullen er dus voor zorgen dat alle toplevelfuncties van de module worden aangeboden in de vorm van een toegangspunt. Op deze manier kan de eventhandler van een bepaald hardware-event (buiten de modules) ingesteld worden om de toplevelfunctie op te roepen die zich heeft geregistreerd voor dit specifieke inputevent.

5.2.2 Eventloop

Tenslotte wordt er nog een mainfunctie gegenereerd buiten alle modules. In deze functie kan eerst het systeem en daarna de verschillende modules geïnitialiseerd worden. Naast de initialisatie van het framework die nodig is voor de modules moeten we zelf ook nog een initialisatie van de data in de modules voorzien. Variabelen die in een module gedeclareerd worden, krijgen standaard de waarde 0. Indien een behaviorwaarde een andere startwaarde dan 0 moet bevatten is het nodig om deze te initialiseren alvorens events verwerkt worden. Vervolgens komt het systeem in een eventloop terecht die de verschillende hardware-events de kans geeft nieuw opgetreden events door te geven aan de geregistreerde modules. In de volgende sectie komen we terug op de voorbeeldapplicatie van hoofdstuk 3 als concreet voorbeeld voor deze nieuwe toevoegingen.

5.3 Voorbeeldapplicatie

In de vorige sectie hebben we gezien welke extra constructies er nodig zijn in de EDSL om een concreet programma op te bouwen voor Sancus. Belangrijk daarin zijn het gebruik van modules enerzijds en specifieke randapparatuur anderzijds. Daarom lijkt het ons nu een goed moment om dit te bundelen in een voorbeeld. We grijpen daarvoor terug naar het voorbeeld uit hoofdstuk 3 waarin we een eenvoudige applicatie opbouwden. Om dit hoofdstuk te besluiten tonen we hoe we deze applicatie volledig voorstellen in de EDSL. Daarnaast kunnen we nu ook laten zien hoe het gegenereerde programma voor het Sancus-framework eruit ziet. In codefragment 5.4 zien we hoe de volledige applicatie eruit ziet.

Listing 5.4: Voorbeeldapplicatie

```
1 override def createApplication: List[Module[_]] = {
2
3   val mod1 = createModule { implicit n: ModuleName =>
4     val input1 = ButtonEvent(button1)
5     val input2 = ButtonEvent(button2)
6     val negate2 = input2.map( (i) => 0-i)
7     val merged = input1.merge(negate2, (x,y) => x + y)
```

```
8     val filtered = merged.filter( x => Math.abs(x) < 10)
9     val counter = filtered.foldp( (x,state) => state + x, 0)
10    val changing = counter.changes()
11    val outputevent = out(changing)
12    Some(outputevent)
13   }
14
15   val mod2 = createLCDModule { implicit n: ModuleName =>
16    val input: Event[Int] = ExternalEvent(mod1.output)
17    val display: Behavior[Int] = input.startsWith(0)
18    (display, (0,0) )::Nil
19   }
20
21   mod1::mod2::Nil
22
23  }
```

Met de toevoegingen van dit hoofdstuk kunnen we nu nog kort even bekijken hoe de gegenereerde code er op een hoog niveau uitziet voor deze voorbeeldapplicatie. In codefragment 5.5 zien we een fractie van de gegenereerde code. We hebben ervoor gekozen om alle functiedetails weg te laten om het overzicht te kunnen behouden in de gegenereerde code. We zien voor de eerste module dat voor *topfunctie_input1* alleen de oproepen naar de nodefuncties overblijven. Merk op dat de gelijkenis met de toplevelfunctie uit figuur 4.5b zeer sprekend is. Het enige verschil is wel dat we nu de geoptimaliseerde toplevelfunctie gebruiken. Verder zien we ook een mainfunctie waarin de eventhandler bij een inputevent de correcte toplevelfunctie kan oproepen, in dit geval *topfunctie_input1*. In bijlage E kan u de volledige gegenereerde code terugvinden, weliswaar nog steeds zonder functiedetails.

Listing 5.5: Voorbeeldapplicatie

```
1  SM_DATA(mod1) int mod1_initialiased;
2  SM_DATA(mod1) int counter;
3
4  SM_FUNC(mod1) void init_mod1 () {...}
5  SM_FUNC(mod1) void input_button1 (...) { ... }
6  SM_FUNC(mod1) void mergefun (...) { ... }
7  SM_FUNC(mod1) void filterfun (...) { ... }
8  SM_FUNC(mod1) void foldpfun (...) { ... }
9  SM_FUNC(mod1) void changesfun (...) { ... }
10 SM_FUNC(mod1) void outputfun (...) { ... }
11
12 SM_INPUT(mod1,topfunctie_input1,data,len) {
13    init_mod1();
```

```
14    bool status = false;
15    int waarde;
16    input_button1(data,len,&status,&waarde);
17    mergefun(...);
18    foldpfun(...);
19    filterfun(...);
20    changesfun(...);
21    outputfun(...);
22  }
23  ...
24  static void button1_handler (int pressed) {
25    if(pressed) topfunctie_input1(...);
26  }
27  ...
28  int main() {
29    init_board();
30    deploy_modules();
31    buttons_register_callback(Button1,button1_handler);
32    ...
33    while(1) {
34      buttons_handle_events();
35    }
36  }
```

In het volgende hoofdstuk worden de resultaten van de EDSL besproken. We maken daarbij ook een vergelijking met gerelateerde onderzoeksprojecten en formuleren aan het einde daarvan een conclusie voor deze resultaten.

Hoofdstuk 6

Resultaten en future work

In de vorige twee hoofdstukken hebben we de implementatie van onze FRP EDSL voorgesteld. In hoofdstuk 4 bespraken we hoe de basisimplementatie voor het FRP-framework eruit ziet. Deze implementatie laat toe een FRP-programma op te bouwen en vormt de basis van dit werk. Om deze implementatie echter te kunnen gebruiken op een ingebed systeem zijn er nog extra voorzieningen nodig om events die gegenereerd worden door de hardware aan te bieden via de FRP EDSL. In hoofdstuk 5 hebben we dan ook uitgelegd hoe een aantal nieuwe operaties daarvoor kunnen zorgen. Omdat we ons richten op een zeer specifiek framework, namelijk Sancus, komen er naast deze concrete inputstromen ook nog een aantal nieuwe constructies bij in de EDSL. Deze kunnen we gebruiken om modules te maken en zo een FRP-programma te bouwen met modules die onderling van elkaar zijn afgeschermd.

In dit hoofdstuk nemen we de tijd om eerst onze implementatie te evalueren. We overlopen eerst de belangrijkste resultaten van onze codegenerator en het ontwikkelde FRP-framework. Daarna maken we een vergelijking met twee vergelijkbare werken, namelijk FRP-Arduino [1] en Flask [7]. Aan het einde van dit hoofdstuk formuleren we welke stappen er nog kunnen gezet moeten worden om de EDSL te vervolledigen. We sluiten het hoofdstuk af met een overzicht waarin we onze belangrijkste bevinden nog even op een rijtje zetten.

6.1 Resultaten

In deze sectie overlopen we opnieuw hoe de belangrijkste problemen in het FRP-framework worden opgelost en bespreken we de eigenschappen van onze FRP EDSL. Daarna bekijken we hoe de gegenereerde code zich verhoudt tot het ingevoerde FRP-programma en trachten we dit in kaart te brengen.

6.1.1 Glitchpreventie en uitbreidbaarheid door toplevelfuncties

Om te beginnen bespreken we hoe de verschillende nodes in de FRP-graaf geëvalueerd worden. Zoals voordien besproken gebruiken we een toplevelfunctie per inputnode.

In deze toplevelfunctie bepalen we de volgorde waarin de verschillende nodefuncties uitgevoerd worden. Dit zorgt ervoor dat we een *broad-first* aanpak kunnen hanteren bij het evalueren van de nodes. Dit leunt het dichtst aan bij het mentale model dat we hebben bij een event dat door de graaf van boven naar beneden propageert. Deze keuze zorgt ervoor dat alle nodes op hetzelfde niveau uitgevoerd worden alvorens er nodes van het volgende niveau worden uitgevoerd. Dit zorgt ervoor dat nodes die afhankelijk zijn van meerdere nodes, zoals de mergenode, kunnen beschikken over invoerwaardes die up-to-date zijn. Op deze manier zorgen we voor glitchpreventie, een belangrijke eigenschap van het FRP-framework.

Om te beginnen zorgt de keuze voor een toplevelfunctie voor een extra functie die we moeten genereren tegenover een FRP-framework waarin de verschillende nodefuncties elkaar oproepen. Het voordeel is wel dat alle nodefuncties eenvoudiger blijven. Er is geen logica nodig in de nodes om verschillende toekomende stromen te synchroniseren. Dit is nodig als events *depth-first* doorheen de graaf propageren. Een erg belangrijke implicatie van onze keuze is ook dat het framework erg eenvoudig kan uitgebreid worden met nieuwe FRP-operaties. Omdat elke node functioneel gezien op zichzelf staat en niet met de functionaliteit van andere nodes rekening moet houden, kunnen nieuwe operaties eenvoudig toegevoegd worden. Dit is een eigenschap van het framework dat niet onderschat mag worden.

