FFI-RAPPORT

18/01982

Kan resonnering rundt sikkerhetsarkitektur automatiseres?

— en studie i sikkerhetsattributter og automatisk resonnering

Gudmund Grov
Elsie Margrethe Staff Mestl
Federico Mancini
Nils Agne Nordbotten

Kan resonnering rundt sikkerhetsarkitektur automatiseres?

– en studie i sikkerhetsattributter og automatisk resonnering

Gudmund Grov
Elsie Margrethe Staff Mestl
Federico Mancini
Nils Agne Nordbotten

Forsvarets forskningsinstitutt (FFI) 10. januar 2019

Emneord
Sikkerhetsarkitektur
Sikkerhetsanalyse
Analyseverktøy
Automatisering

FFI-rapport
18/01982

Prosjektnummer
1464

ISBN
P: 978-82-464-3134-5
E: 978-82-464-3135-2

Godkjennere
Nils Agne Nordbotten, *forskningsleder*
Jan Erik Voldhaug, *konstituert forskningssjef*

Dokumentet er elektronisk godkjent og har derfor ikke håndskreven signatur.

Opphavsrett

Sammendrag

En virksomhetsarkitektur er en strukturert tilnærming som beskriver relasjonen mellom organisering, prosesser og IT-løsninger. I denne rapporten anser vi sikkerhetsarkitektur som sikkerhetsaspektene ved virksomhetsarkitekturen.

Ett av målene med en sikkerhetsarkitektur er å gi et helhetlig bilde av sikkerheten, blant annet for å kunne gi bedre støtte til planlegging og utvikling av sikkerhet. En slik støtte krever også en evne til helhetlig resonnering rundt viktige sikkerhetsaspekter ved virksomheten. Vår erfaring tilsier at en slik resonnering i all hovedsak utføres manuelt i dag. Støtteverktøy brukes hovedsakelig av visualiseringsgrunner for å støtte den manuelle resonneringen eller for å kommunisere konklusjonen av resonneringen.

En slik visualisering er ofte veldig nyttig. Med noen små endringer i hvordan man modellerer virksomheten, kan man øke graden av automatisk resonnering rundt arkitekturen betraktelig. Disse endringene krever imidlertid mer konsistens og en klarere semantikk i modelleringen.

I denne rapporten introduserer vi en ny tilnærming som kan gi en slik støtte. Her utvider vi arkitekturmodeller med sikkerhetsattributter som gjør det mulig å inkludere ytterligere sikkerhetsrelatert informasjon. Disse attributtene kan blant annet brukes til å spesifisere krav som det må resonneres rundt, og å spesifisere egenskaper som kan brukes i en automatisert resonnering. Rapporten indikerer derfor hvilket potensiale slike automatiske resonneringsteknikker og sikkerhetsattributter har for sikkerhetsarkitektur spesielt, og virksomhetsarkitektur generelt.

Resonneringen foregår gjennom en tolkning av semantikken til arkitekturmodellene ved hjelp av formell logikk. Den formelle representasjon gjør det mulig å automatisere resonneringen rundt viktige sikkerhetsrelaterte egenskaper ved arkitekturen. Det er viktig å understreke at det formelle aspektet er gjemt for brukeren.

I rapporten demonstrerer vi potensialet for slike teknikker gjennom flere eksempler som viser ulike muligheter. Vi viser at dette lar seg implementere ved å utvide Enterprise Architect, som er et rammeverktøy brukt blant annet av Forsvaret. Alle eksemplene lar seg automatisere med denne utvidelsen. Til slutt ser vi også nærmere på andre potensielle anvendelsesområder for slike teknikker, som vi håper vil stimulere til diskusjon og debatt i det videre arbeidet.

Forsvaret anser i sin digitaliseringsstrategi virksomhetsarkitektur som et strategisk område og vi mener at arbeidet påbegynt her har stort potensiale for Forsvaret, blant annet for sikkerhetsstyring og endringsstyring av prosjektporteføljen.

Summary

An enterprise architecture is a structured approach that describes the relationship between organisation, processes and CIS solutions. For this report, we consider enterprise security architecture to consist of the security aspects of the enterprise architecture.

Enterprise security architecture aims to provide a holistic view of an enterprise's CIS security, which may, for instance, help to support the planning and development of new security capabilities and measures. Such support requires the ability to reason comprehensively about crucial security aspects of the architecture models. Our experience has shown that such reasoning is largely a manual activity, where the tool support is merely used for visualisation purposes, in order to both support the reasoning, and to communicate its conclusions.

Although visualisation has many merits, it does not exploit the full potential of automating the required reasoning process. Minor changes to how the enterprise security architecture is modelled can increase the level of automation considerably. These changes include, in particular, more consistent modelling and clear semantics of the modelling language.

In this report we introduce a new approach towards achieving the desired automated support. The language used for modelling is extended with security attributes, which are labels with additional security-related information. These attributes can be used to specify desirable security properties, as well as auxiliary properties used to support the reasoning process itself. The report thus provides an indication of the potential that automated reasoning techniques and security attributes have for use with enterprise security architecture in particular, and with enterprise architecture more generally.

The reasoning is achieved through a formal interpretation of the semantics of the architecture models by means of formal logic. This formal representation enables automated reasoning of crucial security-related properties of the architecture. The formal aspects and reasoning will remain hidden from the user.

The report demonstrates the potential for such reasoning techniques through several examples, illustrating a range of opportunities. Technical feasibility is shown through an extension to the Enterprise Architect toolset, which is able to automate the reasoning required by the examples. The report concludes with a discussion of other potential applications of such techniques, which we hope will stimulate debate on the next steps.

Innhold

Forord

Mesteparten av arbeidet beskrevet i denne rapporten har blitt utført sommeren 2018, i forbindelse med Elsie Mestls ansettelse som sommerstudent hos FFI. All implementasjon av verktøyet som er beskrevet her er utført av Elsie Mestl i denne perioden.

Rapporten gir en indikasjon på hvilket potensiale automatiske resonneringsteknikker og sikkerhetsattributter har for å støtte sikkerhetsarkitekter spesielt, og virksomhetsarkitekter generelt. Videre spekuleres det rundt andre potensielle anvendelsesområder for slike teknikker. Målet med rapporten er å danne en byggestein for videre arbeid og potensielt samarbeid, samt å stimulere til diskusjon og debatt om veien videre.

Arbeidet er primært rettet mot sikkerhetsarkitekter, men burde også være av interesse for virksomhetsarkitekter generelt. Vi tror også arbeidet er relevant for andre aktører som jobber mot, eller bruker, arkitektur. Dette inkluderer ledelsen, som for eksempel Forsvars-departementet, som kan få utbytte av dette i sitt planleggingsarbeid. Forskere innenfor anvendt logikk og resonnering vil også se nye mulige bruksområder for slike teknikker.

Takk til Audun Stolpe, Jan Erik Voldhaug, Ida Karine Grefslie og Ålov Synnøve Runde for tilbakemelding på rapporten.

Kjeller, 10. januar 2019

Gudmund Grov, Elsie Margrethe Staff Mestl, Federico Mancini og Nils Agne Nordbotten

1 Innledning

Digitaliseringsstrategien for Forsvaret (Forsvarsstaben, 2018) har virksomhetsarkitektur som et strategisk område hvor «*styring av digitalisering skal forbedres med omforent virksomhetsarkitektur*» og «*[virksomhetsarkitekturen] beskriver relasjonen mellom organisering, prosesser og IT-løsninger*».

Begrepet «virksomhetsarkitektur» har ulike meninger, og er ofte avhengig av situasjonen den beskriver (Ajer & Olsen, 2018). I dette dokumentet bruker vi ofte begrepet «arkitektur» for «virksomhetsarkitektur», og har samme forståelse av arkitektur- og sikkerhetsarkitektur som Mancini et al. (2017):

Definisjon (Arkitektur & sikkerhetsarkitektur (Mancini, Farsund, & Lillevold, 2017)**):** *Arkitektur* er «en strukturert tilnærming til planlegging og utvikling av komplekse systemer over tid, samt en formell beskrivelse eller detaljert plan av systemet på komponentnivå» og *sikkerhetsarkitektur* «skal sørge for at sikkerhet forankres i virksomhetens strategiske mål og behov og at også andre relevante aspekter enn de tekniske blir identifisert og integrert i planlegging og utvikling av sikkerhetsløsninger.»

Et mål med en sikkerhetsarkitektur er dermed å gi et helhetlig bilde av sikkerheten, blant annet for å kunne støtte bedre planlegging og utvikling av sikkerhetskapabiliteter og -tiltak. For at sikkerhetsarkitekturen skal kunne bidra med dette må en helhetlig resonnering rundt viktige sikkerhetsaspekter støttes. Med resonnering mener vi prosesser som fører til en slutning.[1] Aktuelle spørsmål man ønsker å besvare inkluderer:

S1. Er sikkerhetsmål, -krav og -behov realisert i IKT-systemene?

S2. Er alle tekniske (sikkerhets)løsninger nødvendige; utgjør de en funksjon i forhold til fastsatte krav og behov?

S3. Hva er konsekvensen dersom et system/tjeneste endres, oppdateres eller erstattes?

S4. Hva skjer dersom sikkerhetsmekanismer endres, oppdateres, erstattes eller fjernes? Er sikkerheten fortsatt god nok?

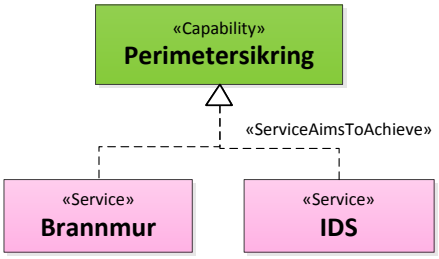
S5. Er sikkerheten godt nok forankret i en grundig risikovurdering og et oppdatert trusselbilde? Er risiko håndtert og restrisiko forstått?

S6. Hvordan er sikkerheten i forhold til et sikkerhetsmålbilde som endrer seg?

S7. Hva om trusselbilde, verdier eller risiko endres? Oppdateres sikkerheten fortløpende ved endring i risiko- og trusselbilde til et tilfredsstillende nivå?

[1] Se avsnitt 2.3 for en definisjon.

Et overordnet mål er en automatisert resonnering som kan hjelpe med å besvare slike spørsmål. Vår erfaring er at arkitektur primært brukes som støtte til manuell resonnering i form av visualisering[2], noe som skalerer dårlig ved økt størrelse og kompleksitet av modellene som skal analyseres. En slik visualisering er ofte et veldig nyttig verktøy[3]. Problemet slik vi ser det er at man ikke utnytter det fulle potensiale med arkitektur når den kun brukes for visualisering. Et eksempel på bedre utnyttelse er å bruke verktøy for automatisk støtte til selve resonneringen, som er tema for denne rapporten. Dette krever dog noen endringer i hvordan man modellerer.



Figur 1.1 Eksempel på en arkitekturmodell.

Figur 1.1 inneholder en veldig enkel diagrammatisk arkitekturmodell, representert i et rammeverk kalt NATO Architecture Framework (NAF). I denne rapporten brukes versjon 3 av dette rammeverket (NATO Consultation, Command and Control Board), som blant annet Forsvaret bruker. I rapporten vil vi bruke følgende navn på komponentene i diagrammene:

Definisjon (entitet & relasjon): Vi bruker begrepet *entiteter* for nodene og deres innhold i diagrammene, og *relasjoner* for kantene inkludert merkingen på kantene.

I Figur 1.1 ser vi en kapabilitet (en evne eller en ønsket evne) som heter *Perimetersikring.* Den realiseres via en «*ServiceAimsToAchieve*»-relasjon av en brannmurtjeneste (*Brannmur*) og en inntrengningsdeteksjonstjeneste (*IDS*). Dette eksempelet er relatert til spørsmål S1 og S2 i listen ovenfor, siden fokuset er på realisering av en kapabilitet, samt å vise hvilken kapabilitet en tjeneste understøtter.

Før vi går videre definerer vi noen begreper som vil bli brukt i denne rapporten:

Definisjon (Arkitekt, view, modell og formell modell): Vi bruker begrepet *arkitekt* for (menneskelige) aktører som utvikler arkitekturmodeller. Selv om det er andre aktuelle aktører (ledelse, utviklere, etc.) så fokuserer dette arbeidet mot selve arkitektene som faktisk utvikler modellene. En *modell* inneholder alle modellerte entiteter og relasjoner i en struktur, mens et *view* er en diagrammatisk fremstilling av ett eller flere aspekter ved modellen.[4] Merk at

[2] Visualiseringen støttes ofte gjennom verktøy. For eksempel bruker Forsvaret et verktøy kalt Enterprise Architect (EA) [`https://sparxsystems.com/products/ea/`].
[3] Blant annet er diagrammatiske representasjoner ofte nærmere våre kognitive prosesser enn tekstlige representasjoner (Jill & Simon, 1987), og dermed enklere å forstå.
[4] Dersom naturlig ut fra konteksten brukes også betegnelsen *diagram* for et view eller en modell.

nødvendigvis er det ikke en en-til-en-relasjon fra et view til en modell, siden et view for eksempel kan vise en abstraksjon av modellen. Hver entitet i en modell er unik, og identifisert basert på entitetsnavnet[5]. I Figur 1.1 er Perimetersikring, Brannmur og IDS entitetsnavn. En formell modell er en logisk (entydig) representasjon av en modell.

En stor del av resonneringen som gjøres i en arkitektur er sporing av krav/kapabiliteter gjennom relasjonene. For eksempel i Figur 1.1 kan vi spore både hvordan parametersikringen er realisert (ved å følge relasjonene «nedover») og hvorfor tjenestene trenges (ved å følge relasjonene «oppover» for å finne hvilken kapabilitet de realiser).

Vi mener at modellen i Figur 1.1 er en naturlig representasjon for en arkitekt med bakgrunn i system- eller programvareutvikling. For en arkitekt med en mer virksomhetsfokusert bakgrunn kan det være mer naturlig å ta utgangspunkt i kapabilitetene og dermed tegne pilene i motsatt retning. Siden det er flere arkitekter med ulik bakgrunn ender man ofte opp med en kombinasjon av slike måter å modellere på, som kan gi inkonsistens i modellen.