6.1.2 Analyse van de gegenereerde code

Om de EDSL te evalueren maken we een vergelijking tussen het FRP-programma aan de ene kant en het gegenereerde programma aan de andere kant. We proberen vast te stellen in welke mate het gegenereerde programma zal toenemen afhankelijk van het invoerprogramma van de codegenerator.

Om te beginnen maken we een inschatting van de hoeveelheid code die gegenereerd zal worden voor een FRP-programma. In sectie 4.2 kwam gedetailleerd aan bod hoe elke node omgezet wordt naar een nodefunctie. Dit geeft aan dat de codegenerator voor elke node die we definiëren in het FRP-programma overeenkomstig een functie zal genereren. De hoeveelheid nodefuncties is dus lineair afhankelijk van de hoeveelheid nodes. Afhankelijk van de aanpak die we gebruiken om de toplevelfuncties op te bouwen zal het aantal globale variabelen die we genereren verschillen. Bij de eenvoudige opbouw zullen er per nodefunctie twee globale variabelen gegenereerd worden per nodefunctie. Er is steeds een status nodig voor elke node waardoor afhankelijke nodes kunnen bepalen of ze al dan niet opnieuw geëvalueerd moeten worden. Wanneer we echter de geoptimaliseerde opbouw bekijken worden er veel minder globale variabelen gegenereerd. Aangezien de eventwaardes niet meer globaal toegankelijk moeten zijn, zal er per nodefunctie nul of één globale variabele in de gegenereerde code terechtkomen. Alleen de behaviornodes introduceren zo een globale variabele om de toestand van het systeem vast te kunnen houden over verschillende functieoproepen. De status van de verschillende nodes wordt nu als lokale variabelen doorgeven in de toplevelfunctie. We geven in tabel 6.1 hier nog even een overzicht van.

Opbouw toplevelfunctie	Behaviornode	Eventnode
Eenvoudig	2	2
Optimalisatie	1	0

Tabel 6.1: Aantal gegenereerde globale variabelen per node

Het is duidelijk dat bij de geoptimaliseerde versie de geheugenvereiste heel wat lager is. Dit is een belangrijke factor om rekening mee te houden bij ingebedde systemen waarbij de hoeveelheid geheugen vaak beperkt is. Naast minder geheugengebruik zorgt de geoptimaliseerde toplevelfunctie er ook voor dat eventdata niet steeds bij een nieuw inputevent geïnitialiseerd moet worden. De geoptimaliseerde toplevelfunctie is een duidelijke verbetering ten opzichte van de eenvoudige aanpak. Wat betreft de uitvoeringstijd van het programma kunnen we echter geen formele garanties voorleggen. We bespreken wel een aantal vaststellingen.

Een eerste vaststelling is dat de graaf acyclisch moet zijn. We bieden immers geen constructies aan die het mogelijk maakt om een graaf te definiëren die cyclisch is. Dit wil zeggen dat een event dus steeds na een eindige tijd stopt met doorheen de graaf te propageren op voorwaarde dat de nodefuncties eindig zijn.

Een tweede vaststelling is dat er momenteel geen beperking is op de expressies die de gebruiker kan formuleren in de nodefuncties. Daarom is het dus mogelijk om een oneindige lus te creëren in een nodefunctie. Het limiteren van de expressies voor functies is een oplossing om dit tegen te gaan. Dit is ook behoorlijk eenvoudig met het LMS-framework. Door selectief een aantal impliciete conversies onbeschikbaar te maken voor de gebruiker zullen deze expressies niet meer omgezet kunnen worden naar de nodige Rep-types. De keerzijde is dus wel dat de EDSL minder expressief wordt. Dit is een zeer terechte vaststelling die we al zagen terugkomen in al het voorgaande werk: expressiviteit gaat ten koste van uitvoeringsgaranties.

Een derde vaststelling is dat modules niet cyclisch gekoppeld kunnen worden zodat er ook tussen verschillende modules geen oneindige lus gevormd kan worden. Hoewel we geen formele garanties bieden, geven deze vaststellingen wel een indicatie voor de eindigheid van het FRP-programma.

6.1.3 Type-veilige modules voor Sancus

Om te beginnen is de beschikbaarheid van de FRP-abstracties voor het genereren van programma's voor het Sancus framework een duidelijk voordeel. Het biedt een abstractere manier om een collectie van samenwerkende modules op te bouwen tot een IoT-applicatie. Een mooie extra toevoeging is dat we deze modules ook op een type-veilige manier kunnen koppelen. We bieden in de EDSL de nodige constructies aan om ervoor te zorgen dat de uitvoerstroom van een module hetzelfde type zal hebben als de invoerstroom van de gekoppelde module. Op die manier kunnen tijdens de uitvoering van het programma *runtime* uitzonderingen vermeden worden. Dit kan niet gegarandeerd worden indien het programma via de low-level programmeertaal geprogrammeerd wordt.

6.1.4 Synchrone eventafhandeling

Een laatste belangrijke eigenschap van onze EDSL is dat events synchroon afgehandeld worden. Zoals al aangegeven in hoofdstuk 2 kan dit zorgen voor vertraging wanneer een event dat een langere verwerking nodig heeft ook andere events weerhoudt van afgehandeld te worden. In het geval dat er een event optreedt dat van hogere prioriteit is dan het event dat momenteel afgehandeld wordt, kan dit momenteel enkel als eerstvolgend event gepland worden om afgehandeld te worden als het huidige event volledig door graaf gepropageerd is. Eens een event aangeboden wordt aan het FRP-framework moet dit volledig doorheen de graaf propageren om een consistente toestand te behouden. Binnen het FRP-famework is er momenteel dus geen ondersteuning voor asynchrone eventafhandeling. Om deze reden nemen we dit ook op in het future work in sectie 6.3.

6.2 Related work en vergelijking

In deze sectie behandelen we twee andere frameworks die zich op hetzelfde domein focussen. Het eerste is FRP-Arduino [1] dat niet als academisch onderzoeksproject is ontwikkeld. Het leunt echter wel sterk aan bij de doelen die we in dit werk voorop hebben gesteld. Het tweede werk is Flask [7] en richt zich op FRP voor sensornetwerken.

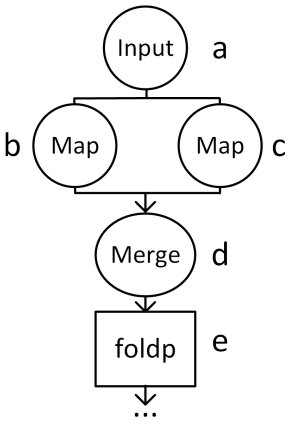
6.2.1 FRP-Arduino

FRP-Arduino biedt een FRP-framework aan voor het ingebed systeem, Arduino Uno. Net zoals in dit werk wordt er een EDSL voorgesteld, deze keer met Haskell als hosttaal. Een codegenerator zal uiteindelijk het FRP-uitvoerprogramma genereren dat voor het Arduino Uno systeem gecompileerd kan worden. Om de lezer de kans te geven een zicht te krijgen op de gelijkenissen en verschillen tussen onze EDSL en FRP-Arduino hebben we het voorbeeld zoals in sectie 3.1 van dit werk ook geïmplementeerd in dit framework. Zowel de implementatiecode als de gegenereerde code kan teruggevonden worden in bijlage F.

Listing 6.1: Filter- en foldp-operatie in FRP-Arduino

```
static void stream_9(uint8_t arg, void* value) {
  uint16_t input_0 = *((uint16_t*)value);
  // filter function
  if ((10 > input_0)) {
    stream_10(0, (void*)(&input_0));
  }
}

static void stream_10(uint8_t arg, void* value) {
  uint16_t input_0 = *((uint16_t*)value);
  static uint16_t input_1 = 0;
```



Figuur 6.1: Volgorde van node evaluatie

```
  // foldp function
  input_1 = (input_1 + input_0);
  stream_11(0, (void*)(&input_1));
}
```

Het belangrijkste verschil met onze implementatie is dat er geen gebruik wordt gemaakt van toplevelfuncties maar dat in de plaats daarvan de verschillende node-functies elkaar oproepen. Codefragment 6.1 toont hoe de verschillende nodefuncties elkaar oproepen. Merk ook op dat de nodes in kwestie de filternode en de foldpnode voorstellen. Omdat de verschillende nodes elkaar op een statische manier oproepen, betekent dit dat de afhankelijkheden tussen de verschillende nodes vast liggen. Het is duidelijk dat men hierin dus ook kiest om eerste-orde FRP te ondersteunen met een statische FRP-graaf. Omdat de verschillende nodes elkaar direct oproepen wil dat zeggen dat een event de graaf in *depth-first* volgorde doorkruist. We hebben echter wel gemerkt dat er geen synchronisatie is in de nodefuncties die meerdere ouderwaardes verwachten. In deze gevallen wordt de FRP-graaf vanaf deze node en de daarop volgende nodes dubbel uitgevoerd. Om dit toe te lichten geven we in figuur 6.1 een situatie waarbij dit probleem zich voordoet. In ons framework zullen de nodes in de volgende volgorde geëvalueerd worden:

$$a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$$

Bij een *depth-first* aanpak moet synchronisatie in de merge-node er voor zorgen dat de volgende volgorde afgedwongen wordt:

$$a \rightarrow b \rightarrow d \rightarrow c \rightarrow d \rightarrow e$$

We hebben echter vastgesteld dat het FRP-Arduino framework deze synchronisatie niet toepast en de nodes als volgt geëvalueerd worden:

$$a \rightarrow b \rightarrow d \rightarrow e \rightarrow c \rightarrow d \rightarrow e$$

Het gevolg daarvan is dat in dit voorbeeld de foldpnode tweemaal uitgevoerd wordt en dat hierdoor één inputevent zorgt dat deze node twee keer geëvalueerd wordt. Dit is mogelijks niet wat de gebruiker van de EDSL zal verwachten. De Elm taal [3] hanteert ook een *depth-first* aanpak maar in de implementatie zorgt men daar wel voor het nodige synchronisatiemechanisme. Het probleem wordt in de voorgestelde EDSL dus op een andere manier aangepakt, maar onze oplossing zorgt er eveneens voor dat het FRP-programma het gedrag vertoont dat de gebruiker conceptueel verwacht.

6.2.2 Flask

Het tweede project dat we bekijken is Flask [7]. In dit project wordt een FRP-framework voorgesteld waarin applicaties ontwikkeld kunnen worden voor een sensornetwerk. Een sensornetwerk bestaat typisch uit een grote verzameling van kleine ingebedde systemen die zeer beperkt zijn in tijd en ruimte. Flask focust dan ook op de garantie dat het gegenereerde programma zowel in tijd als ruimte begrensd is. Het doet dit door in de EDSL een taal aan te bieden waarin expressies steeds eindigen en ook een eindige hoeveelheid geheugen consumeren. Ook in Flask wordt aangegeven dat dit alleen mogelijk is wanneer de FRP-graaf statisch is. Daarom bieden ze dus ook een framework aan dat om kan met eerste-orde FRP zoals in het framework dat in dit werk wordt voorgesteld.

Omdat de Flask-implementatie afhankelijkheden heeft op een oude build van Haskell is het niet voor de hand liggend om opnieuw het voorbeeldprogramma te implementeren. We zijn er daardoor niet in geslaagd het codevoorbeeld in Flask te implementeren. Omdat we niet beschikken over een gegenereerd programma maakt dit een vergelijking van de aanpak met de onze moeilijk. In de plaats daarvan kunnen we wel iets dieper ingaan op de manier waarop Flask verschillende modules koppelt. In het Flask project is er uiteraard geen sprake van modules maar er is wel communicatie tussen verschillende nodes. Ook daar wordt aangegeven dat het belangrijk is om de nodes op een type-veilige manier met elkaar te verbinden. Omdat het gaat over een sensornetwerk is het voor de hand liggend dat verschillende nodes met elkaar kunnen communiceren. De nodes kunnen zoals in Sancus een stroom van waardes doorsturen. Flask gebruikt een abstractie genaamd *channels* of kanalen om af te stemmen op een bepaalde stroom van eventwaarden. Met behulp van de twee primitieven *send* en *recv* kunnen verschillende nodes gekoppeld worden. We beginnen met de *send* primitief:

$$send :: FlowChannel \rightarrow S\ \alpha \rightarrow S\ ()$$

Zoals de signatuur van *send* aangeeft kan een identificatie van een bepaald kanaal gecombineerd worden met de uitvoerstroom voor de node. Indien de stroom een event propageert van type α wordt dit verzonden op het kanaal.

Vervolgens kan een andere node instemmen op dit kanaal met de *recv* functie. Op basis van de kanaalidentificatie verkrijgt deze node een getypeerde eventstroom.

$$recv :: FlowChannel \rightarrow S\ \alpha$$

Als we dit vergelijken met onze implementatie om modules te koppelen, zien we dat in er in beide gevallen een getypeerde stroom gecreëerd wordt. Het enige verschil is wel dat het in onze implementatie onmogelijk is een module te koppelen (via ExternalEvent) zonder dat er op voorhand een uitgang gecreëerd is. In Flask is dit wel mogelijk omdat het zendende kanaal niet type-veilig gekoppeld wordt aan het ontvangende kanaal. De koppeling gebeurt op basis van een kanaalidentificatienummer en wanneer deze niet overeenkomen zullen er geen events doorgegeven worden. Dit kan een lastig te traceren fout in het programma vormen. Met andere woorden kan een *recv*-operatie gedefiniëerd worden op een kanaal dat verschilt van het kanaal waarop de zendende node is afgestemd. Dit is niet het geval in onze EDSL omdat naast de koppeling van de eventstromen ook de modules zelf op een type-veilige manier gekoppeld worden.

6.3 Future work

Er zijn nog een aantal aspecten in de implementatie waar uitbreiding mogelijk is. We geven hier een kort overzicht van met daarbij eventueel een voorstel voor de invulling ervan.

- Om te beginnen zijn automatische testen belangrijk bij het ontwikkelen van software. Dit is typisch moeilijker bij event-gedreven programma's omdat er enorm veel verschillende uitvoeringspaden mogelijk zijn. Een mogelijke uitbreiding van de huidige codegenerator is om naast het FRP-programma ook een testscenario op te kunnen stellen. Dit houdt in dat er een reeks events gedefinieerd kunnen worden met daarbij de toestand die men verwacht voor het systeem na elk event. Op basis van dit testscenario zou dan een programma gegenereerd kunnen worden dat de huidige toestand na elk event vergelijkt met de verwachte toestand.

- Hoewel modules een FRP-graaf van arbitraire grootte kunnen bevatten, is er momenteel slechts ondersteuning om één eventstroom te gebruiken als uitvoer van de module. Een eerste mogelijkheid om dit uit te breiden is om verschillende methodes te voorzien die modules maken met één, twee, ... uitgangen. Dit schaalt echter niet goed naar grotere aantallen. Een betere oplossing lijkt ons om een algemene module te kunnen creëren met een heterogene lijst van eventstromen die men wil uitvoeren. Het voordeel daarvan is dat er eventstromen van verschillende types de module kunnen verlaten. Het tweede voordeel is dat de verschillende elementen in deze lijst op een veilige manier uit de lijst geselecteerd kunnen worden op basis van een uniek type per element in de lijst. Zo kan men in het FRP-programma niet buiten de lijst van mogelijke uitvoerstromen treden.

- De module die we kunnen creëren via de methode *createLCDModule* zoals voorgesteld in sectie 5.1.3 laat momenteel enkel toe om integers op het scherm

te tonen. We hebben deze beperking opgelegd omdat de nodige ondersteuning nog ontbreekt in het LMS framework om alle types om te zetten naar een *String*voorstelling. Een mooie uitbreiding is om alle types op het scherm te kunnen tonen.

- Verder onderzoek is nodig om na te gaan hoe events asynchroon afgehandeld kunnen worden. Dit kan nuttig zijn om bijvoorbeeld *preemptive* events toe te laten. Een event van een hogere prioriteit kan dan voorrang krijgen op een event met lagere prioriteit dat momenteel afgehandeld wordt.

6.4 Overzicht

De FRP EDSL die we hebben voorgesteld heeft een aantal duidelijke eigenschappen. We vatten deze hier opnieuw even beknopt samen:

- Topologische volgorde wordt gerespecteerd in de toplevelfuncties. Deze functies kunnen de verschillende nodes in een *broad-first* volgorde evalueren. Dit is belangrijk om ervoor te zorgen dat er geen glitches optreden in het FRP-programma. Omdat de toplevelfunctie de verschillende nodefuncties elk afzonderlijk oproept, dient er in de nodefunctie alleen rekening gehouden te worden met de functionaliteit van die node.
- Omdat er in de afzonderlijke nodefuncties geen rekeningen gehouden moet worden met de positie van deze node in de graaf, is het eenvoudig om nieuwe FRP-operaties toe te voegen. Dit komt de uitbreidbaarheid van het framework ten goede.
- Wanneer een event optreedt in het systeem moet dit event eerst doorheen de gehele FRP-graaf propageren alvorens het volgende event afgehandeld kan worden. Daarom is onze FRP-implementatie synchroon. Hoewel de implementatie ervoor zorgt dat de propagatie van een event steeds eindigt, kunnen we geen garanties bieden dat het FRP-programma dat ook doet. We kunnen dit oplossen door het FRP-programma slechts toegang te geven tot een beperkte set van expressies.
- De programma's die gegenereerd worden zijn efficiënt omdat we geen eventmanager nodig hebben *at runtime* om binnen een bepaalde FRP-graaf te bepalen welke nodes afzonderlijk opgeroepen moeten worden. De relaties tussen verschillende events en behaviors liggen vast. We spreken van een statische FRP-graaf. We kunnen de aangeboden FRP-concepten bestempelen als eerste-orde FRP door de combinatie van een synchroon FRP-framework met een statische FRP-graaf.
- Als laatste zorgt de EDSL ervoor dat modules op een type-veilige manier aan elkaar gekoppeld kunnen worden. Dit geeft ons de garantie dat er *at runtime* geen typefouten zullen optreden tussen de eventstromen die de verschillende modules met elkaar verbinden.