Et annet problem med modellen i Figur 1.1 er hvordan man skal tolke de to relasjonene. En tolkning er at perimetersikringskapabiliteten krever en form for *redundans* og derfor må realiseres gjennom begge disse to tjenestene (det vil si en OG-tolkning). I en slik tolkning vil det ikke være godt nok dersom bare én av tjenestene er på plass. En annen tolkning er at kapabiliteten kan realiseres gjennom én av tjenestene (det vil si en ELLER-tolkning), og i en slik tolkning vil det derfor anses som godt nok dersom bare én av tjenestene støttes.

I denne rapporten introduserer vi en ny tilnærming som kan hjelpe oss på veien mot å kunne svare på spørsmålene stilt i punktlisten fra begynnelsen av kapittelet. Denne tilnærmingen krever mer konsistens i selve modelleringen gjennom en felles og enhetlig forståelse av betydningen til de ulike entitetene og relasjonene i et diagram, samt en utvidelse av modelleringsrammeverket med *sikkerhetsattributter* som kan brukes for å gi ytterlige (sikkerhetsrelatert) informasjon, som for eksempel hvordan man skal tolke Figur 1.1[6]. I vår sammenheng har vi følgende definisjon av et sikkerhetsattributt:

Definisjon (Sikkerhetsattributt): Et *sikkerhetsattributt* er sikkerhetsrelatert informasjon som en entitet er merket med gjennom et strukturert språk.[7]

Vår tilnærming oppnås gjennom

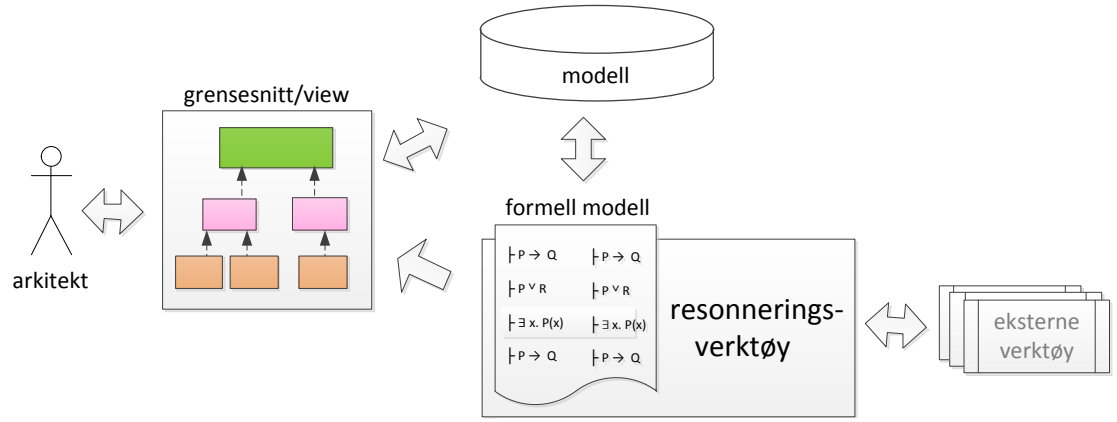
- en formell tolkning av semantikken til diagrammene (som det må være enighet om) og sikkerhetsattributtene, samt

[5] I EA-verktøyet er ikke navnet unikt, men hver boks har en annen unik identitet.
[6] I dette tilfellet kan man også gjøre det med grafiske virkemidler, for eksempel AND/OR-trær (som blant annet brukes i angrepstrær (Schneider, 1999)). Vi har ikke sett på hvordan attributter kan realiseres grafisk. Fra egen erfaring har vi inntrykk av at dette ofte vil redusere lesbarheten til diagrammene, men vi har kun anekdotisk bevis for dette.
[7] Kapittel 3 gir en definisjon av et slikt språk.

- bruk av formell logikk og automatisk resonnering for å kunne resonnere rundt, og svare på, aktuelle spørsmål[8].

Dette vil også muliggjøre en forklaring på både hvorfor sikkerheten er god nok ved å tolke resonnementet, og eventuelt hva problemet er dersom den ikke er god nok (ved å analysere hvor resonnementet stopper opp).

En slik formell tolkning og resonnering vil skje i bakgrunnen og arkitekten trenger ikke kunnskap om hverken logikk eller automatisk resonnering. Arkitekten må derimot forstå (og følge) semantikken til entitetene, relasjonene og sikkerhetsattributtene. Interaksjon med et verktøy som tilbyr slike resonneringskapabiliteter vil fortsatt foregå gjennom et vanlig grensesnitt til et verktøy (utvidet med sikkerhetsattributter) som vist i Figur 1.2.



Figur 1.2 Overordnet systemarkitektur for resonneringsrammeverket.

Figuren viser at arkitekten samhandler med et view av modellen gjennom et grensesnitt til det aktuelle verktøyet. Figur 1.1 er et eksempel på et slikt view. Modellen vil automatisk kodes i logikk, som gir en formell modell som resonneringsverktøyet bruker. Resonneringen kan medføre at modellen oppdateres (og dermed de grafiske modellene brukeren ser), eller at tilbakemeldinger (for eksempel forklaring) gis direkte til det grafiske grensesnittet uten at modellen oppdateres. Vi omhandler ikke endringer av modellen her (utenom visualiserings-endringer), men diskuterer dette under videre arbeid i kapittel 7.

[8] Enkle «resonneringer» kan allerede gjøres gjennom spørringer til databasen hvor modellen er lagret i Enterprise Architect, men dette er begrenset til hva som kan utrykkes i det underliggende spørrespråket (SQL).

1.1 Problemstilling, mål og begrensinger

Selv om vårt overordnete mål er å støtte arkitekturresonneringen for spørsmålene stilt i begynnelsen av kapittelet (S1–S7) gjennom automatisering, er ikke målet med arbeidet i denne rapporten å utvikle en løsning for å oppnå dette. Målet er derimot å vise at det er mulig å få mye bedre utnyttelse av verktøystøtten enn det som finnes i dag, som vil være en start og pådriver for å kunne hjelpe med å automatisere resonneringen rundt disse spørsmålene. Rapporten er i all hovedsak ment som et utgangspunkt for diskusjon om veien videre og muligheter for samarbeid, både innad i forsvarssektoren og med andre aktører. Mer spesifikt er målet

- å illustrere potensiale med automatisert resonnering for å øke utbytte av sikkerhets-arkitektur,
- en utvidelse av NAF-rammeverket med sikkerhetsattributter, og
- å vise at det er mulig å implementere slik automatisk resonnering med støtte for sikkerhetsattributter i et standard/COTS rammeverk.

1.2 Metode

Metoden brukt for å oppnå målet er eksempel-basert, hvor vi gjennom flere eksempler viser ulike aspekter av resonneringen. Sikkerhetsattributtene, og språket for å representere dem, er avledet fra utfordringene i de ulike eksemplene. Teknisk gjennomførbarhet dokumenteres ved å implementere en prototype, som støtter alle eksemplene.

For å bedre fremstillingen er eksemplene små, enkle og minimale, samtidig som at de er illustrative for hva en sikkerhetsarkitektur kan inneholde. Det er verdt å merke at grunnet både forenklingen av eksemplene, og størrelsen på dem, kan en slik resonnering virke noe overflødig og kunstig siden man lett kan se problemet og løsningen. Dette er ikke noe vi ser som problematisk grunnet hensikten med rapporten, men vi diskuterer skalerbarhet i kapittel 7.

1.3 Rapportens oppbygging

Kapittel 2 inneholder relevant bakgrunn om modelleringsrammeverk og automatisk resonnering. I kapittel 3 introduseres et språk for sikkerhetsattributter, samt forslag til noen spesifikke sikkerhetsattributter som brukes i denne rapporten. Deretter diskuteres det hvordan automatisert resonnering fungerer i denne konteksten. Dette krever en mer formell/matematisk tilnærming, men alle disse detaljene er flyttet til vedlegget for å gjøre rapporten mer tilgjengelig for et bredere publikum. Kapittel 4 beskriver hvordan resonneringsrammeverket er implementert som et verktøy kalt AREA i Enterprise Architect, før vi gir eksempler på attributter og resonnering i kapittel 5. Disse eksemplene illustrerer ulike aspekter av resonneringen. I kapittel 6 introduserer vi noe relatert arbeid av andre. Vi konkluderer og diskuterer veien videre i kapittel 7.

2 Rammeverk og automatisk resonnering

Dette kapittelet gir relevant bakgrunn om modelleringsrammeverkene og automatisk resonnering som brukes i resten av rapporten.

2.1 NATO Architecture Framework (NAF) og Enterprise Architect

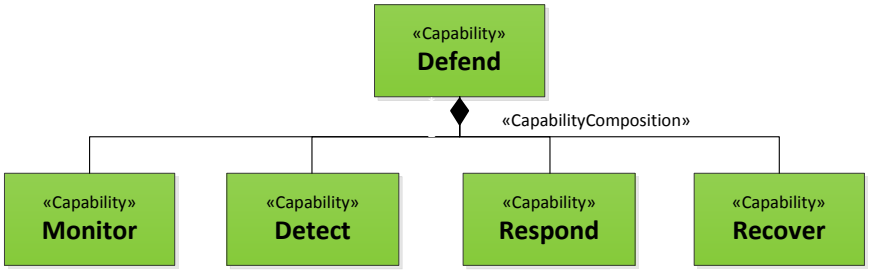
NATO Architecture Framework (NAF) er et rammeverk for arkitekturmodeller. Versjon 3, som brukes her, er organisert gjennom syv *views*, som igjen er delt inn i ca. 40 til 50 *subviews*, hvor et subview kan sees på som en diagramtype (NATO Consultation, Command and Control Board). Av de syv views representerer fire av dem ulike abstraksjonsnivå for arkitekturen. Disse nivåene er: kapabiliteter, operativ virksomhet¸ tjenester og systemer.

I tillegg er det ett view for standarder, ett for informasjonsstyring og ett for prosjektstyring. Merk at selv om diagrammene er delt inn i forskjellige views vil typisk et diagram også inneholde komponenter fra andre views. Dette er nødvendig for sporing. For eksempel inneholder diagrammet i Figur 1.1 både kapabiliteter og tjenester. Denne rapporten omhandler kapabiliteter, tjenester, systemer og prosjekter. Disse vil forklares i kapittel 5.

Enterprise Architect (EA) er et verktøy for design og analyse av blant annet UML (*Unified Modelling Language*) modeller utviklet av SparxSystems. I EA er NAF realisert som en dialekt av UML *klassediagram*. I kapittel 4 beskriver vi vår implementasjon av et verktøy kalt AREA for å støtte automatisk resonnering i EA. Til dette har versjon 10 av EA blitt brukt.

2.2 NATO CIS Security Capability Breakdown

NATO CIS Security Capability Breakdown er en definisjon/klassifisering av sikkerhetskapabiliteter til informasjonssystemer som kan brukes til å uttrykke styrkenivået til kapabilitetene. Den har vært brukt til å uttrykke nåsituasjon, målbilder og gapet mellom dem (Hallingstad, Gay, S. Gay, & Virvilis-Kollitiris, 2014).



Figur 2.1 «Defend» kapabilitet av NATO CIS Security Capability Breakdown (Hallingstad, Gay, S. Gay, & Virvilis-Kollitiris, 2014).

Figur 2.1 illustrerer underkapabilitetene for evnen til å forsvare informasjonssystemet (*Defend)*. Denne brytes videre ned i fire underkapabiliteter: evnen til å overvåke (*Monitor*), evnen til å detektere (*Detect),* evnen til å reagere (*Respond*), og evnen til å gjenopprette (*Recover).* Kapabiliteten er modellert i NAF, og styrkenivået til *Defend* vil være en aggregering av styrkenivået til underkapabilitetene (f.eks. gjennomsnittet av dem).

NATO CIS Security Capability Breakdown er her kun brukt som illustrasjon av resonneringen. Vi gir ytterlige detaljer i avsnitt 5.2. For en full introduksjon til rammeverket refererer vi til Hallingstad (2014).

2.3 Automatisert resonnering

For å kunne automatisere resonneringen trengs en passende underliggende representasjon som kan støtte dette. Enterprise Architect støtter (enkle) spørringer til databasen som lagrer modellene. Dette gjøres i SQL, som er et standardspråk for slike databasespørringer, men som er begrenset til enkle sporinger og er ikke rikt nok for den type resonnering vi er ute etter.

Vi baserer derfor resonneringen på logikk, som gir en entydig representasjon, samt muligheter for å bruke nye resultater og teknikker fra dette aktive forskningsområdet. Vi kjenner ikke til en ensbetydende definisjon av «*automatisert resonnering»*, men følgende mer generiske beskrivelse passer måten vi bruker det her:

Definisjon (Automatisk resonnering (AR)) Vi gir en noe generisk definisjon av *automatisk* som noe som blir utført av en applikasjon, med minimal interaksjon med mennesker og en ganske snever definisjon av *resonnering* hvor det involver bruk av formell logikk for å gi en slutning. Dette er likt definisjon brukt av Harrison (2009).

Merk at for videre arbeid vurderes en mer generisk tolkning av resonnering slik at det også inneholder forklaring, men dette er ikke noe som fokuseres på her. Andre lignende begrep som brukes for noe av det samme er *formell (automatisk) resonnering* og *logisk resonnering.* Mer generelt er dette en viktig del av et område kalt *formelle metoder*, hvor logiske/matematiske tilnærminger brukes for system- og programvareutvikling.

Fra vårt ståsted involverer bruk av slike teknikker følgende steg:

- En logisk representasjon/koding av arkitekturmodellene (som for eksempel i Figur 1.1), og eventuelle andre egenskaper utrykt gjennom sikkerhetsattributter. Det vil skje automatisk som vist i kapittel 3 og vedlegg A.