Hoofdstuk 7

Besluit

Het doel van de masterproef was het opstellen van een *Embedded Domain Specific Language* die functioneel reactief programmeren mogelijk maakt voor ingebedde systemen. Net omdat we ons richten op ingebedde systemen is efficiëntie daarbij erg belangrijk.

Om dit te bewerkstelligen zijn we systematisch te werk gegaan. We hebben in de literatuurstudie eerst voorgesteld hoe verschillende implementaties het onderzoek rond FRP vooruit hebben gebracht. Daarbij lag de focus op de afweging tussen efficiëntie en expressiviteit. We zijn dan ook op zoek gegaan naar de belangrijkste oorzaken van *runtime overhead* bij bestaande FRP-implementaties. De efficiëntie van zo'n implementatie is zeer sterk verbonden met de graad van expressievermogen in de reactieve taal. Zo stellen we vast dat implementaties met continu gedefinieerde behaviors en een dynamische graaf typisch minder efficiënt zijn, maar wel zeer expressief.

Vervolgens hebben we getoond hoe de ontwikkelde FRP API gebruikt kan worden om FRP-programma's op te stellen. Pas daarna zijn we in hoofdstuk 4 en 5 dieper in gegaan op de implementatiedetails. De belangrijkste keuzes voor de EDSL die in dit werk voorgesteld zijn, richten zich op efficiëntie. We hebben daarom op een succesvolle manier een FRP-framework ontwikkeld met ondersteuning voor statische FRP-grafen. Door de relaties tussen de FRP-concepten vast te leggen is er geen dynamische dispatching van events nodig *at runtime*. Een tweede belangrijke eigenschap van de FRP EDSL is dat events synchroon worden afgehandeld. Beide eigenschappen maken dat onze oplossing gecategoriseerd kan worden als eerste-orde FRP-framework. Op die manier slagen we erin om de gegenereerde code efficiënt uit te voeren op een ingebed systeem. Een belangrijk neveneffect daarvan is dat de expressiviteit daardoor naar beneden gaat.

Als specifieke toepassing richten we ons met de FRP EDSL op Sancus, een beveiligd module architectuur. Door de flexibiliteit van de codegenerator is het relatief eenvoudig om nieuwe doeltalen te ondersteunen. Door een toevoeging van domein specifieke operaties voor Sancus, kan de FRP EDSL een aantal voordelen bieden

ten opzichte van de imperatieve manier van programmeren. Zo zijn ten eerste de voorgestelde FRP-abstracties beschikbaar om op een hoger-niveau event-gedreven programma's te ontwikkelen voor Sancus. Ten tweede kunnen beveiligde modules nu op een type-veilige manier met elkaar verbonden worden om *runtime*fouten te vermijden.

Bijlagen

Bijlage A

Wetenschappelijk artikel

Functional Reactive Programming on Embedded Devices

Ben Calus, Bob Reynders, Dominique Devriese, Frank Piessens

Abstract

Event-driven programming is sensitive to hard to find bugs because of explicit state changes. Because there are often a large amount of different incoming events, manually altering the whole state consistently is hard. Just as creating GUIs, embedded systems are also typically event-driven. With IoT applications growing in popularity, embedded systems are catching up in relevance with regard to event-driven programming. Functional Reactive Programming (FRP) is an alternative that can be used to program event-driven systems. FRP is a declarative way of composing event-driven programs, relieving the programmer of explicit state changes. We present an Embedded Domain Specific Language (EDSL) that provides FRP primitives that can be used specifically to create programs for embedded systems. We achieved this by building a code generator using the Lightweight Modular Staging (LMS) framework. This code generator makes it possible to translate an FRP program, written in the EDSL language, to C-code to be compiled for an embedded system. Because of the flexibility of the code generator, we managed to generate specialized C code for Sancus, a secure modules system that enables modular applications to be executed on embedded systems with an untrusted software base.

Keywords Functional Reactive Programming, Embedded systems, Lightweight Modular Staging, Sancus

1 Introduction

With Internet of Things applications growing in importance and popularity, efficiently creating event-driven programs on embedded systems is important. In a network of collaborating nodes, each node reacts to events from one of its peer nodes to perform specific actions. Every node can possibly react to incoming events from different nodes or can react to locally generated events. After processing the event it can eventually propagate an event across the network for further processing. The EDSL that will be presented tries to make it more convenient to compose programs for this particular domain. By providing FRP primitives in the EDSL, it becomes more natural to describe this type of programs.

In the following section we will explain why event-driven programs can be hard to compose and how FRP can be used as a declarative alternative. The next section will then describe how the Lightweight Modular Staging (LMS) framework can be used to implement the EDSL that provides the FRP abstractions. In the fourth section we present

how the FRP abstractions can be used to create programs for a specific embedded system in a sensor-data network with a concrete example. Next, we point out the most important contributions our EDSL has to offer. Our work is then compared to other related work and in the last section we state our most important conclusions.

2 FRP

Event-driven programs on embedded systems are very important in a network for IoT applications. Most of the time a very procedural style of programming is used to create event-driven applications. This means different callback routines must be registered to listen to external or internal events. When an event is received, the reaction will typically be based on the type of event and/or its content as well as the internal state of the node. When the internal state becomes large, it becomes more likely errors are introduced [1, 11]. Complex conditional logic to determine the reaction to the incoming event is a first classical source of error. When reacting to an event, the internal state often has to be updated accordingly. This is another possibility to introduce errors. If the internal state is only partially updated, inconsistencies in the internal state can produce wrong reactions in the future or even a crash of the application. Functional Reactive Programming can offer a solution. The introduction of additional abstractions moves the responsibility of consistent updates to the framework providing the FRP abstractions, relieving the programmer from this task. Since these abstractions are easy to compose, the conditional logic becomes more manageable.

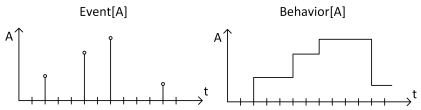


Figure 1: Event vs. Behavior

The EDSL provides two abstractions: Events and Behaviors [3].

Events can be seen as a sequence of discrete events, each containing a concrete value. This sequence of events can be presented as a list of time-value pairs.

$$\text{Event a} = [(\text{Time, a})]$$

Behaviors can be seen as a continuous defined value that change over time. It can be represented as a function from time to value.

$$\text{Behavior a} = \text{Time} \longrightarrow \text{a}$$

FRP uses both abstractions to handle an asynchronous stream of events, where events are used to represent a stream of events and behaviors are used to build internal state. In addition to these new primitives, the EDSL defines different operations on them to make it possible to transform them or convert between both. Next, we will first introduce a small portion of our FRP API on events and behaviors with an example. Later on, we dive into some details how the EDSL is created using the LMS framework and how this example will be translated.

3 FRP API in action

We will start with the introduction of a graphical way of representing an FRP program. With a simple example the most important transformations on events and behaviors can be shown.

Suppose we want to create a node that can receive two different input events, both carrying an integer value. When it receives an event on *input1* it adds it to an internal counter. In case it receives one on *input2*, it substracts it from the counter. As an extra case, we only want the counter to be adapted if the value to be added or subtracted is smaller than 10. The reaction of the node is always the new value of the counter.

To define this program we need different ways to transform events and behaviors. The program can be visualized easily using a graph as in figure 2. Note that Events are visualized with a circle, while Behaviors are with a rectangle.

To start we define two input events which carry an integer value. In case of *input2* we use the map operation on the input event to transform the value into another value. This mapping negates the incoming integer. Next, both event streams are merged together. This means the resulting event stream can carry as well positive as negative values. Next, we want to filter out all events that carry an absolute value greater than 10. At the end we transform the event to a behavior with the folp operation. Foldp

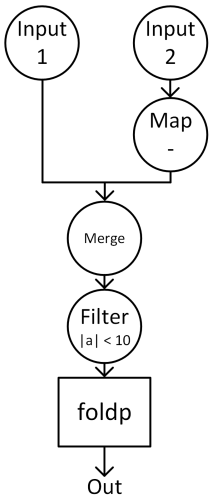


Figure 2: Counter example

stands for fold from the past, meaning a value is build up from values that occurred over time. Because this value is defined at all times, this is a behavior and will represent the state of our application. Each time an event is received, the value of the behavior will be updated accordingly to the function that is defined with the foldp-operation. In code snippet 1 you can see how this can be formulated in our FRP EDSL. Also note the merge operation requires an extra argument. When both inputs streams to the merge operation propagate a value at the same moment, the function in this argument is used to determine the resulting value to be propagated. We also need to define a starting value as a second argument to the foldp-operation to make sure the behavior is always defined.

`Listing 1: FRP EDSL code`

```
1  val input1 = InputEvent[Int]
2  val input2 = InputEvent[Int]
3  val negate2 = input2.map( (i) => -i)
4  val merged =
5   input1.merge(negate2, (x,y) => x + y)
6  val filtered =
7   merged.filter( x => abs(x) < 10)
8  val counter =
9   filtered.foldp((state,x)=>state + x, 0)
10 out(counter.changes())
```

In figure 3 the created streams are visualized with concrete examples for the event values to show how the application works. The first stream represents the *input1* event while the following two streams model the *input2* and *negate2* events. Next, we see how both streams are merged into the *merged* event stream and so can contain positive as negative event values. Afterwards the filter operations will create the *filtered* event stream where only events with a smaller value than 10 will be present. This *filtered* stream feeds the foldp-behavior, *counter*. Each time this stream propagates a new event, the function associated to the foldp-behavior is used to update its value. Notice that the behavior is defined throughout the whole lifetime. Eventually this *counter* stream can be used as the output of the network node.

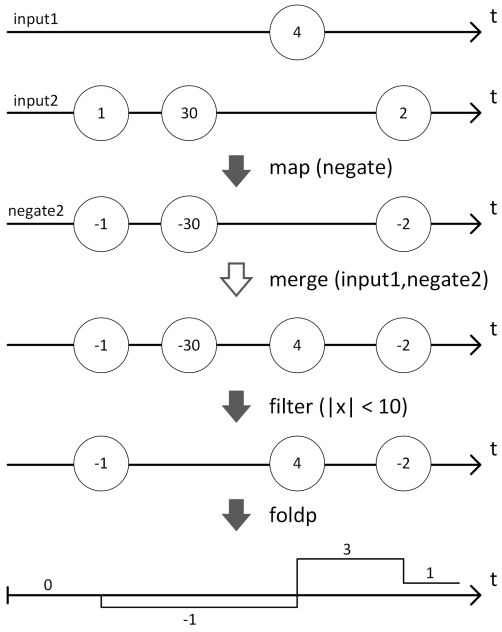


Figure 3: Stream visualization

4 LMS

To make it possible to provide the new primitives introduced in the previous section, we need to create an EDSL. The Lightweight Modular Staging framework [9] provides a library of core components for building high performance code generators and enables us to create an EDSL, embedded in scala. Because the framework is provided as a library it is easy to extend it to provide extra features.

The LMS framework enables us to convert the high-level FRP code into low-level constructs in the target language. This means that all abstractions will eventually be converted to low-level C-code where the explicit state changes will be generated by the framework. The state changes will effectively be hidden by the framework in this way. LMS uses staging [10] to make it possible to execute code at different points in time. LMS uses types to make a distinction between code to be evaluated *now* (T), in contrast to code to be evaluated

later (Rep[T]). In our EDSL, *now* means all code that is evaluated when generating the target C program and *later* means the code that eventually will be present in the target application. Because of this distinction, we are able to create an internal representation of all dependencies specified by the FRP primitives. Typically we will build an FRP graph that represents the logic of the target application as shown in the previous section. Based on this graph representation we generate low-level code that represents the same application logic. The great thing about this is that the generated code won't need any graph information. This is important because the target program will not have the runtime overhead due to graph traversal. This means the target program will be more optimal than an FRP framework that keeps track of the graph at runtime in the target program. An implication of this technique is that the FRP graph is staticly embedded in the target program and cannot be changed at runtime. We won't go into detail about how the different constructs are specifically translated to the target language.

5 Contributions

As shown in the previous sections the EDSL makes it possible to create event-driven applications using high-level abstractions. The first main contribution is the development of the FRP API. This API is mainly inspired by the FRP API of elm [2] and flapjax [6]. The second contribution is the translation of these abstractions to procedural constructions. As stated in the previous section, most of the graph overhead isn't present in the generated target code. This implies our FRP graph is of first-order. This means the graph is static and can not be changed at runtime. Some result of this generation process was shown in the previous section and the resulting generated code can be explored in listing **??** in the appendix.

The LMS framework, used to implement the code generator, is very flexible to extend and can be used to translate the FRP program to different target languages. In the first place, the target language was the standard C language, but because LMS is implemented as a library it is easy to add very specific constructs and/or languages. This is ideal to achieve code generation for very specific target languages. The third and largest contribution of this thesis is the possibility to target Sancus systems [8]. With Sancus, it is possible to securely run applications in an untrusted software environment. Very specific constructs implemented as macros in the C language can be used to create modules that run securely. Because the code generator can be easily adapted to these specific language constructs, it can be used to generate programs for Sancus modules using the FRP concepts. An added value of using the FRP EDSL for this domain is that additional type safety can be provided to couple different modules. In Sancus, each module is self-contained, but different modules can be chained together to compose a network of nodes. The example program [7] implements a network of nodes collaborating to detect parking violations. In listing 2 we will focus

on the chaining of modules.

Listing 2: Procedural style

```
1  module A(C1,S1;V1);
2  on S1(x) {
3    if x then taken = 1;
4    else taken = 0; count = 0; V1(0);
5  }
6
7  on C1(x) {
8    if taken then count = count + 1;
9    if count > MAX then V1(1);
10 }
11
12 module B(C2,S2;V2);
13 ... similar
14
15 module C(VP1,VP2; D);
16 on VP1(x) {
17   if x then c1 = 1 else c1 = 0;
18   if c1 then D(1)
19   if c2 then D(2)
20 }
21 on VP2(x) ... similar
22
23 V1 -> VP1
24 V2 -> VP2
```

However the different modules can be linked to each other, type safety is not guaranteed and is the responsibility of the programmer. Notice how for example V1 is chained to VP1, but both input and output have no type. By providing specific primitives for the Sancus modules in our EDSL, it becomes possible to create multiple modules and afterwards link these modules together in a type safe way. Listing 3 gives an idea of how modules can be constructed in the EDSL.

Listing 3: FRP style

```
1  val modA: Module[Boolean] =
2   createModule {
3    val s1 = SensorInput[Boolean]
4    val c1 = TimerInput[Int]
5    ...
6    val violation: Event[Boolean] = ...
7    out[Boolean](violation)
8  }
9
10 val modB: Module[Boolean] =
11  createModule {
12   ... similar
13 }
14
15 val modC = createModule {
16   val vp1 = ExternalEvent(modA.out)
17   val vp2 = ExternalEvent(modB.out)
18   ...
19 }
```

Note that *vp1* and *vp2* are of type Event[Boolean] since this can be inferred from the module.

As well the FRP concepts, as the type safety between different modules is the added value of using the presented EDSL to generate programs for a Sancus network.

6 Related Work

The presented EDSL, used to generate low level C code, uses concepts of Functional Reactive Programming. The classical definition of FRP was introduced in Fran by Elliot and Hudak [3]. From this point on, very different FRP implementations were introduced focusing on different aspects of FRP. Fran or Functional Reactive Animation introduced functional reactive programming in a very expressive way. The initial problem with this implementation was a lack of efficiency. This is why future work on FRP like Real-time FRP [12] focussed on more efficient implementations, often in combination with a reduction of expressiveness. The domain of GUIs in

combination with the web are also a great place to use the concepts of FRP since these are typically event-driven. Some important FRP implementations are the elm language [2] and the flapjax framework [6], which were a great inspiration for the different API operations on Events and Behaviors in our EDSL. While elm uses signals to combine events and behaviors, flapjax still makes the distinction between these, as in the original definition of FRP by the Fran system. Since FRP is relevant for all event-driven domains, the domain of embedded software is another interesting domain to use FRP. Some other implementations also try to provide a language or EDSL to make the FRP concepts available. Real-time FRP [12] focuses on efficiency and does so by providing an unrestricted base language and a more limited reactive language. The reactive language provides an efficient implementation but as a consequence is less expressive. This language is still very usefull in time critical embedded systems. Another interesting project contains a library to create FRP programs using Haskell to target the arduino platform [4]. Relevant to the sensor-network subdomain in the embedded systems domain, the work on Staged Functional Programming on Sensor Networks [5] is related to the domain our EDSL focusses on.

7 Conclusion

We have presented our EDSL targetting the domain of embedded software. It enables programmers to use concepts of Functional Reactive Programming to write embedded software in a more declarative way. Using the Lightweight Modular Staging framework, we created a code generator that transforms the high-level FRP program to a low-level C program that can be compiled for a specific embedded system. Thanks to the flexibility of the LMS framework, we were able to adapt the code generator to generate a more specialized C dialect, used for Sancus embedded systems. This makes it possible to not only use FRP concepts on a high-level to compose the applications, but also combine different Sancus modules in a type safe way.

References

[1] A possible future of software development.