- En representasjon av egenskapene som vi vil resonnere rundt og trekke en slutning om. Dette kan uttrykkes gjennom sikkerhetsattributter, eventuelt kan vi resonnere rundt en «standardegenskap» som ikke må spesifiseres av arkitekten. Dette kan for eksempel være utledet fra lovtekst som vi illustrerer i avsnitt 5.4. I begge tilfeller vil kodingen til logikk være automatisk.

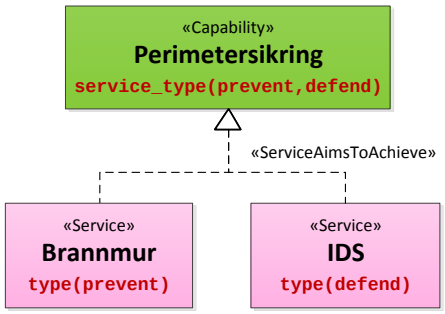
- Til slutt brukes teknikker fra automatisert resonnering for å trekke en slutning om de ønskede egenskapene holder for modellen, ideelt med en «forklaring» på hvorfor egenskapen holder eller ikke holder.

Et viktig punkt for brukervennlighet, og for at slike teknikker skal tas i bruk, er at alt det logiske skjer i bakgrunnen. En bruker vil tegne arkitekturmodellen som før, samt muligens bruke noen sikkerhetsattributter i tillegg. Dette vil da sjekkes med AR-teknikker og deretter gi et svar tilbake til brukeren (på et nivå som gir mening for brukeren).

I denne rapporten er fokuset på mulighetene slike teknikker kan gi, og ikke hvordan de fungerer. Vi går derfor ikke inn på de logiske detaljene, som vi isteden gjør tilgjengelig i vedlegg A for de som er interesserte i dette. Det er derimot viktig å forstå kompleksiteten til problemstillingen: for egenskaper av en viss kompleksitet er dette et *undecidable* problem, som betyr at algoritmer for å løse problemet ikke finnes. AR-teknikker bruker blant annet søk, abstraksjon og heuristikk for å kunne automatisere dette men vi kan ikke garantere at de vil gi et svar. For mer detaljer om AR refereres det til Harrison (2009).

3 Sikkerhetsattributter og automatisk resonnering

En måte å uttrykke «redundanstolkningen» fra Figur 1.1 er å utvide modellen med et språk for *sikkerhetsattributter*. Figur 3.1 oppdaterer modellen fra Figur 1.1 ved å merke entitetene med passende sikkerhetsattributter ifølge et sikkerhetsattributtspråk som vi definerer her.



Figur 3.1 Eksempel på redundans gjennom attributter.

I Figur 3.1 har *Brannmur*- og *IDS*-tjenestene et `type`-attributt som her betyr at tjenestene er henholdsvis av type `prevent` og `defend`. Dette gir ytterligere informasjon som kan brukes

for (automatisk) resonnering rundt sikkerheten. Målet med slike `type-`attributter er å gi en abstraksjon av formålet med disse tjenestene.[9]

*Perimetersikring*skapabiliteten er merket med et sikkerhetsattributt `service_type (prevent,defend)` som betyr at det må være én tjeneste av type `prevent` og én tjeneste av type `defend` som realiser denne kapabiliteten. Dette sikkerhetsattributtet er gyldig for dette eksemplet på grunn av relasjonene fra *Brannmur*- og *IDS*-tjenestene til kapabiliteten.

I avsnitt 3.1 introduserer vi syntaksen til dette språket, mens i avsnitt 3.2 gir vi semantikken til det ved å diskutere hvordan resonneringen fungerer.

3.1 Syntaks for sikkerhetsattributter

For å kunne bruke sikkerhetsattributtene i et AR-verktøy må de ha en form som verktøyet forstår. Her defineres syntaksen til dette språket. I det neste avsnittet gis semantikken til de ulike sikkerhetsattributtene som støttes i dette språket.

Syntaksen defineres i såkalt *Backus-Naur Form* (BNF): alternativer skilles av |, *r** betyr at *r* repeteres *0* eller flere ganger, *r+* betyr at *r* repeteres *1* eller flere ganger, *r?* betyr at *r* er valgfri, og *[m-n]* betyr alle symbol mellom *m* og *n* (inklusivt). Syntaks til et sikkerhetsattributt `A` er definert som:

```
A    ::= Id(Args)
Args ::= Arg | Arg,Args
Arg  ::= Id | Var | Num | A
Id   ::= [a-å][a-å|A-Å|0-9|_]*
Var  ::= [A-Å][a-å|A-Å|0-9|_]*
Num  ::= -?[0-9]+
```

Et attributt `A` har et navn (`Id`) og en liste med minst 1 argument, og hvert argument er separert med komma. Et argument er enten et navn, en variabel, et heltall, eller så kan argumentet være et attributt.

Syntaksen er inspirert av programmeringsspråket Prolog (Clocksin & Mellish, 1994)[10], hvor konstanter/navn har liten forbokstav, mens variabler har stor forbokstav. I eksemplene brukes bare variabler internt for resonneringen, men det er verdt å merke at det er ingen grunn til at man ikke kan bruke variabler som argumenter i et sikkerhetsattributt i fremtiden.

[9] Et opplagt alternativ er å gjøre dette grafisk ved å ha mer generelle tjenester/kapabiliteter for *Defend* og *Prevent* som *IDS og Brannmur* spesialiserer/realiserer. Denne informasjon kan da brukes i resoneringen istedenfor sikkerhetsattributtene. Det er derimot ikke alltid klart om det er lettest for brukere å fremstille slike «meta»-egenskaper gjennom sikkerhetsattributter eller diagrammatisk.

[10] Syntaksen er nærmere bestemt inspirert av tidligere arbeid av Lin et al. (2016) på et sammenlignbart grafisk språk, men for en helt annen kontekst.

Følgende sikkerhetsattributter er brukt i eksemplene i denne rapporten og er støttet i AREA-prototypen:

```
type(T)                 eal(N)              gap_color(N)
service_type(Ts)        value(E,V)          avg(Vs)
system_type(Ts)         status(S)           sum(Vs)
env(P)                  scb_value(N)        sub(A,B)
```

Vi diskuterer detaljene til disse, inkludert semantikken, i det følgende.

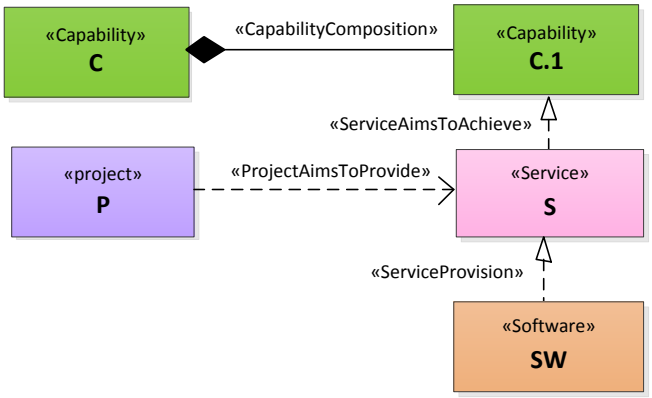
3.2 Semantikk for resonnering

For å kunne automatisere resonneringen rundt sikkerheten trengs en representasjon som støtter slik resonnering. I vårt tilfelle er denne representasjonen en formell logikk. Selv om resonneringen vil skje «under panseret», er det viktig å ha en forståelse av meningen til de forskjellige komponentene, samt å ha en viss forståelse av underliggende begrensinger. Uten disse er det ikke klart om egenskapene brukeren er interessert i faktisk er de det blir resonnert rundt.

Siden målet med rapporten ikke er hvordan selve logikken og resonneringen fungerer fokuserer vi her på de delene av semantikken som det er viktig at arkitekten forstår. Vi gir derfor en ganske uformell introduksjon, og refererer til vedlegg A for en mer formell beskrivelse av semantikken. Merk at dette bare er et forslag til semantikk. Videre diskusjon og arbeid kan medføre endringer.

3.2.1 Begrensninger ved bruk av NATO Architecture Framework

NAF har et stort antall forskjellige entiteter og et stort antall ulike relasjoner som binder sammen entitetene. Måten blant annet Forsvaret bruker NAF som en UML-dialekt i EA betyr også at standard UML notasjon kan brukes der. Vi støtter derfor bare en del av NAF, og Figur 3.2 viser de støttede entitetene og relasjonene.



Figur 3.2 Støttede NAF entiteter og relasjoner.

Som illustrert i Figur 3.2 er følgende entiteter støttet:

- *Kapabiliteter*, som har stereotype[11] «Capability» i NAF.
- *Tjenester*, som har stereotype «Service» i NAF.
- *Programvare*, som har stereotype «Software» i NAF.
- *Prosjekter*, som har stereotype «Project» i NAF.

To entiteter kan ha flere typer *relasjoner* mellom seg, og både NAF-relasjoner og UML-relasjoner kan benyttes. I tillegg er *retningen* til relasjonen avhengig av arkitekten, hvor noen leser en relasjon fra A til B, mens andre leser akkurat samme relasjon andre veien.

For vårt arbeid er det viktig å oppnå enighet om relasjonstype og retning. Vi støtter tre «typer» relasjoner mellom entiteter, som tilsvarer flere ulike relasjonstyper i NAF (som vist i Figur 3.2):

- *Realisering* av en entitet gjennom en annen entitet:
    - «ServiceAimsToAchieve» er en realiseringsrelasjon fra en tjeneste til en kapabilitet (dvs. med retning fra tjenesten til kapabiliteten).
    - «ServiceProvison» er en realiseringsrelasjon fra en programvare til en tjeneste.
- *Aggregering*, hvor en kapabilitet brytes ned i delkapabiliteter gjennom «CapabilityComposition». Denne relasjonen vil ha en svart diamantform i starten på relasjonen (nærmest relasjonen som brytes ned som vist i Figur 2.1).
- *Utvikling* av nye tjenester/kapabiliteter i et prosjekt gjennom «ProjectAimstoProvide»-relasjonen. Dette modelleres som en relasjon fra prosjektet til tjenesten/kapabiliteten.

Kapittel 5 inneholder eksempler på alle disse entitetene og relasjonene.

3.2.2 Støtteattributter

Sikkerhetsattributtene er inndelt i tre forskjellige klasser: støtteattributter, egenskapsattributter og visualiseringsattributter. Her gir vi en kort beskrivelse av de som vi har definert og brukt i eksemplene. Deres anvendelse kan best forstås gjennom eksemplene i kapittel 5.

[11] En stereotype er en utvidelsesmekanisme i UML, som her brukes for å klassifisere entitetene og relasjonene.

Støtteattributter brukes for å støtte resonneringen ved å definere visse egenskaper til en entitet, inkludert dens type eller begrensing av entitetens omfang. Følgende støtteattributter har blitt identifisert:

- `type(T)` betyr at entiteten er av type `T`. Siden `T` er en stor bokstav betyr det at det er en variabel og dermed en plassholder for en faktisk verdi. Vi har allerede sett eksempel på dette i Figur 3.1, hvor `T` var henholdsvis `prevent` og `defend`.

- `env(P)` begrenser omfanget til en entitet til det som er definert av **`P`**. For eksempel så kan dette brukes for å spesifisere at en programvareentitet kun er støttet for et gitt system. For eksempel betyr `env(system1)` at programvaren bare (kan) brukes i `system1`.

- `status(S)` gir status til en entitet i henhold til utviklingsfasen. Vi støtter kun to faser hvor `S` enten kan være `planlagt` eller `støttet`.

- `eal(N)` gir *Evaluation Assurance Level* (fra *Common Criteria*) til en entitet. Dette er et uttrykk for «tillitsnivået» fra en sikkerhetsevaluering og skal være mellom 1 og 7 (hvor 7 er best).

- `value(K,V)` brukes for å gi (konstanten) `K` verdien `V`. Det kan medføre at `V` må evalueres og det ikke kan være sirkulær avhengighet mellom forskjellige `E`. For eksempel kan man ikke ha både `value(x,y)` og `value(y,x)`. Dette sikkerhets-attributtet er uavhengig av hvilken entitet det er deklarert i.

- `avg(Vs)` returnerer gjennomsnittet av verdiene til (listen av verdier) `Vs`. Dersom listen inneholder en udefinert verdi vil ikke den verdien tas hensyn til.[12]

- `sum(Vs)` returnerer summen av verdiene til (listen av verdier) `Vs`. Dersom det er en verdi som er udefinert vil den ikke tas hensyn til.

- `sub(A,B)` trekker `B` fra `A` og returner resultatet.

3.2.3 Egenskapsattributter

Egenskapsattributter definer egenskaper av interesse som det skal resonneres rundt for å nå en slutning. To slike attributter støttes:

- `service_type(Ts)` betyr at for hvert element `T` i `Ts` må det være en tjeneste som realiser denne entiteten og tjenesten må ha type `T`. Den kan enten realiseres direkte, dvs. gjennom en direkte relasjon til den aktuelle entiteten, eller indirekte via andre entiteter. Figur 3.1 inneholder et eksempel på en «direkte» realisering, mens Figur 6.1 inneholder

[12] Et eksempel på at en verdi ikke er definert er dersom den er begrenset av `env` eller `status` som man ikke kan vise at holder.

et eksempel på en «indirekte» realisering (for `system_type`). I det siste tilfellet må man kunne følge en sekvens av realiseringsrelasjoner fra tjenesten av type `T` til den aktuelle entiteten.

- `system_type(Ts)` er som `service_type(Ts)` med den forskjellen at en entitet av type `T` som realiserer den, må være en programvarekomponent (på systemnivået i arkitekturen).

3.2.4 Visualiseringsattributter

Visualiseringsattributter brukes for å automatisere/bedre visualiseringen for brukere. De er derfor ikke noe som direkte trenger resonnering, men kan for eksempel brukes til å visualisere et resultat av en annen resonnering.

I vårt tilfelle har vi identifisert to slike attributter, som begge brukes for å fargelegge kapabiliteter basert på en form for «modningsverdi»:

- `scb_color(N)` brukes for å fargelegge en entitet i henhold til fargene som brukes på kapabilitetsnivået til *NATO CIS Security Capability Breakdown* (Hallingstad, Gay, S. Gay, & Virvilis-Kollitiris, 2014). Her må `N` være mellom 0 og 6. Dette resulterer i 6 forskjellige nivå/farger som vist i Figur 3.3.

Level	Meaning	Indicative ranking	Definition
0	Non existing	0/10	At this level, the capability, for all intents and purposes, does not exist.
1	Basic	2/10	The level of the capability can be globally considered embryonic.
2	Progressing	4/10	The level of the capability can be globally considered below average.
3	Established	6/10	The level of the capability can be globally considered above average.
4	Proficient	8/10	The level of the capability can be globally considered satisfying, and would be considered more than enough for most organizations. The capability is seen as a model for other organizations.
5	Excelling	9/10	The level of the capability can be considered "world leading".
6	Perfect	10/10	It is not expected that any organization will have a capability at this level.

Figur 3.3 Kapabilitetsnivå for NATO CIS Security Capability Breakdown (Hallingstad, Gay, S. Gay, & Virvilis-Kollitiris, 2014).

- `gap_color(N)` kommer også fra *NATO CIS Security Capability Breakdown*, men brukes for å fargelegge gapsnivået mellom målbildet og nåsituasjonen. Her er fargene hentet fra (Thorbruegge, Hallingstad, & Gay, 2017), og skal fargelegge entiteten som følger (i forhold til verdien av `N`):

- N = 0 (og mindre) tilsvarer at det ikke er noe gap og derfor brukes samme farge som på nivå (*level*) 4 av Figur 3.3 (grønn)
- N = 1 tilsvarer farge på nivå 2 (gul)
- N = 2 tilsvarer farge på nivå 1 (oransje)
- N = 3 (og større) tilsvarer stort gap og derfor brukes farge på nivå 0 (rød)

Merk at attributtene som er presentert her har blitt hardkodet i AREA-prototypen som beskrives i kapittel 4. Når man har opparbeidet en bedre oversikt over type attributter som skal utrykkes vil det være naturlig å utvide språket presentert her til å kunne definere nye attributter gjennom kombinasjoner av eksisterende og generiske attributter. Det gjøres i Lin et al. (2016).

Attributtene presentert her følger direkte fra eksemplene i kapittel 5. Vi har derfor utelatt naturlige attributter, som for eksempel å regne ut minimums- og maksimumsverdier.

4 AREA: verktøystøtte for Enterprise Architect

Ett av hovedmålene med dette arbeidet er å gi en indikasjon på potensiell bruk av AR og sikkerhetsattributter når det gjelder å støtte resonnering rundt sikkerhetsarkitekturer.

For å kunne dokumentere at dette er teknisk gjennomførbart har vi implementert en prototype *add-in* til Enterprise Architect. Her beskriver vi versjon 0.15 av dette verktøyet som vi har kalt *AREA* (Automatisk Resonnering for Enterprise Architect). Verktøyet støtter versjon 10 av Enterprise Architect, og alle eksemplene i det neste kapittelet er støttet av AREA.

Et viktig designprinsipp har vært at verktøyet skal endre arbeidsmåten i minst mulig grad. I avsnitt 4.1 beskriver vi hvordan verktøyet brukes, og i avsnitt 4.2 beskriver vi hvordan AREA virker og hvordan det har blitt implementert.

4.1 Hvordan bruke AREA?

For å kunne bruke AREA kreves det først noe konfigurasjon av EA rammeverket. Dette trenger man bare å gjøre én gang og det tar bare noen få minutter.

Når AREA er installert brukes EA som vanlig for modellering av virksomhetsarkitekturer. Når man definerer en ny entitet i EA, er det også vanlig å gi en beskrivelse av hva denne entiteten er. Dette gjøres typisk i vanlig (uformell) norsk tekst. I tillegg kan den merkes med

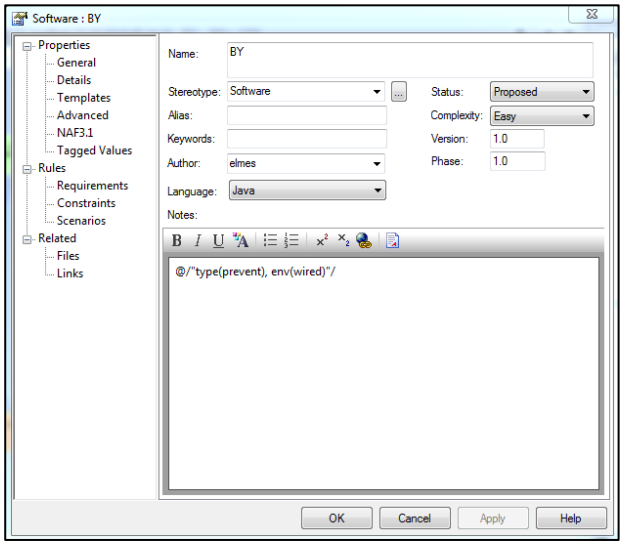
sikkerhetsattributter. Dette må være med formell tekst (med syntaks som gitt i kapittel 3) som må kunne forstås av AREA.

Det er flere måter man kunne ha implementert en slik støtte for sikkerhetsattributter i EA. Siden NAF er en form for UML klassediagram ville noe som heter *Object Constraint Language* (OCL) (Warmer & Kleppe, 1998) være et naturlig valg. Dette er et språk hvor man kan spesifisere egenskaper til klasser i UML klassediagrammer. Ved første øyekast virker det som at det passer godt til vårt formål, men problemet er at det er veldig fokusert mot bruk av klasser for objektorientert modellering, som ikke er måten den brukes for arkitekturprodukter. Dette vil kreve at flere nye elementer legges til for entitetene som ikke vil være naturlig for ønsket funksjonalitet.

Vi har istedenfor valgt å kombinere (de formelle) sikkerhetsattributtene med den uformelle teksten som brukes for å beskrive en entitet i dag. For å gjøre dette må den formelle teksten, som AREA må kunne forstå, separeres fra den uformelle beskrivelsen. Denne separasjon oppnås ved å «ramme inn» sikkerhetsattributtene med følgende syntaks:[13]

```
@/" <sikkerhetsattributter> "/
```

Figur 4.1 viser et skjermbilde av hvordan dette ser ut i EA.



Figur 4.1 Skjermbilde som viser representasjon av sikkerhetsattributter i EA.

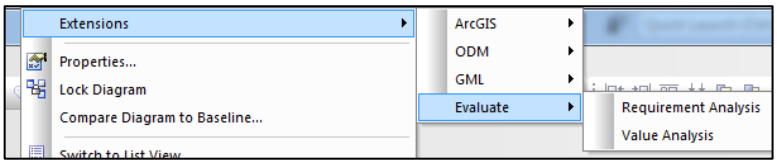
Når entitetene har blitt merket med sikkerhetsattributter må en bruker kunne starte resonnerings-verktøyet.

[13] Dette er inspirert av *antiquotations* som brukes i et bevisføringssystem kalt Isabelle (Wenzel & Chaieb, 2007).

Siden dette er en prototype har vi valgt å ha ekstra detaljert kontroll på interaksjonen med AREA.[14] Vi har derfor separert rammeverket inn i to hovedfunksjoner. Figur 4.2 viser et skjermbilde av hvordan disse funksjonene kan kalles i EA:

- Den ene menyen er kalt «*Requirement Analysis*» og brukes til å resonnere rundt egenskapsattributtene, og for å gi feilmeldinger dersom det er potensielle avvik.

- Den andre menyen er kalt «*Value Analysis*» og brukes til å beregne verdier til deklarerte variabler og å oppdatere (farge) entitetene i diagrammet i forhold til disse verdiene. Dette illustreres i avsnitt 5.2.



Figur 4.2 Skjermbilde av meny for AREA-funksjonaliteter.

Vi vil gi mer konkrete eksempler på bruk av begge funksjonene i det neste kapittelet.

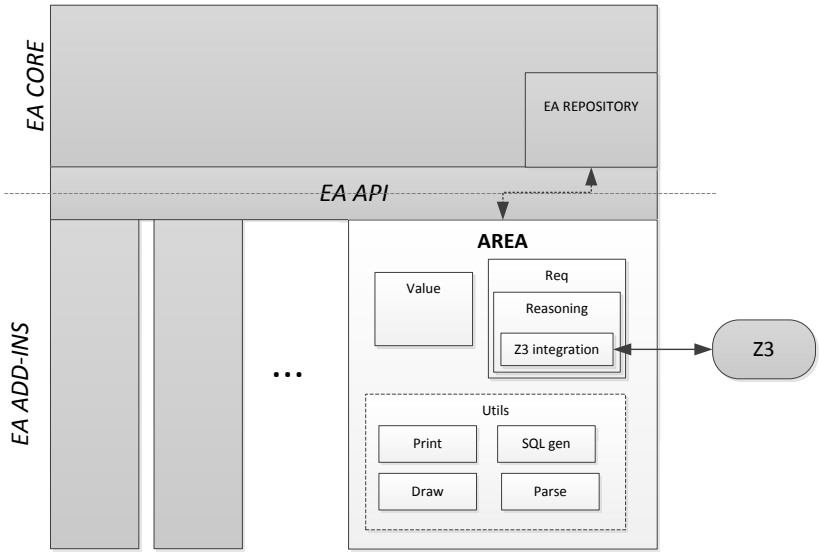
4.2 Hvordan fungerer AREA[15]?

AREA er implementert som en såkalt *add-in* til EA. Dette kan sees på som et tilleggsprogram som gir ytterligere funksjonalitet til EA.

Dette fungerer ved at EA tilbyr et grensesnitt (*Application Programming Interface* eller *API*) som utvikleren av en *add-in* får tilgang til gjennom et sett biblioteker som importeres i en egenstående applikasjon.[16] APIen brukes blant annet til å gi beskjed til EA når en *add-in* skal kalles og å gi tilgang til å oppdatere databasen med modellen og visualiseringen av et view. AREA vil kalles når de rette menyvalgene, som vist i Figur 4.2, velges av en bruker.

Figur 4.3 gir et overordnet bilde av komponentene til AREA og hvordan de kommuniserer med EA og andre programmer. AREA er implementert i programmeringsspråket C#.

[14] Alternative interaksjonsmodeller er å kjøre AREA i bakgrunnen hele tiden, eventuelt hver gang man lagrer, som er nærmere hva som er normalt for moderne utviklingsverktøy.
[15] Prototypen har brukt som utgangspunkt Bellekens tutorial *«Create your first C# Enterprise Architect addin in 10 minutes»*, tilgjengelig fra [goo.gl/c9LWFs] (23. juli 2018).
[16] Teknisk sett må dette kompileres som et *ActiveX COM*-objekt.



Figur 4.3 Programvarearkitektur til AREA.

I Figur 4.3 skiller vi mellom selve kjernen til EA (*EA CORE)* og tilleggsfunksjonaliteten som oppnås gjennom vår *add-in* (*EA ADD-INS*). APIen brukes for å støtte kommunikasjon mellom disse delene. En viktig del av kjernen er *EA REPOSITORY*, som blant annet inneholder selve databasen med arkitekturmodellen som AREA analyserer og oppdaterer.

Hovedfunksjonene nevnt ovenfor er representert av to moduler: «*Value*» og «*Req*». I tillegg er det en del funksjonalitet som brukes av begge disse modulene i en egen «*Utils*»-modul: «*Print*» brukes til å skrive ut (feil-)meldinger i konsollen til EA, «*Draw*» oppdater diagrammene som vises (for eksempel ved å fargelegge bokser), «*SQL gen*» brukes for å genere (SQL) spørringer til EA-databasen (i *EA REPOSITORY*), og «*Parse*» brukes til å parse/tolke sikkerhetsattributtene i modellentitetene.

Sikkerhetsattributtet `value(K,V)` brukes for å gi en konstant `K` verdi `V`. Når en bruker velger «*Value Analysis*» i menyen (se Figur 4.2) så kalles funksjonaliten i «*Value*»-modulen. Det første som gjøres er at verdiene til alle konstantene evalueres, oppdateres og lagres i en egen datastruktur. Deretter fargelegges nodene som er merket med `scb_color` eller `gap_color` som beskrevet i avsnitt 3.2.4 med hjelp av disse verdiene.

Selve resonneringen pågår i «*Req*»-modulen, og mer spesifikt i undermodulen kalt «*Reasoning*». Denne vil trigges når «*Requirement Analysis*» menyen (se Figur 4.2) velges av brukeren. Versjon 0.1 av AREA bruker ikke Z3, mens versjon 0.2 vil bruke Z3 for all resonneringen. I versjonen presentert her brukes en kombinasjon av intern resonnering og Z3, og har derfor fått versjonsnummer 0.15.

Den ene versjonen implementerer semantikken (som forklart i kapittel 3) til de forskjellige sikkerhetsattributtene og diagrammene direkte i koden.[17] Dette var versjon 0.1 av prototypen. Problemet med dette er at det er lett å gjøre feil, selv en liten feil kan ha store konsekvenser, og koden er vanskelig å vedlikeholde. Vedlikehold vil bare bli mer krevende i fremtiden med større og mer komplekse diagrammer og attributter.

I versjon 0.15 har vi derfor påbegynt en ny versjon som baserer seg på et eksternt verktøy for resonnering kalt *Z3* (De Moura & Bjørner, 2008). Dette er en «*state-of-the-art» SMT solver* som fungerer ved at vi gir den modeller og egenskaper formalisert i en logikk (som beskrevet i vedlegg A), og som svar vil Z3 si at egenskapen holder eller så vil den gi et moteksempel som gir en indikasjon på hvorfor egenskapen ikke holder. Dette arbeidet er ikke ferdigstilt så i denne versjonen støttes kun tilnærmingen beskrevet i avsnitt 5.4 i Z3. For de andre eksemplene brukes versjonen som er implementert direkte (versjon 0.1).