[2] E. Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

[3] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[4] frp arduino. Frp arduino, 2015. https://github.com/frp-arduino/frp-arduino.

[5] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, volume 43, pages 335–346. ACM, 2008.

[6] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[7] J. Noorman. Authentic execution of distributed event-driven applications with a small trusted computing base, 2015.

[8] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, pages 479–494, 2013.

[9] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM.

[10] W. Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[11] G. M. Tchamgoue, K.-H. Kim, and Y.-K. Jun. Testing and debugging concurrency bugs in event-driven programs. *International Journal of Advanced Science and Technology*, 40:55–68, 2012.

[12] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *ACM SIGPLAN Notices*, volume 36, pages 146–156. ACM, 2001.

Bijlage B

Populariserend artikel

Functional Reactive Programming on Embedded Devices

Ben Calus

8 februari 2016

De complexiteit om een computerprogramma te begrijpen is sterk afhankelijk van de manier waarop het programma wordt uitgedrukt. We stellen een zeer eenvoudig programma voor dat een getal kan op- en aftellen en vervolgens ook kan weergeven van 0 tot 99. We interageren met het programma door middel van een Grafische User Interface of GUI met daarop twee knoppen om op en af te tellen. Het resultaat wordt ook in de GUI weergegeven. Het programma blijft zichzelf herhalen tot het venster gesloten wordt. We vergelijken twee mogelijke manieren om dit programma voor te stellen.

```
programma 1 {
 if (knopEvent ?= plusknop)
   { teller + 1 }
 else if (knopEvent ?= minknop)
   {
     if (teller > 0)
       { teller - 1 }
   }
 if (teller >= 0 & teller < 100)
   { update teller }

 repeat program
}
```

```
programma 2 {
 plusknop.do(teller+1)
 minknop.do(teller-1
            if teller > 0)

 update teller
    if between 0 to 99

 repeat program
}
```

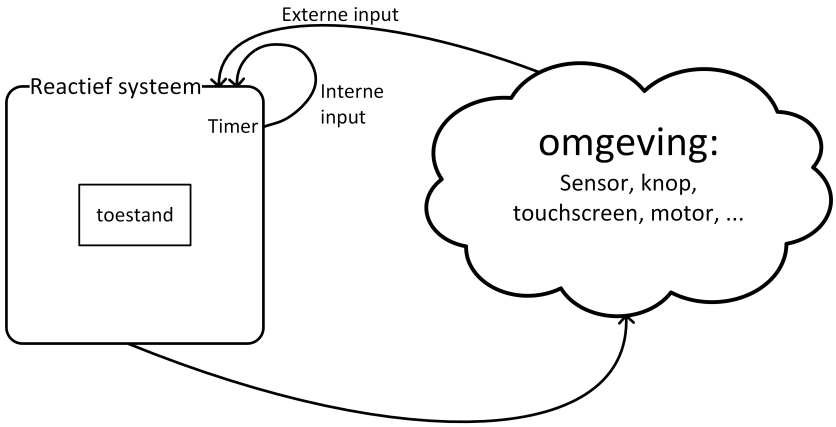
Hoewel beide programma's hetzelfde doel hebben, is het tweede programma beter leesbaar. Dit komt omdat de (pseudo)programmeertaal van het tweede programma korter aanleunt bij de manier waarop we het algoritme in spreektaal zouden beschrijven. We zeggen in het geval van het tweede programma

dat de gebruikte taal van een hoger niveau is dan die van het eerste programma omdat we het algoritme in het programma natuurlijker kunnen uitschrijven. Een hogere programmeertaal bevat abstracties zodat we ons niet in alle details moeten verdiepen die nodig zijn voor de computer om het programma uit te voeren. In het eerste programma moeten we weten dat beide knoppen reageren op een binnenkomend event waarbij we dit event moeten herleiden naar de correcte knop dat dit event heeft gegenereerd. We zien dus al snel dat de aangebode abstractie in het tweede programma ons van deze details afschermt en we eenvoudigweg het gewenste gedrag toekennen aan de afzonderlijke knoppen.

Welke taal we ook gebruiken, uiteindelijk zal deze vertaald moeten worden naar een taal die de uitvoerende machine begrijpt. Uiteindelijk wordt het programma vertaald naar een reeks nullen en enen aangezien dit de enige taal is die door een computer begrepen kan worden. Deze taal is perfect begrijpbaar door een computer maar is enorm moeilijk te interpreteren voor een mens. Dit is met andere woorden de programmeertaal op het laagst mogelijke niveau. De vertaling naar deze binaire taal is typisch moeilijker in het geval van een hogere programmeertaal zoals in het tweede programma. Des te meer abstracties aangeboden worden in de taal, des te meer werk het vergt om deze om te zetten naar de laag-niveau programmeertaal die de computer begrijpt.

Het doel van deze thesis is om een hoog-niveau programmeertaal aan te kunnen bieden, specifiek voor het domein van software voor ingebedde systemen. We willen dus een taal aanbieden die extra abstracties aanbiedt ten opzichte van de conventionele talen die momenteel voor dit domein gebruikt worden. Deze taal moet het eenvoudiger maken om programma's te schrijven voor dit specifieke domein. Dit is nodig omdat momenteel programma's voor ingebedde systemen nog vaak geschreven worden in laag-niveau programmeertalen. De programma's zijn dus moeilijker begrijpbaar en het duurt langer om deze foutloos op te stellen en aan te passen.

Software voor ingebedde systemen verschilt in essentie niet erg veel van de type programma's die we zonet hebben voorgesteld. De invoer bij een ingebed systeem komt bijvoorbeeld van een sensor in plaats van een knop in een venster. In plaats van iets aan te passen in het venster als uitvoer, kan de uitvoer bij een ingebed systeem het aansturen van een motor zijn.
Een voorbeeld van zo'n soort systeem is bijvoorbeeld een automatisch remsysteem. Indien via een sensor een object voor de wagen gedetecteerd wordt, zal het systeem vervolgens de remmen automatisch aansturen. Beide pro-



gramma's hebben gemeen dat ze reageren op bepaalde gebeurtenissen of events. Dit soort programma's noemt men dan ook event-driven. Het grote verschil met andere programma's is dat we niet precies weten wat er op welke moment zal gebeuren in het programma. Beschouw volgend programma dat het weer tracht te voorspellen.

```
programma 3 {
   luchtdruk = 10
   temperatuur = 15
   neerslag = 75

   voorspelling = bereken_via(luchtdruk, temperatuur, neerslag)

   display voorspelling

   end of program
}
```

Dit programma begint met het definiëren van een aantal parameters die de huidige toestand van de atmosfeer voorstellen. Deze worden verzameld om ze vervolgens te gebruiken in de berekening. Daarna starten we een complexe berekening die deze parameters gebruikt om tot een voorspelling te komen. Het programma is pas van enig nut als we aan het einde deze voorspelling tonen aan de gebruiker. Dit soort programmeren noemen we procedureel programmeren. Het programma wordt lijn per lijn uitgevoerd en we kennen de exacte volgorde waarin de instructies worden uitgevoerd.

In tegenstelling weten we bij event-driven programma's niet op voorhand in welke volgorde de instructies uitgevoerd zullen worden. Dit hangt af van

de volgorde waarin verschillende events door het systeem ontvangen worden. Omdat we niet weten in welke volgorde de instructies worden uitgevoerd, maakt dit het redeneren over het programma moeilijker. Zelfs in het eenvoudige programma 1 zien we dat er in de laag-niveau programmeertaal al wel wat conditionele logica aanwezig is. We hebben ten eerste logica nodig om verschillende events van elkaar te onderscheiden. Daarnaast hebben we ook logica nodig om op basis van de huidige toestand te bepalen hoe het systeem dient te reageren. De reactie van het systeem is typisch sterk afhankelijk van de huidige toestand waarin het systeem verkeerd. Wanneer we programma's ontwerpen die veel complexer zijn dan de gepresenteerde programma's, stellen we vast dat de hoeveelheid conditionele logica zeer sterk toeneemt. Hierdoor bevatten event-driven programma's vaak meer fouten dan procedurele programma's. Functional Reactive Programming is een programmeermodel dat een aantal abstracties voorstelt om dit te verhelpen. FRP is toepasbaar op ingebedde systemen omdat de abstracties verband houden met event-gebaseerde programma's. Het laat ons toe de relaties tussen de toestand en verschillende events te definiëren zonder al te veel complexe logica te moeten gebruiken. Net als in programma 2 zien we dat eenvoudige abstracties ervoor zorgen dat de leesbaarheid verbetert en er zo automatisch voor zorgt dat er minder fouten in het programma sluipen. Wanneer we het programma beter begrijpen, is het uiteraard eenvoudiger om fouten te voorkomen of verhelpen.