Før AREA kan brukes må den gjøres tilgjengelig og registreres i EA. Vi vil ikke gå inn i slike detaljer her. Mer tekniske detaljer om hvordan AREA er implementert finnes i vedlegg B.

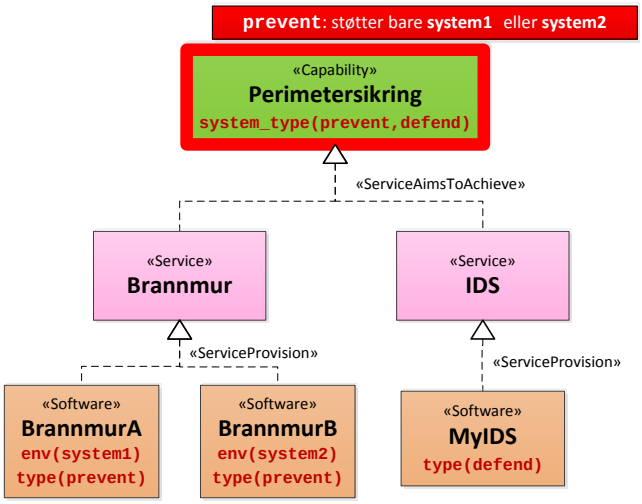
5 Eksempler på automatisk resonnering

Utviklingen av sikkerhetsattributtene, og den type resonnering beskrevet ovenfor, har vært empirisk og drevet frem av eksempler. Dette kapittelet gir flere slike eksempler og illustrerer samtidig bruk av resonneringen og sikkerhetsattributter. Vi gir flere ulike typer eksempler for å gi et bilde av omfanget av slike teknikker. Dette inkluderer kravsporing, gapanalyse, endringshåndtering og automatisk resonnering i forhold til lovfastsatte krav.

5.1 Kravsporing – redundans

Figur 3.1 ga et lite eksempel på *kravsporing* hvor et krav spesifiseres gjennom et sikkerhets-attributt i en kapabilitet og automatisk resonnering brukes til å vise at dette er oppfylt gjennom realiserte tjenester. Vedlegg A.4 gir de formelle stegene som brukes for resonneringen for dette eksempelet. Figur 5.1 gir en mer kompleks variant av en lignende problemstilling. Eksempelet tar for seg et scenario hvor en kapabilitet bare er støttet for et gitt miljø. Det kan for eksempel være en brannmur som bare er sikkerhetsgodkjent opp til BEGRENSET nivå (og kan derfor ikke brukes på systemer som krever høyere gradering), eller at brannmuren bare er støttet for visse systemer. Eksempelet her ser på det siste scenarioet. Som for Figur 3.1 er dette eksempelet relevant for spørsmål S1 og S2 fra introduksjonen.

[17] Grunnen til at vi valgte dette var at det var den raskeste måten å få noe til å fungere for å være sikker på at vi var på rett vei.



Figur 5.1 Kravsporing og «redundans» med programvarekomponenter som kjører i spesifikke miljø.

I eksempelet vist i Figur 5.1 må perimetersikringskapabiliteten realiseres gjennom programvarekomponenter (`system_type`) som er merket med type `prevent` og type `defend`. I modellen realiseres disse respektivt gjennom en *Brannmur*-tjeneste og en *IDS*-tjeneste.

Realisering av en programvaretype `defend` oppnås gjennom en programvarekomponent kalt `MyIDS`. Den er merket med `defend` og realiserer perimetersikringskapabiliteten gjennom en *IDS*-tjeneste. Denne egenskapen holder derfor.

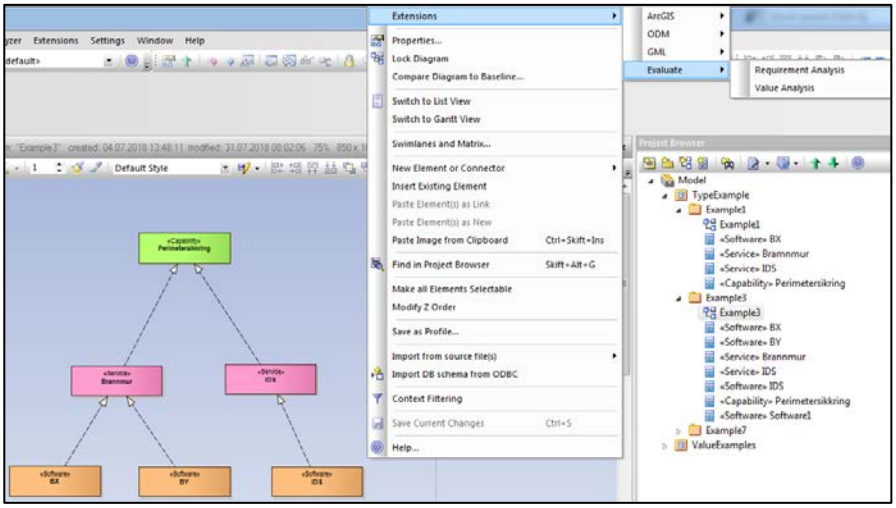
Realisering av en programvaretype `prevent` er mer kompleks. Den implementeres av to ulike brannmurer, `BrannmurA` og `BrannmurB`, og kan spores til perimetersikringskapabiliteten gjennom *Brannmur*tjenesten. Disse har derimot krav til det miljøet hvor de kjører, hvor `BrannmurA` kun implementeres i `system1` (gjennom `env(system1)` attributtet) og `BrannmurB` bare er implementert i `system2` (gjennom `env(system2)` attributtet).

Siden resonneringen bare kan vise at kapabilitetene er realisert for disse to systemene kan man ikke nå en slutning om at kapabiliteten er realisert for alle miljø (man vet for eksempel ikke om det er flere aktuelle systemer uten brannmurstøtte). Det gis derfor en alarm om at `prevent` for disse tilfellene muligens ikke er støttet. Hvis det er tilfelle kan en bruker ignorere denne alarmen (eventuelt legge til funksjonalitet i AREA for ikke å vise denne alarmen igjen).

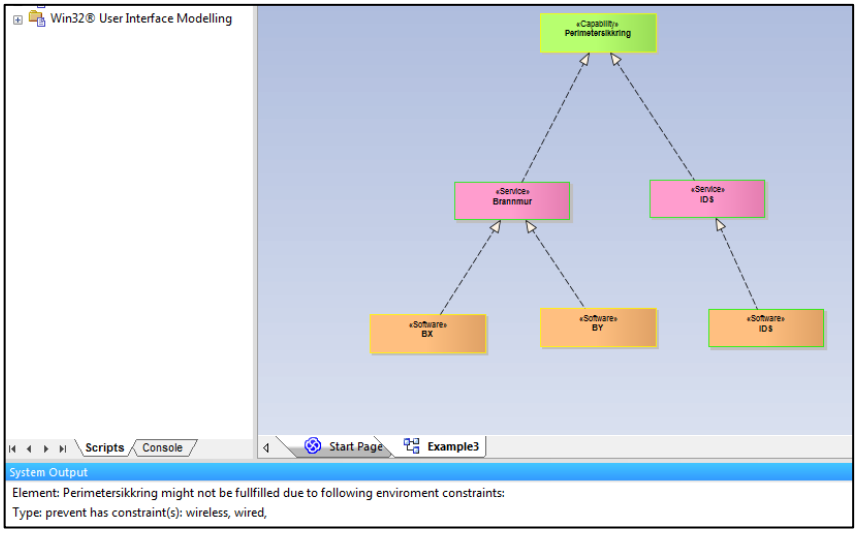
Observasjon 1. Automatisk resonnering og sikkerhetsattributter kan støtte sporing (og gi mer detaljer) mellom krav (på operativt nivå) og representasjon/implementasjon (på tjeneste- eller systemnivå). Dette kan også brukes til å gi begrensninger i form av miljø/kontekst/egenskaper som kreves i en realisering, eller som kan antas når man spesifiserer en egenskap gjennom et attributt. Disse kan for eksempel gi ytterlige detaljer enn at en kapabilitet bare er «delvis støttet», slik som vi har sett i flere modeller.

5.1.1 Analyse i AREA

Figur 5.2 og Figur 5.3 illustrerer hvordan AREA kan brukes for eksempel fra Figur 5.1.



Figur 5.2 Skjermbilde av EA før/mens bruker velger AREA funksjonalitet fra meny.



Figur 5.3 Skjermbilde av resultat fra Requirement Analysis.

Figur 5.2 viser hvordan AREA (*Requirement Analysis)* kalles fra menyen, og Figur 5.3 viser resultatet. Her fargelegges *Perimetersikrings*boksen med en oransje ramme for å indikere at egenskapen ikke holder, men *BX* og *BY*, som tilsvarer `BrannmurA` og `BrannmurB` i Figur

5.1, har en gul ramme for å illustrere at dette er potensiell kilde til alarmen. I tillegg gis noen meldinger i konsollen på bunnen av skjermbildet.[18]

5.2 Virksomhetssporing og gapanalyse

Kravsporing er tett knyttet opp mot utviklingsprosessen av systemer hvor målet er å spore krav gjennom design- og implementasjonsfasene.

I denne delen fokuseres det på selve virksomhetsarkitekturen for sikkerhet – mer spesifikt ses det på bruk av sikkerhetsattributter for å støtte arkitekten ved kommunikasjon av kapabilitets-nivå for *NATO CIS Security Capability Breakdown* (se avsnitt 2.2).

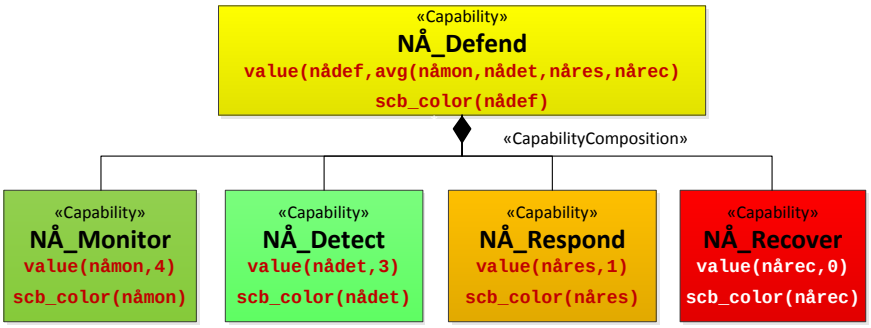
5.2.1 Aggregering av verdier

I Figur 2.1 så vi hvordan *NATO CIS Security Capability Breakdown* hjelper med «nedbrytning» av en kapabilitet inn i mindre, og mer håndterlige, delkapabiliteter.

For å kartlegge nivået til kapabilitetene må de analyseres. Hallingstad et al. (2014) anbefaler videre at nivået kvantifiseres mellom 0 og 6 og fargelegges for enkel visualisering av nivået (se Figur 3.3).

Delkapabilitetene på det laveste nivået, hvor det ikke er ytterligere delkapabiliteter, analyseres først og gis en verdi. Verdien til (del)kapabiliteten på nivået over vil da være en aggregering av verdiene til underkapabilitetene.

Dette er illustrert i Figur 5.4, hvor en oppdiktet nåsituasjonen er modellert. Merk at *NATO CIS Security Capability Breakdown* har flere lag som vi her utelater.



Figur 5.4 Virksomhetssporing – nåsituasjon.

Dette eksempelet illustrerer to nye funksjoner som AREA gir:

[18] Brukervennlige grensesnitt og interaksjoner har ikke vært fokus under utviklingen av AREA prototypen. Fokus har i stedet vært på å illustrere funksjonalitet. Se videre diskusjon i kapittel 7.

1. Automatisk fargelegging av entitetene gjennom `scb_color` visualiseringsattributtet.

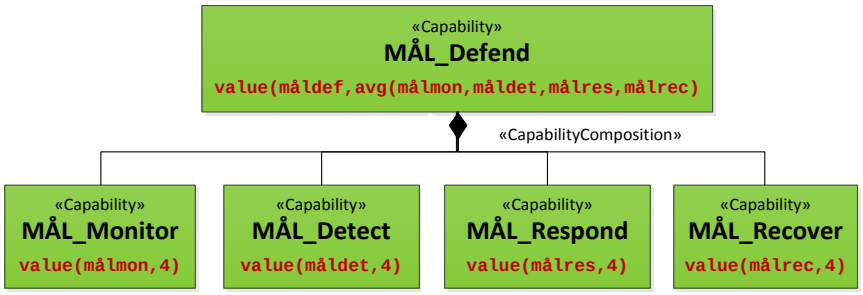
2. Automatisk oppdatering av farge/kapabilitetsnivå av entiteten ovenfor basert på kapabilitetsnivå til kapabilitetene under. Dette gjøres ved å regne ut gjennomsnittet av kapabilitetene under (`avg`) .

En (positiv) konsekvens av punkt 2 er at fargene vil oppdateres automatisk når en underkapabilitet endres (og AREA kalles igjen). For eksempel dersom kapabilitetsnivået til *NÅ_Recover* økes til fire (for eksempel på grunn av ny funksjonalitet) vil denne boksen (automatisk) bli grønn, samt at *NÅ_Defend* vil bli lysegrønn siden gjennomsnittet av underkapabilitetene vil økes fra to til tre.

Merk at navn på konstanter (satt ved hjelp av `value` attributtet) må være unike og kan derfor bare deklareres i én entitet i modellen.

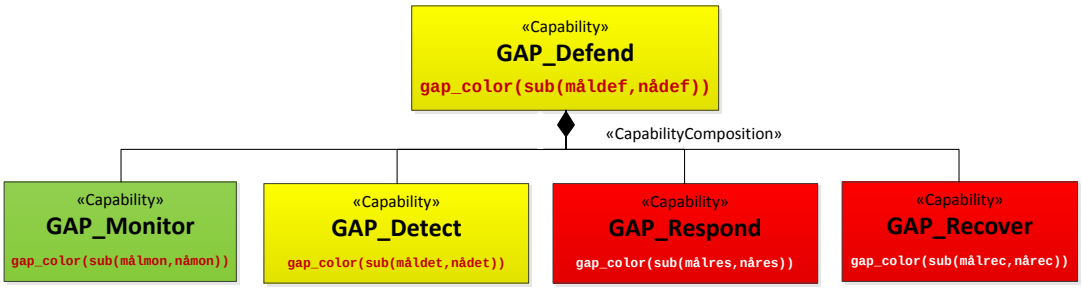
5.2.2 Sporing på tvers av diagrammer

Disse funksjonalitetene kan også brukes for å støtte en gapanalyse. For å illustrere, anta at det har blitt utviklet et målbilde hvor målet er at alle kapabilitetene skal være på nivå 4, som vist i Figur 5.5.



Figur 5.5 Virksomhetssporing – målbilde.

Vi kan da lage et view som regner ut, og fargelegger, gapet mellom målbildet og nåsituasjonen slik at det automatisk endrer seg dersom nåsituasjonen eller målbildet endrer seg.



Figur 5.6 Virksomhetssporing – gapanalyse.

Figur 5.6 viser et slikt «gapsview», ved å regne ut gapet mellom målbildet fra Figur 5.5 og nåsituasjonen som vist i Figur 5.4. Dette gjøres ved å trekke nåsituasjonsnivået fra målbilde-nivået mellom hver delkapabilitet. For eksempel så vil «gapsverdien» for overvåking (*GAP_Monitor)* kalkuleres ved å trekke nivået til *NÅ_Monitor* (4) fra nivået til *MÅL_Monitor* (4), som gir et gap på 0. Disse fargelegges så ved hjelp av `gap_color` visualiseringsattributtet (se avsnitt 3.2.4).

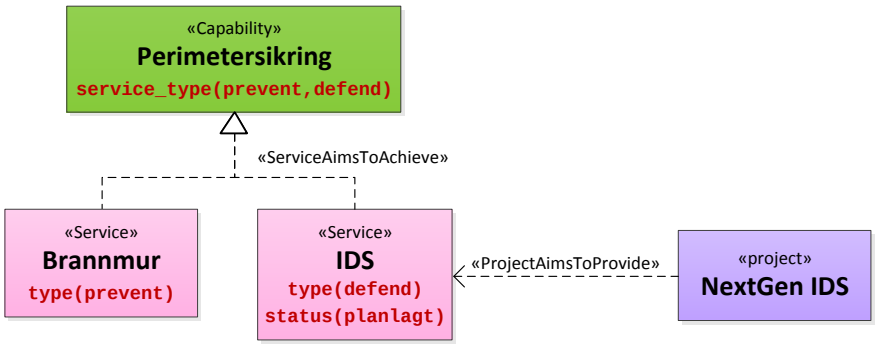
Observasjon 2. Kombinasjonen av sikkerhetsattributter og AREA kan gi en dynamisk sanntidsvisualisering av kapabilitetsnivået til *NATO CIS Security Capability Breakdown* (og muligens lignende «taksonomier» som kan kvantifiseres), både for nåsituasjon og gapsanalyse på tvers av view.

5.3 Virksomhetssporing og endringshåndtering

En viktig egenskap til virksomhetsarkitekturer er å kunne gjøre konsekvensutredninger av (potensielle) endringer i virksomheten. Her fokuserer vi på en virksomhet hvor utviklingen er prosjektbasert, som er tilfellet for Forsvaret, og et mål med arkitekturen er å kunne se virksomheten helhetlig og på tvers av prosjektene. Her vil vi illustrere hvordan AREA kan støtte, og automatisere, en konsekvensutredning for to enkle, men aktuelle, scenarioer hvor man vurderer å kansellere et prosjekt. Begge eksemplene illustrerer resonnering rundt spørsmål S4 i introduksjonen, hvor sikkerhetsmekanismer endres.

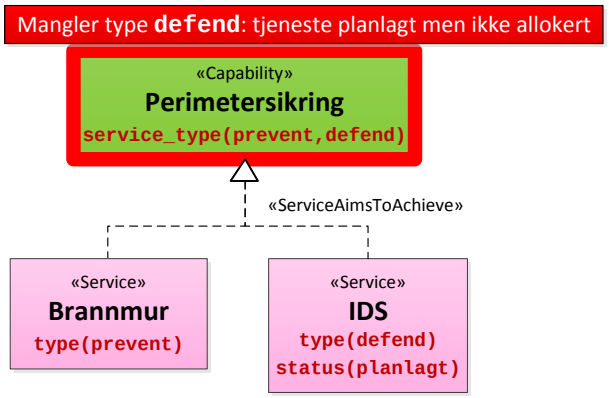
5.3.1 Endring og kravsporing

For det første scenarioet viser vi til eksempelet fra Figur 3.1. Dette oppdateres med at *IDS*-tjenesten ikke eksisterer, men er planlagt utviklet innenfor rammen av et prosjekt kalt «*NextGen IDS*». Figur 5.7 inneholder et view av et slikt scenario.



Figur 5.7 Endring & kravsporing – planlagt tjeneste.

Merk at sikkerhetsattributtet `status(planlagt)` brukes for å indikere at *IDS*-tjenesten ikke eksisterer per i dag, men at den er planlagt utviklet.[19] Dette attributtet begrenser omfanget av denne entiteten med den konsekvens at det må være en planlagt aktivitet som skal utvikle denne tjenesten for at AREA kan bruke de andre attributtene i entiteten i resonneringen (i dette tilfellet `type(defend)`). Her skal denne tjenesten utvikles i prosjektet «*NextGen IDS*», som er illustrert med en «*ProjectAimsToProvide*»-relasjon fra prosjektet til tjenesten. AREA vil derfor konkludere med at de tjenestetypene som kreves realisert i *Perimetersikring* faktisk er realisert, siden denne planlagte tjenesten har et prosjekt allokert til seg.



Figur 5.8 Endring & kravsporing – prosjekt med planlagt tjeneste kansellert.

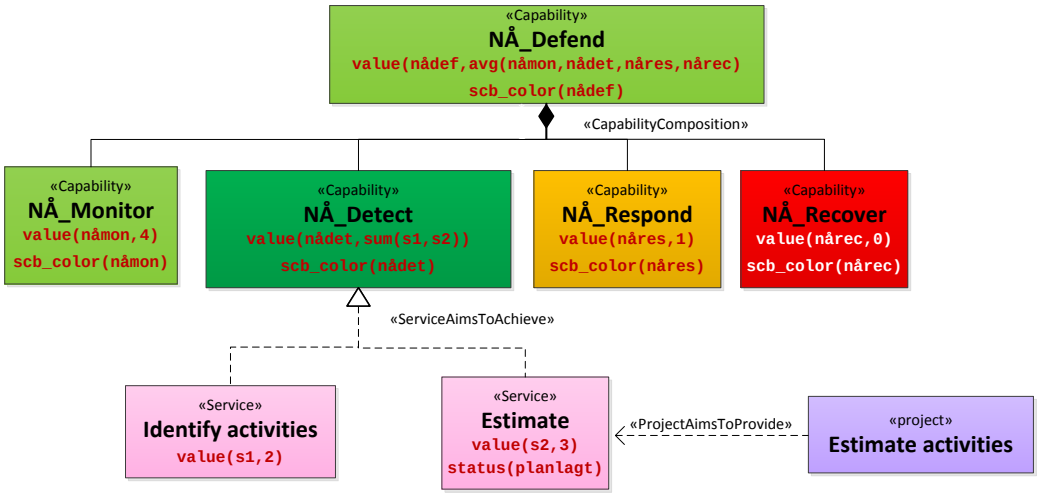
Det bestemmes så at prosjektet skal kanselleres. Dette vises i Figur 5.8. Her er det ikke lenger en planlagt aktivitet (prosjekt) som skal utvikle *IDS*-tjenesten. Som en konsekvens av dette kan

[19] I kapittel 7 diskuterer vi en utvidelse av status-konseptet hvor også tidsperspektiv kan spesifiseres og resonneres rundt.

ikke lenger AREA bruke `type(defend)` i resonneringen (på grunn av `status planlagt`)). AREA kan dermed ikke vise at det er en tjeneste av type `defend` som realiserer *Peremetersikring*skapabiliteten. AREA vil derfor gi en alarm til brukeren som illustrert i Figur 5.8.

5.3.2 Endring og kapabilitetsnivå

En lignende funksjonalitet kan også oppnås ved bruk av *NATO CIS Security Capability Breakdown*. Dette er illustrert i det fiktive eksempelet vist i Figur 5.9. Her er *Detect*-kapabiliteten blitt videre brutt ned til to tjenester: «*Identify activities*» og «*Estimate*». Disse vektlegges forskjellig ved utregning av kapabilitetsnivået ved at *Estimate* er viktigere. I dette eksempelet eksisterer ikke *Estimate*tjenesten, men den er planlagt utviklet i et prosjekt kalt «*Estimate activities*».

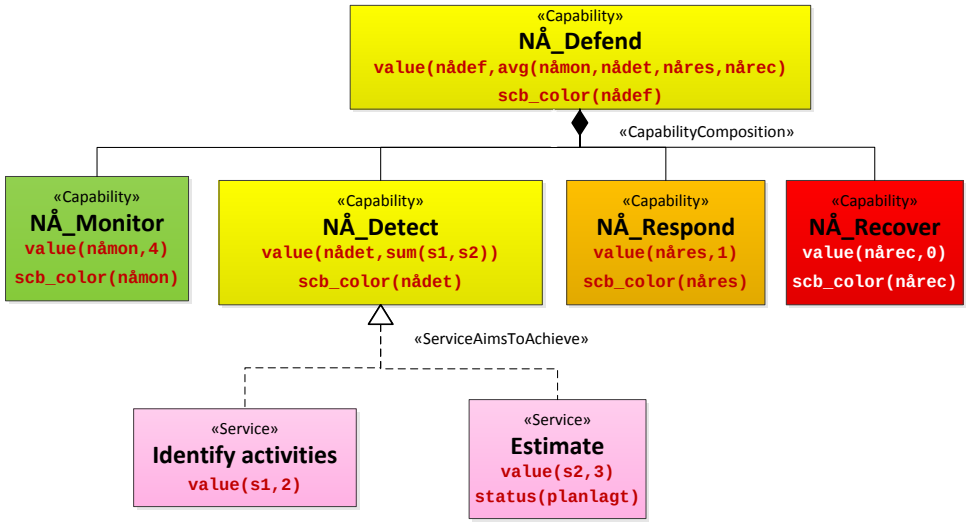


Figur 5.9 Endring og kapabilitetsnivå – planlagt tjeneste.

Observarsjon 3. Som illustrert i Figur 5.9 så kan «underkapabilitetene» (her representert som tjenester) vektlegges forskjellig (og fargelegges deretter).

Her bestemmes det igjen at prosjektet skal kanselleres og Figur 5.10 illustrerer konsekvensen av dette. Siden det ikke lenger er et prosjekt som skal utvikle tjenesten deaktiverer `status(planlagt)` attributtet `value(s2,3)` som gjør at `s2` er udefinert og vil derfor få en standardverdi som er `0`[20]. Dette betyr at kapabilitetsnivået til *NÅ_Detect* blir det samme som `s1`, som igjen vil redusere nivået til *Defend*-kapabiliteten. Som et resultat vil disse to kapabilitetene få en annen «nivåfarge».

[20] Slike standardverdier er på generelt grunnlag ikke en god ide, men vi har valgt dette for å gjøre bruken av sikkerhetsattributtene så enkel som mulig. Dette er noe vi vil vurdere nærmere i fremtiden.



Figur 5.10 Endring og kapabilitetsnivå – prosjekt med planlagt tjeneste kansellert.

Observasjon 4. AREA kan gi automatiske konsekvensutredninger ved virksomhetsendringer, som for eksempel kanselleringer av prosjekter. Dette må i dag gjøres manuelt. Det er flere måter man kan gi tilbakemeldinger, som for eksempel: ved en alarm, ved at en annen egenskap ikke holder, eller visualisering av nivåendring. Merk at dette er en følge av at relasjonen fra prosjektet til tjenesten fjernes (og ikke at selve prosjektet fjernes). Dette betyr at vi ville ha fått samme resultat hvis leveransene til prosjektet ble endret slikt at den aktuelle tjenesten ikke lenger ville ha blitt levert innenfor rammen av prosjektet.

En naturlig utvidelse av dette er å støtte tidsaspekter, hvor man også vil kunne resonnere rundt konsekvenser av å utsette prosjekter. Vi diskuterer dette videre i kapittel 7.

5.4 Bruk av formaliserte krav og lover

Graderte systemer, som kommer under Sikkerhetsloven, må sikkerhetsgodkjennes. Sikkerhetsloven, forskriftene og veiledninger utarbeidet av Nasjonal sikkerhetsmyndighet (NSM), har til dels klare føringer på hva som må være på plass av sikkerhet for at et informasjonssystem skal kunne godkjennes til et gitt graderingsnivå.

Mange av kravene er avhengig av konteksten til systemet, og/eller en underliggende risikovurdering. I tillegg til disse er det også flere entydige krav som må oppfylles. Et mål er å kunne automatisere verifiseringen av slike krav i en arkitektur. Som illustrasjon brukes eksempelet vist i Figur 5.11.

Figur 5.11 System med BEGRENSET gradering.

Dette eksempelet viser et program *S* som skal kunne håndtere informasjon med *BEGRENSET* gradering[21], og det må derfor kunne sikkerhetsgodkjennes for *BEGRENSET*. I eksempelet er gradering representert som en tjeneste[22].

En veiledning kunne for eksempel ha gitt følgende krav for en slik programvare[23]:

> *«All programvare som håndterer BEGRENSET informasjon skal ha et Evaluation Assurance Level (EAL) på minimum nivå fire»*

Slike generiske regler kan man formalisere direkte i logikk. Dersom de riktige forutsetningene holder så kan AREA automatisk forsøke å verifisere at reglene holder.

For dette eksempelet vil en forutsetning være en relasjon til *BEGRENSET*, og AREA må da kunne vise at *S* har EAL på nivå fire eller mer. I dette tilfellet antar vi at forutsetningen holder, men AREA kan ikke vise at *S* har det nødvendige EAL-nivået, og vil derfor gi en alarm til brukeren.