De eerste uitdaging in deze thesis is om de voorgestelde FRP abstracties aan te bieden in een specifieke programmeertaal voor ingebedde software. Om het programma interpreteerbaar te maken voor een ingebed systeem moet dit programma vertaald worden naar een laag-niveau taal zodat het systeem dit kan uitvoeren. De tweede uitdaging ligt erin om het vertaalde programma zo efficiënt mogelijk te maken. Dit is belangrijk bij ingebedde software en event-driven programma's in het algemeen om ervoor te zorgen dat de reactie op een binnenkomend event niet te lang op zich laat wachten. Om dit te realiseren bouwen we een programmagenerator. De generator, zelf een programma, neemt als invoer een programma en vertaalt dit naar een ander programma. Het gegenereerde programma stelt hetzelfde voor als het invoerprogramma (net zoals programma 1 en programma 2 hetzelfde voorstellen), maar dan uitgedrukt in een laag-niveau programmeertaal die interpreteerbaar is voor het ingebed systeem. Omdat het input programma voor deze generator geschreven is in onze hoger-niveau FRP programmeertaal is het eenvoudiger op te stellen dan in een meer conventionele taal voor ingebedde systemen.

Bijlage C

FRP API

In deze bijlage kan de volledige FRP API teruggevonden worden. We onderscheiden daarin de Event API van de Behavior API. Zoals we in hoofdstuk 3 al hebben vermeld, bevat deze bijlage de werkelijke API. Merk op dat de argumenten van de operaties van type *Rep[T]* zijn.

C.1 Event

```
trait Event[A] {
  def constant[B] (c: Rep[B]): Event[B]
  def map[B](f: Rep[A] => Rep[B]): Event[B]
  def filter(f: Rep[A] => Rep[Boolean]): Event[A]
  def merge(e:Event[A], f:(Rep[A],Rep[A])=>Rep[A]): Event[A]
  def startsWith(i: Rep[A]): Behavior[A]
  def foldp[B](f:(Rep[A],Rep[B]) => Rep[B], init:B): Behavior[B]
}
```

C.2 Behavior

```
trait Behavior[A] {
  def map2[B,C]
    (b: Behavior[B], f:(Rep[A], Rep[B]) => Rep[C]): Behavior[C]
  def snapshot[B](e: Event[B]): Event[A]
  def changes(): Event[A]
}

def ConstantB[A](value: Rep[A]): Behavior[A]
```

Bijlage D

Sancus voorbeeldapplicatie

In deze bijlage tonen we de volledige code voor de voorbeeldapplicatie van het Sancus framework. De code is in dit geval manueel opgesteld en dus niet met de FRP EDSL. We willen met dit codefragment aantonen hoe de gespecialiseerde C-code eruit ziet om modules te creëren.

```
1  // A module ==========================
2  SM_DATA(modA) uint8_t countA;
3  SM_DATA(modA) uint8_t takenA;
4
5  SM_OUTPUT(modA, o1);
6
7  SM_INPUT(modA, s1, data, len)
8  {
9    uint8_t x = data[0];
10   if(x) {
11     takenA = 1;
12   }
13   else {
14     takenA = 0;
15     countA = 0;
16     const uint8_t o1_data = 0;
17     o1(&o1_data, sizeof(o1_data));
18   }
19 }
20
21 SM_INPUT(modA, t1, data, len)
22 {
23   if(takenA){
24     countA = countA + 1;
25   }
26   if(countA > 1) {
27     const uint8_t o1_data = 1;
28     o1(&o1_data, sizeof(o1_data));
29   }
30 }
31
32 // B module ==========================
33 SM_DATA(modB) uint8_t countB;
```

```
34 SM_DATA(modB) uint8_t takenB;
35
36 SM_OUTPUT(modB, o2);
37
38 SM_INPUT(modB, s2, data, len)
39 {
40   uint8_t x = data[0];
41   if(x) {
42     takenB = 1;
43   }
44   else {
45     takenB = 0;
46     countB = 0;
47     const uint8_t o2_data = 0;
48     o2(&o2_data, sizeof(o2_data));
49   }
50 }
51
52 SM_INPUT(modB, t2, data, len)
53 {
54   if(takenB){
55     countB = countB + 1;
56   }
57   if(countB > 1) {
58     const uint8_t o2_data = 1;
59     o2(&o2_data, sizeof(o2_data));
60   }
61 }
62
63 // C module ==========================
64 SM_DATA(modC) uint8_t c1;
65 SM_DATA(modC) uint8_t c2;
66
67 SM_OUTPUT(modC, s);
68
69 SM_FUNC(modC) void printViolations()
70 {
71   if(c1) s(1)
72   if(c2) s(2)
73 }
74
75 SM_INPUT(modC, po1, data, len)
76 {
77   uint8_t violation = data[0];
78   if(violation) c1 = 1; else c1 = 0;
79   printViolations();
80 }
81
82 SM_INPUT(modC, po2, data, len)
83 {
84   uint8_t violation = data[0];
85   if(violation) c2 = 1; else c2 = 0;
86   printViolations();
87 }
```

Bijlage E

FRP Teller applicatie

Deze bijlage bevat de gegenereerde code voor de voorbeeldapplicatie zoals voorgesteld in hoofdstuk 3. Dit voorbeeld vormt doorheen de tekst de leidraad om verschillende concepten uit te leggen. We focussen in het codefragment op de opbouw van de gegenereerde code. Daarom hebben we de functiedetails het voorbeeld verwijderd. Dit zorgt ervoor dat de lezer het overzicht over de code beter kan bewaren.

De code kan in grote lijnen opgesplitst worden in 3 delen. Het spreekt voor zich dat er per module die we hebben gedefiniëerd een deel voorzien is. Het laatste deel bevat de eventhandlers en de mainfunctie die zich buiten de modules bevindt.

```
#include ...

// module 1
SM_DATA(mod1) int mod1_initialiased;
SM_DATA(mod1) int counter;

SM_FUNC(mod1) void init_mod1 () {...}
SM_FUNC(mod1) void input_button1
    (uint8_t* x1,int x2,int* x3,bool* x4) { ... }
SM_FUNC(mod1) void mergefun
    (int x80,bool x81,int x82,bool x83,int* x84,bool* x85) { ... }
SM_FUNC(mod1) void filterfun
    (int x112,bool x113,int* x114,bool* x115) { ... }
SM_FUNC(mod1) void foldpfun (int x135,bool x136) { ... }
SM_FUNC(mod1) void changesfun (int* x146,bool* x147) { ... }

SM_OUTPUT(mod1,x218);

SM_FUNC(mod1) void outputfun (int x207,bool x208) { ... }
SM_INPUT(mod1,toplevelfun1,data,len) {
  init_mod1();
```

```
    bool status = false;
    int waarde;
    input_button1(data,len,&status,&waarde);
    ...
    mergefun(...);
    foldpfun(...);
    filterfun(...);
    changesfun(...);
    outputfun(...);
}

SM_FUNC(mod1) void input_button2
    (uint8_t* x32,int x33,int* x34,bool* x35) { ... }
SM_FUNC(mod1) void mapfun
    (int x63,bool x64,int* x65,bool* x66) { ... }
SM_INPUT(mod1,toplevelfun2,data,len) {
    init_mod2();
    bool status = false;
    int waarde;
    input_button2(data,len,&status,&waarde);
    mapfun(...);
    mergefun(...);
    filterfun(...);
    foldpfun(...);
    changesfun(...);
    outputfun(...);
}

// module 2
SM_DATA(mod2) int mod2_initialised;
SM_FUNC(mod2) void init_mod2 () {...}
SM_FUNC(mod2) void externalinputfun
    (uint8_t* x285,int x286,int* x287,bool* x288) { ... }
SM_FUNC(mod2) void lcd_printfun
    (int x316,bool x317,int* x318,bool* x319) { ... }

SM_INPUT(mod2,toplevelfun1,data,len) {
    init_mod2();
    externalinputfun(...);
    lcd_printfun(...);
}

DECLARE_SM(mod1, 0x1234);
DECLARE_SM(mod2, 0x1234);
```

```
// eventhandlers en mainfunctie
static void init_board () {...}
static void deploy_modules () {
  sancus_enable(&mod1);
  sm_register_existing(&mod1);
  sancus_enable(&mod2);
  sm_register_existing(&mod2);

  REACTIVE_CONNECT(mod1, x218, mod2, x363);
}

static void button1_handler (int pressed) {
  if(pressed) toplevelfunction1(...);
}
static void button2_handler (int pressed) {
  if(pressed) toplevelfunction2(...);
}

int main() {
  init_board();
  deploy_modules();

  buttons_register_callback(Button1,button1_handler);
  buttons_register_callback(Button2,button2_handler);

  while(1) {
    buttons_handle_events();
  }
}
```

Bijlage F

FRP-arduino Teller applicatie

In deze bijlage presenteren we hoe de voorbeeldapplicatie zoals voorgesteld in hoofdstuk 3 eruit ziet wanneer we het implementeren in het FRP-arduino framework. Op basis daarvan hebben we in hoofdstuk 6 dit framework vergeleken met het framework dat we in dit werk voorstellen.