Dette kan naturligvis skyldes at *S* ikke har det nødvendige nivået, og må derfor evalueres. I så fall vil en slik alarm støtte selve utviklingsprosessen. Et annet alternativt er at den har et høyt nok EAL-nivå, men det har ikke blitt modellert. Figur 5.12 viser hvordan dette kan modelleres med et `eal`-sikkerhetsattributt. Her kan AREA vise at egenskapen holder og vil derfor ikke lenger gi en alarm til brukeren.

[21] For detaljer om gradering henviser vi til «Lov om forebyggende sikkerhetstjeneste (sikkerhetsloven)».
[22] Vi har sett flere eksempler hvor gradering er modellert på denne måten som er grunnen til at det brukes her.
[23] Dette er en forenkling/tilpasning av lignende krav fra Nasjonal sikkerhetsmyndighets «Veiledning i grunnleggende sikkerhetsarkitektur og -funksjonalitet for PARTISJONERT operasjonsmåte» (Nasjonal sikkerhetsmyndighet, 2006): «*Tillitsnivået til partisjonsmekanismer skal da i størst mulig grad tilsvare Common Criteria (CC) EAL4. Hver partisjon betraktes som et fellesnivåsystem. Det kan føres gode argumenter for at fellesnivåfunksjonalitet med CC EAL4 tillit tilfredsstillende kan beskytte data mot tilgang fra ikke-autoriserte brukere – særlig når partisjonene separeres av en EAL4-sertifisert brannmur.*»

Figur 5.12 System med BEGRENSET gradering og merket med EAL sikkerhetsattributt.

Observasjon 5. Ved å formalisere entydige krav – hentet fra lover, regler eller veiledninger – direkte i logikk kan AREA automatisk resonnere uten å kreve ytterligere spesifikasjon fra arkitekten, og gi tilbakemelding om disse kravene er oppfylt i modellen.

Det er viktig å forstå at en slik formalisering er både utfordrende og krevende og at det vil anbefales ytterligere utredninger for å undersøke om dette kan la seg gjøre i praksis. En formell tolkning av eksempelet over er gitt i vedlegg A.5 som gir et bedre bilde av utfordringene. Merk at dette er det eneste eksempelet hvor Z3 har blitt brukt av AREA.

6 Relevant arbeid

Her diskuterer vi arbeid med automatisk resonnering i arkitektur og sikkerhetsattributter hver for seg. Vi har brukt NAF som modelleringsspråk i arbeidet, hovedsakelig siden dette brukes av Forsvaret. Et alternativt modelleringsspråk er *ArchiMate*[24]. Vi har inntrykk av dette er mer brukt, og mye av litteraturen vi har sett på baserer seg på dette språket.

6.1 Automatisk resonnering

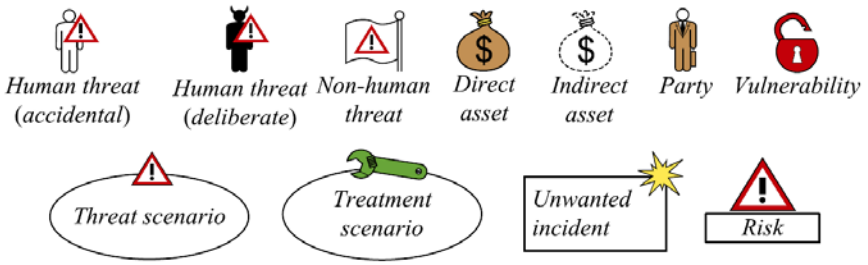
Sunkle et al. (2013) definer en virksomhetsbasert *ontologi*[25], som ArchiMate-modeller kan oversettes til. Dette muliggjør bruk av ontologiverktøy for resonnering, som inkluderer konsistenssjekking, verifisering, regler og spørringer. Sunkle et al. (2013) fokuserer på resonnering for to analysetyper. I «*change impact analysis*» analyseres effekten av en endring i en del av virksomheten på andre deler av virksomheten, inkludert mulige dominoeffekter. Dette er likt det vi kalte «virksomhetssporing». Den andre analysen er «*landscape mapping*», som forsøker å gi et overordnet bilde til ikke-tekniske aktører. Her brukes en komponeringsoperator (*composition operator*), utviklet av van Buuren et al. (2004), som abstraherer bort mellomliggende konsepter for relasjoner. Dette minner om en teknikk vi bruker i

[24] ArchiMate er et modelleringsspråk utviklet av *the Open Group*: [`http://www3.opengroup.org/subjectareas/enterprise/archimate`].
[25] En ontologi kan sees på som en formell representasjon av konsepter i et domene, inkludert egenskaper til, og relasjoner mellom.

resonneringen[26], men de «løfter» dette til selve viewet av en modell. Dette er noe vi kan vurdere å gjøre blant annet for å generere forklaring, som vi diskuterer i det neste kapittelet. En annen forskjell er at vi fokuser på sikkerhetsaspekter i arkitekturen, som ikke er tilfelle for Sunkle et al. (2013). Dalton et al. (2006) anbefaler bruk av formell logikk i sikkerhetsarkitektur og foreslår en tilnærming basert på *Petri nets*. Denne publiseringen er fra 2006 og det virker ikke som at arbeidet har blitt tatt videre.

I EA-verktøyet representeres NAF som en type UML-diagram. Flere har jobbet med resonnering rundt sikkerhet i UML. *UMLSec* er en utvidelse av UML med støtte for å modellere sikkerhetsaspekter, inkludert en formell semantikk (Jürjens, 2002). Disse kan sjekkes gjennom en teknikk kalt *modellsjekking* (Jürjens & Shabalin, 2004). Modellsjekking er en algoritmisk metode som man kan sees på som en komplett måte å teste alle mulige kombinasjoner og konfigurasjoner.[27] I UMLSec kan man også modellere trusselscenarioer, som man da kan bruke i resonneringen. Dette kan også gjøres gjennom *abuse cases* (McDermott & Fox, 1999) *og misuse cases* (Guttorm & Opdahl, 2005). For «*safety*»-egenskaper (dvs. utilsiktede hendelser) er arbeid med *accident scenarios* (Murali, Ireland, & Grov, 2015) relevant siden dette arbeidet har en mer formell tilnærming en både *abuse* og *misuse cases*.

I *CORAS*-metoden for risikoanalyse (Lund, Solhaug, & Stølen, 2010) brukes fem ulike typer diagrammer. Disse er verdidiagrammer, trusseldiagrammer, risikodiagrammer, risiko-behandlingsdiagrammer, og overordnede risikobehandlingsdiagrammer. Disse diagrammene bruker entitetene vist i Figur 6.1. Ulike diagrammer bruker overlappende undergrupper av entitetene vist i figuren. Diagrammene har blitt gitt en formell semantikk (Dahl, Hogganvik, & Stølen, 2007).



Figur 6.1 Entiteter/symboler brukt i risikomodelleringsspråket CORAS (Lund, Solhaug, & Stølen, 2010).

STRIDE-metoden for trusselmodellering (Shostack, 2014) kan også sees på som diagrambasert, men definerer ikke et eget diagram. Her utvides istedenfor eksisterende diagram (med fokus på flytdiagrammer) med «tillitsgrenser» som brukes for trusselmodelleringen.

SecureUML er en lignende tilnærming til UMLSec, med fokus på tilgangskontroll (Brucker, Doser, & Wolff, 2006). Der oversettes sikkerhetsaspektene til *Object Constraint Language*

[26] Spesifikt gjelder dette transitiviteten til `realise`-relasjonen. Se vedlegg A for forklaring.
[27] For mer detaljer om modellsjekking refererer vi til Clarke et al. (1999).

(Warmer & Kleppe, 1998), som har en definert semantikk, og interaktiv bevisføring brukes for resonneringen.

En utfordring med å utvikle en enhetlig semantikk for modelleringsspråk for en sikkerhets-arkitektur er at vår erfaring per i dag tilsier at ulike brukere har sin egen tolkning av språket, som kan gjøre det vanskelig å omforenes om en felles og enhetlig mening. Slike modelleringsspråk for arkitekturer har derfor ikke blitt gitt en formell semantikk. For eksempel sier dokumentasjonen til ArchiMate[28]:

> «*No formal, operational semantics has been defined for these relationships, because implementation-level languages such as BPMN and UML differ in their execution semantics and the ArchiMate language does not want to unduly constrain mappings to such languages.*»

Man har sett et lignende problem for utforming av en enhetlig formell semantikk for UML (Broy & Cengarle, 2011). Selv om en enhetlig semantikk er essensiell for vårt videre arbeid med resonnering, mener vi at mangel på det har betydning utover vårt arbeid. For å illustrere brukes ofte et view av en modell for kommunikasjon. Dersom man ikke har en enhetlig tolkning av hva viewet faktisk viser, må man også kombinere det med en forklaring av hvordan det skal tolkes. Man kan da spørre seg om hvilken hensikt arkitekturen egentlig har. Merk dog at det også er rom for å gi ulike tolkninger av visse konsepter i semantikken og ta hensyn til dette i resonneringen.

Teknikker for automatisk resonnering er også brukt mer generelt for resonnering og verifisering av sikkerhetsaspekter for nettverk og systemer. For dette området kan vi ikke være uttømmende, og nevner derfor bare noe arbeid. *Amazon Web Services* har nå en *Automated Reasoning Group* som bruker og utvikler verktøy for å bedre sikkerheten til deres produkter gjennom automatiserte resonneringsteknikker. De har blant annet utviklet verktøyene *Zelkova* og *Tiros* for å resonnere rundt autentisering og tilgang.[29] Verktøyene brukes blant annet for å kunne håndtere komplekse konfigurasjoner som er for vanskelige å analysere manuelt; en lignende utfordring vil man trolig få for sikkerhetsarkitekturer i komplekse systemer. En oversikt over Amazons arbeid er utarbeidet av Cook (2018)[30]. *SecGuru*-verktøyet for *Microsoft Azure* (Jayakaraman, Bjørner, Outhred, & Kaufmann, 2014) bruker Z3 for å resonnere rundt nettverkspolicyer (rutingstabeller, brannmurer, nettverkstilgang, BGP, osv.). Dette kan hjelpe med å unngå misskonfigurasjon av tilgangslister (Bjørner & Jayaraman, 2015), og brukes blant annet for validering av at policyer er korrekt implementert. Dette kan sammenlignes med vår kravsporing, men i en annen kontekst, samt at SecGuru kan også overvåke (og resonnere rundt)

[28] [http://pubs.opengroup.org/architecture/archimate3-doc/chap05.html].
[29] Amazon bruker terminologien «*provable security*» for bruk av logikk og resonnering til å bedre sikkerheten. Dette må ikke forveksles med meningen dette begrepet har innenfor kryptografi, hvor det å «bryte» sikkerheten (kryptografien) i systemet er ekvivalent til å løse et vanskelig problem.
[30] Vi kan også anbefale to YouTube-videoer: En fra Amazon Web Services [https://m.youtube.com/watch?feature=youtu.be&v=JfjLKBO27nw9] og en fra *Bridgewater* [https://m.youtube.com/watch?feature=youtu.be&v=gJhV35-QBE8].

de faktiske kjøremiljøene. SecGuru støtter også konsekvensanalyse ved endringer av tilgangslister (*change-impact analysis*), som kan sammenlignes med vår virksomhetssporing.

Av mer direkte relevans for forsvarsektoren er *DARPAs*[31] *High Assurance Cyber-Military Systems* (*HACMS*) program. Her ble flere ulike resonneringsteknikker tatt i bruk for å oppnå målet med utvikling av sikker programvare for kjøretøy (Fisher, Launchbury, & Richards, 2017).

Det er verdt å merke at mye av arbeidet beskrevet ovenfor følger en lik tilnærming som oss, hvor selve resonneringen skjer «i bakgrunnen» ved å gi en formell semantikk til den underliggende modellen som da kan brukes for resonneringen.

6.2 Sikkerhetsattributter

Et sekundært bidrag i dette arbeidet er et språk for sikkerhetsattributter, som kan brukes til å uttrykke og konkretisere ytterligere egenskaper og krav. Dette er bare en begynnelse på et slikt språk, og vi forventer at både språket og de foreslåtte sikkerhetsattributtene vil endre seg over tid når vi har opparbeidet en bedre forståelse av hva som er ønskelig å utrykke for mer reelle eksempler enn de som er brukt her.

I UMLSec utvides UML med sikkerhetsspesifikke «stereotyper», «merkede verdier» og «begrensninger». Stereotypene har en lignende funksjonalitet som våre «egenskapsattributter», som brukes til å definere krav, mens «merkede verdier» virker som å ha lignende funksjonalitet som våre «støtteattributter». Vi har ikke utviklet en måte å definere spesifikke «begrensinger», men det er verdt å merke at vi har et rikere språk (gjennom argumenter til attributtene) så noe kan kodes direkte i attributtene. I både vår tilnærming og UMLSec spesifiseres sikkerhets-aspekter innenfor eksisterende grafiske komponenter og ikke som egne entiteter i diagrammet. I MODAF, som er et rammeverk fra det britiske forsvaret, annoteres også elementene med sikkerhetsinformasjon, men det er ikke klart hvordan dette gjøres.[32]

I SABSA-tilnærmingen for sikkerhetsarkitektur (Sherwood, Clark, & Lynas, 2005), introduseres det som kalles «forretningsattributter», inkludert attributter for sikkerhet. En taksonomi av eksisterende attributter er gitt i Figur 6.2. Man kan også definere nye attributter. Som i vårt tilfelle har disse attributtene blitt utviklet på en empirisk måte, basert på analyse av konkrete eksempler. I SABSA brukes attributter til å spesifisere hva som skal beskyttes og for å dokumentere forretningskrav i en normalisert form. Som illustrert i eksempelet nederst i Figur 6.2, så brukes slike attributter typisk som egne entiteter i diagrammet.