F.1 FRP-programma

```
1  import Arduino.Uno
2
3  main = compileProgram $ do
4    digitalOutput pin13 =: counter ~> toggle
5      where
6        counter:: Stream Word
7        counter = filterStream ~> foldpS fpfun expb
8
9        fpfun:: (Expression Word -> Expression Word -> Expression Word)
10       fpfun a b = b + a
11       expb:: Expression Word
12       expb = 0
13
14       filterStream:: Stream Word
15       filterStream = mergeStream ~> filterS filterFun
16       filterFun:: (Expression Word -> Expression Bool)
17       filterFun x = greater 10 x
18
19       mergeStream:: Stream Word
20       mergeStream = mergeS streams
21
22       streams:: [Stream Word]
23       streams = [ leftinputstream, rightstream ]
24
25       leftinputstream :: Stream Word
26       leftinputstream = clock
27
28       rightstream :: Stream Word
29       rightstream = rightinputstream ~> mapS mapFun
30       mapFun :: (Expression Word -> Expression Word)
31       mapFun x = 0 - x
32
33       rightinputstream :: Stream Word
34       rightinputstream = analogRead a0
```

F.2 Gegenereerde code

```
1  #include <avr/io.h>
2  #include <util/delay_basic.h>
3  #include <stdbool.h>
4
5  struct tuple2 {
6      void* value0;
7      void* value1;
8  };
9
10 // function declarations
11 ...
12
13 static void bootup() {
14   bool temp0;
15   temp0 = 0;
16   stream_1(0, (void*)(&temp0));
17 }
18
19 static void input_a0() {
20   ...
21   stream_7(0, (void*)(&temp3));
22 }
23
24 static void input_timer() {
25   ...
26   stream_2(1, (void*)(&temp4));
27 }
28
29 static void stream_1(uint8_t arg, void* value) {
30   ...
31   stream_2(0, (void*)(&temp5));
32 }
33
34 static void stream_7(uint8_t arg, void* value) {
35   uint16_t input_0 = *((uint16_t*)value);
36   uint16_t temp6;
37   // negate function
38   temp6 = (0 - input_0);
39   stream_8(1, (void*)(&temp6));
40 }
41
42 static void stream_2(uint8_t arg, void* value) {
43   ...
44   stream_3(0, (void*)(&temp9));
45 }
46
47 static void stream_3(uint8_t arg, void* value) {
48   ...
49   stream_4(0, (void*)(&input_1));
50 }
51
52 static void stream_4(uint8_t arg, void* value) {
53   struct tuple2 input_0 = *((struct tuple2*)value);
54   if (*((uint16_t*)input_0.value0) == 0) {
55     stream_5(0, (void*)(&input_0));
56   }
57 }
58
59 static void stream_5(uint8_t arg, void* value) {
60   ...
61   stream_6(0, (void*)(&temp19));
```

```
62  }
63
64  static void stream_6(uint8_t arg, void* value) {
65    bool input_0 = *((bool*)value);
66    static uint16_t input_1 = 0;
67    input_1 = (input_1 + 1);
68    stream_8(0, (void*)(&input_1));
69  }
70
71  static void stream_8(uint8_t arg, void* value) {
72    uint16_t input_0 = *((uint16_t*)value);
73    stream_9(0, (void*)(&input_0));
74  }
75
76  static void stream_9(uint8_t arg, void* value) {
77    uint16_t input_0 = *((uint16_t*)value);
78    // filter function
79    if ((10 > input_0)) {
80      stream_10(0, (void*)(&input_0));
81    }
82  }
83
84  static void stream_10(uint8_t arg, void* value) {
85    uint16_t input_0 = *((uint16_t*)value);
86    static uint16_t input_1 = 0;
87    // foldp function
88    input_1 = (input_1 + input_0);
89    stream_11(0, (void*)(&input_1));
90  }
91
92  static void stream_11(uint8_t arg, void* value) {
93    uint16_t input_0 = *((uint16_t*)value);
94    bool temp20;
95    temp20 = (input_0) % 2 == 0;
96    stream_12(0, (void*)(&temp20));
97  }
98
99  static void stream_12(uint8_t arg, void* value) {
100   bool input_0 = *((bool*)value);
101   if (input_0) {
102     PORTB |= (1 << PB5);
103   } else {
104     PORTB &= ~(1 << PB5);
105   }
106 }
107
108 int main(void) {
109   ...
110   bootup();
111   while (1) {
112     input_a0();
113     input_timer();
114   }
115   return 0;
116 }
```

Bijlage G

Broncode

De volledige broncode van het werk kan teruggevonden worden op de bijgeleverde CD-ROM. Daarop vindt u ook de instructies om de voorbeeldprogramma's uit te voeren.

Bibliografie

[1] Frp arduino, 2015. https://github.com/frp-arduino/frp-arduino.

[2] Lms-tutorial, 2015. https://scala-lms.github.io//tutorials/start.html.

[3] E. Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

[4] E. Czaplicki. Controlling time and space: understanding the many formulations of frp, 2014. https://www.youtube.com/watch?v=Agu6jipKfYw en https://prezi.com/rfgd0rzyiqp_/controlling-time-and-space/.

[5] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In *ACM SIGPLAN Notices*, volume 48, pages 411–422. ACM, 2013.

[6] C. Elliott and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

[7] G. Mainland, G. Morrisett, and M. Welsh. Flask: Staged functional programming for sensor networks. In *ACM Sigplan Notices*, volume 43, pages 335–346. ACM, 2008.

[8] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, page 10, 2013.

[9] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

[10] J. Noorman. Authentic execution of distributed event-driven applications with a small trusted computing base, 2015.

[11] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *USENIX Security*, pages 479–494, 2013.

[12] S. Parent. A possible future of software development, 2006. `https://stlab.adobe.com/wiki/images/0/0c/Possible\_future.pdf`.

[13] J. Regehr. Random testing of interrupt-driven software. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 290–298. ACM, 2005.

[14] R. Strackx, J. Noorman, I. Verbauwhede, B. Preneel, and F. Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.

[15] G. M. Tchamgoue, O.-K. Ha, K.-H. Kim, and Y.-K. Jun. A taxonomy of concurrency bugs in event-driven programs. In *Software Engineering, Business Continuity, and Education*, pages 437–450. Springer, 2011.

[16] G. M. Tchamgoue, K.-H. Kim, and Y.-K. Jun. Testing and debugging concurrency bugs in event-driven programs. *International Journal of Advanced Science and Technology*, 40:55–68, 2012.

[17] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *ACM SIGPLAN Notices*, volume 36, pages 146–156. ACM, 2001.

[18] Z. Wan, W. Taha, and P. Hudak. Event-driven frp. In *Practical Aspects of Declarative Languages*, pages 155–172. Springer, 2002.

Fiche masterproef

Student: Ben Calus

Titel: Functioneel Reactief Programmeren op Ingebedde Systemen

Engelse titel: Functional Reactive Programming on Embedded Devices

UDC: 681.3

Keywords: Functioneel Reactief Programmeren, ingebedde systemen, Lightweight Modular Staging, Sancus

Korte inhoud:

Event-gedreven applicaties zijn vaak het slachtoffer van moeilijk op te sporen fouten door expliciete wijziging van toestand. Het is daarom verrassend moeilijk om via de klassieke manier van programmeren een robuste event-gedreven applicatie te bouwen. Functioneel Reactief Programmeren (FRP) is een alternatief dat gebruikt kan worden om event-gedreven systemen eenvoudiger te programmeren. FRP introduceert een declaratieve manier om event-gedreven applicaties te ontwikkelen waarbij de programmeur verlicht wordt van de taak om de toestand van het systeem manueel aan te passen.

We presenteren in dit werk een *Embedded Domain Specific Language* (EDSL) die een collectie FRP-primitieven aanbiedt om ingebedde systemen te programmeren. Deze maken het mogelijk een FRP-programma op te stellen dat uitgevoerd kan worden op een ingebed systeem. Omdat ingebedde systemen vaak slechts beschikken over beperkte hardwaremiddelen leggen we de focus op een efficiënte uitvoering van het FRP-programma.

We slagen hierin door met behulp van het Lightweight Modular Staging (LMS) framework een codegenerator te bouwen. Deze maakt het mogelijk om een FRP-programma om te vormen naar C-code dat gecompileerd kan worden voor het ingebed systeem.

Door de flexibiliteit van deze codegenerator slagen we er in om gespecialiseerde C-code te genereren die geschikt is voor het Sancus framework. Sancus is een beveiligd module systeem dat het mogelijk maakt modulaire applicaties op een veilige manier uit te voeren op een ingebed systeem zonder dat de volledige softwarestack van het systeem vertrouwd moet worden.

Thesis voorgedragen tot het behalen van de graad van Master of Science in de ingenieurswetenschappen: computerwetenschappen, hoofdspecialisatie Veilige software

Promotoren: Prof. dr. Ir. F. Piessens
Dr. D. Devriese

Assessoren: Prof. dr. T. Holvoet
Dr. J.T. Mühlberg

Begeleider: B. Reynders