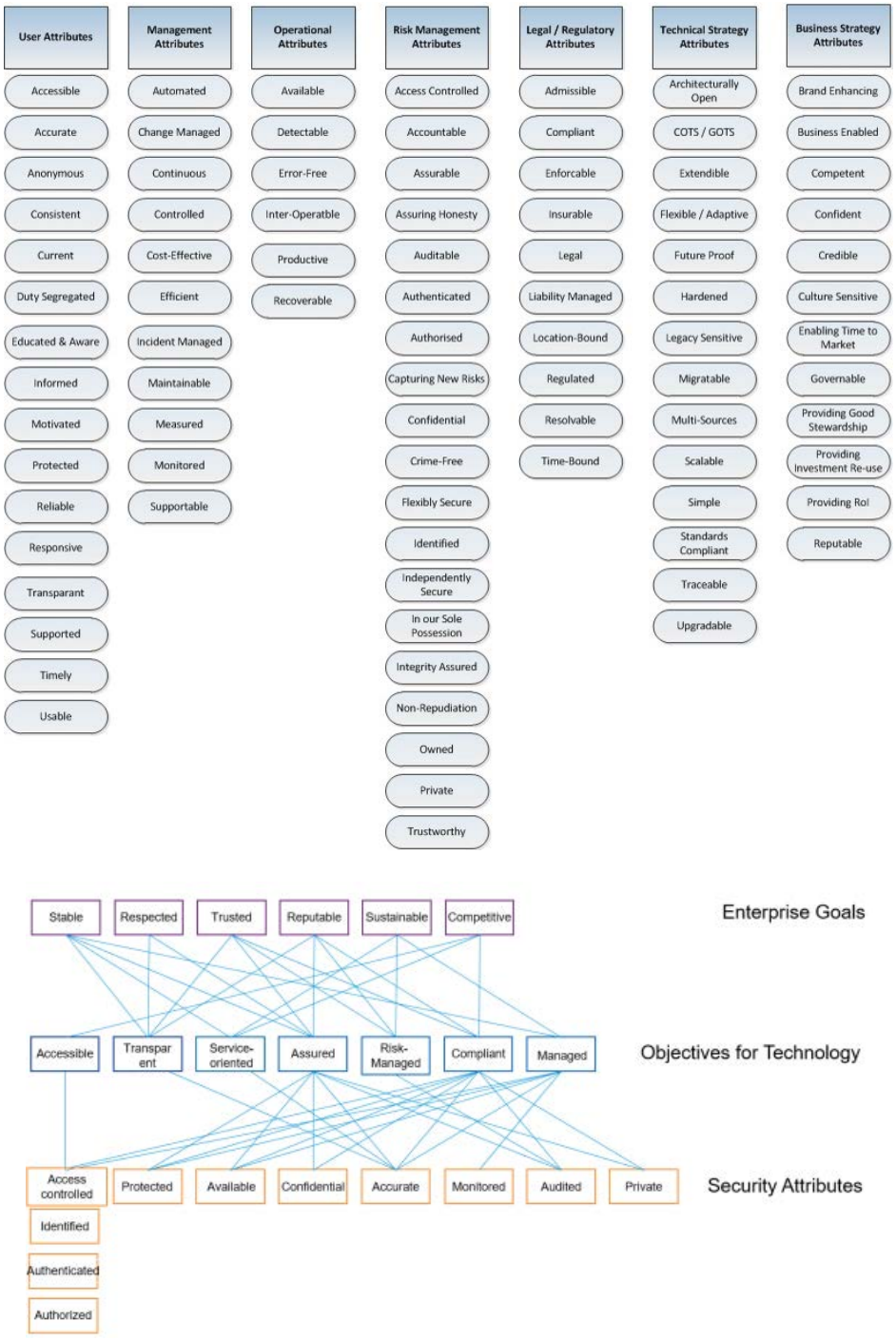
I en masteroppgave av van den Bosch ved universitetet i Twente (van den Bosch, 2014) defineres en sikkerhetsutvidelse av ArchiMate med fem nye konsepter, hvor et konsept (blant annet) kan sees på som en «type» entitet i et diagram. Disse konseptene er sårbarhet, trussel, risiko, sikkerhetsmekanisme og sikkerhetspolicy. Det er verdt å merke at verdivurdering ikke er

[31] *Defence Advanced Research Project Agency* (for amerikansk forsvarssektor)
[32] [https://www.gov.uk/guidance/mod-architecture-framework]

inkludert. Et konsept er mer generisk enn våre sikkerhetsattributter, men dette har noen likheter ved at for en spesifikk setting vil et konsept «instansieres» for denne settingen, slik vi kan instansiere sikkerhetsattributt gjennom deres argumenter. Egenskapsattributtene definert i avsnitt 3.2.3 kan sees på som en form for sikkerhetspolicy, mens støtteattributtene fra avsnitt 3.2.2 kan brukes for å merke sikkerhetsmekanismer. Vi har ikke sett på bruk av sikkerhets-attributter i forbindelse med sårbarheter, trusler eller risiko, og dette er noe vi derfor anser som videre arbeid.

Verken SABSA-attributtene eller ArchiMate-utvidelsen har en formell semantikk som muliggjør resonnering. Videre har vi et rikere språk: SABSA-attributtene er proposisjonelle og kan ikke spesialiseres utenom å definere nye attributter, mens vi kan spesialisere attributtene gjennom argumentene. I ArchiMate-utvidelsen kan man oppnå en liten grad av slik strukturering av egenskaper, men denne tilnærmingen er ganske rigid og vil kun klassifisere et «attributt» i ett av de fem konseptene. Det er verdt å merke at selv om vi har definert et rikere språk for attributter, som gir flere muligheter, er det også fare for at attributtene blir for komplekse for arkitektene. Det er derfor viktig at de forankres hos arkitektene og at deres utvikling er brukerdrevet.

Fra et brukerperspektiv er det stor forskjell på måten man modellerer på i de ulike tilnærmingene. I vår tilnærming (som også brukes i UMLSec og SecureUML) merkes eksisterende objekter med attributter og det betyr at diagrammatisk/strukturmessig vil en slik merking ikke endre diagrammet. Vi kan derfor se på sikkerhetsutvidelsen som et språk som er «innenfor» entitetene i diagrammet. I SABSA-bruken av attributter fra Figur 6.2, samt ArchiMate-utvidelsen, er sikkerhetsattributter egne entiteter i diagrammet og diagrammet vil derfor endres. Vi har ikke vurdert hva som er den beste måten å gjøre det på. Noen vil nok foretrekke en ren diagrammatisk tilnærming, men det er også en fare for at diagrammene blir for kompliserte og detaljerte, som gjør dem vanskeligere å forstå. Merk også at gjennom en mer formell tilnærming kan man også bruke begge disse og transformere mellom dem gjennom graf-omskrivningsteknikker som vi diskuterer i kapittel 7.



Figur 6.2 Øverst: SABSA sikkerhetsattributter (Sherwood, Clark, & Lynas, 2005). Diagram hentet fra [`https://andywood.info/`].
Nederst: Eksempel på bruk fra Leron Zinatullins blogpost «SABSA architecture and design case study» [`https://zinatullin.com/2018/03/11/how-to-solve-a-business-problem-with-security-using-sabsa/`].

7 Konklusjon og veien videre

Vi har demonstrert et potensiale for bruk av en automatisk resonneringsteknikk for å støtte og automatisere resonneringen rundt sikkerhetsarkitekturer. Dette er en begynnelse på en mer langsiktig målsetting om å kunne resonnere mer automatisk om problemstillingene fra introduksjonen (S1–S7). I dette arbeidet har vi sett eksempler på S1, S2 og S4. Arbeidet er demonstrert gjennom flere eksempler som viser ulike muligheter for automatisk resonnering. Vi har også vist at dette lar seg implementere i eksisterende rammeverk gjennom en *add-in* for Enterprise Architect.

En slik tilnærming har også en del utfordringer, både med hensyn til sikkerhetsattributter, resonneringen, og dens påkrevde underliggende formalisering.

For sikkerhetsattributter er det viktig at de gir mening for brukeren og at språket de uttrykkes i er rikt nok for å spesifisere det som skal spesifiseres, uten å være for komplisert for brukeren.

Formaliseringen er avhengig av en enhetlig semantikk som har vist seg å være problematisk, i tillegg til valg av «riktig» måte å formalisere samt en del tolkningsutfordringer, som er diskutert i vedlegget. Det er heller ikke uvanlig at en bruker må gi hjelp/hint til resonneringsverktøyet i modellen, noe som er avhengig av type resonnering, egenskaper og selve modellen. Dette er ikke omhandlet her.

Det er også verdt å merke at selv om fokuset i vårt arbeid har vært på sikkerhet og sikkerhetsarkitektur er det fullt mulig at mye av det som er foreslått kan ha en rolle for andre aspekter av virksomhetsarkitekturer.

Arbeidet er i et tidlig stadium, med flere muligheter for videre arbeid. Det kan også muliggjøre andre aktiviteter, og vi separerer derfor diskusjonen om veien videre fra mer muliggjørende aktiviteter i det følgende.

7.1 Videre arbeid

Her foreslår vi noen naturlige neste steg for dette arbeidet.

Vurdering av resonneringsteknikker. Kapittel 6 inneholder en litteraturstudie som diskuterer andre tilnærminger, som blant annet ontologi og modellsjekking. Et annet spørsmål er om NAF, som vi har brukt siden Forsvaret bruker det, er riktig modelleringsspråk å basere arbeidet på, eller om for eksempel ArchiMate har mer potensiale. Et naturlig neste steg vil være en mer detaljert studie for å kartlegge arbeid som er gjort, eller er under arbeid, for å hjelpe med å ta valg om språk og tilnærming for vårt videre arbeid. Slike valg vil være avhengig av egenskapene som er av interesse, og identifikasjon av disse vil kreve mer realistiske eksempler.

Realistiske, reelle og komplekse eksempler. Vi har illustrert vår resonneringsteknikk på små og illustrative eksempler. Disse egner seg godt til å forklare tilnærmingen og potensialet, selv om resonneringen strengt tatt ikke er nødvendig. Et neste steg vil være å utprøve tilnærmingen vår på større og «*real-world*»-problemstillinger, som for eksempel Forsvarets arkitektur-modeller. I tillegg til å se hvordan teknikken skalerer, vil det hjelpe med å finne ut av hvilke spørsmål som er viktige og for hvilke aktører. Amazon Web Services bruk av automatisk resonnering (Cook, 2018) har vist seg å være veldig verdifull når konfigurasjoner/systemer ble veldig komplekse og umulig å sjekke manuelt. Vi har også sett behov for et slikt verktøy i et arkitekturdrevet arbeid som omhandler Forsvarets IKT-infrastruktur.[33]

«Rikere» diagrammer og egenskaper. Et resultat av større og mer reelle eksempler er trolig at en større del av modelleringsspråket må støttes og at andre (og muligens mer komplekse) egenskaper/sikkerhetsattributter enn de som er beskrevet her må støttes og formaliseres. Et utgangspunkt for å identifisere en passende undergruppe av NAF er «NAF-kjernen» definert av Hansbø et al. (2013). I tillegg vil det være interessant å identifisere egenskaper som ikke krever spesifikasjoner fra brukeren. Et eksempel på egenskaper som ikke krever spesifikasjon er formalisering av lover som diskutert i avsnitt 5.4. Et annet eksempel vil være en form for «typesjekking» som (automatisk) sjekker om diagrammer støttes av AREA og om egenskaper motsier hverandre.

Forbedring av AREA implementasjonen. Implementasjonen av vår AREA add-in må sees på som en «*proof-of-concept*»-prototype. Det er derfor flere rent tekniske forhold av implementasjon som må forbedres eller ferdigstilles. For interaksjonsmodellen mot arkitekten, og andre brukere, gjelder dette både når, og hvordan, AREA skal kjøres og hvordan feil- og tilbakemeldinger vises. Håndtering av feilmeldinger fra SQL-spørringer krever spesielt mer arbeid, siden dette bør skjermes mest mulig fra brukeren. Rent teknisk må integrasjonen til Z3 ferdigstilles slik at dette brukes for alle eksemplene. Siden en liten feil her kan ha fatale konsekvenser bør denne delen testes/valideres/verifiseres grundigere enn vi har gjort i proto-typen. Vi bør også vurdere mer naturlige måter å merke objektene med sikkerhetsattributter.

7.2 Muliggjørende aktiviteter

Vårt eksperiment med automatisk resonnering i dette arbeidet har også åpnet opp for nye muligheter for videre arbeid som ikke kan sees på som en direkte fortsettelse på dette arbeidet. Vi konkluderer rapporten med en kort beskrivelse av noen slike muligheter. Denne delen har en spekulativ natur.

Verifisering. Målet med vår arbeid har vært å støtte arkitekter i sine (manuelle) resonneringer, spesielt ved å gi «alarm» når oppfattede mangler er identifisert. Dette kan derfor sees på som en form for «feilsøking». En mer vanlig motivasjon for resonneringsteknikker er å kunne gi en

[33] Dette er dokumentert i Bloebaum et al. (2018).

formell garanti for at egenskapen faktisk holder når det ikke oppstår en alarm[34]. Tilnærmingen vår kan også brukes til å gi slike garantier. Dette forutsetter full tillitt til formaliseringen, semantikken og implementasjonen. Det er også viktig å forstå at det som verifiseres er at egenskaper holder for modellen, og det sier ingenting om hvor korrekt selve modellen er. Den må i tillegg valideres. Slike behov er mindre kritiske for vårt arbeid hvor AREA brukes mer som et feilsøkningsverktøy. Komplekse løsninger består typisk av ulike komponenter som er satt sammen. Mange feil skyldes interaksjonen mellom de ulike komponentene; selve løsningen som kombinerer dem er ikke nødvendigvis sikker, selv om alle komponentene er sikre. Såkalt «*compositional (security) assurance*» (Rushby & DeLong, 2007) innebærer bruk av resonneringsteknikker for å oppnå en helhetlig sikkerhet fra en sammensetting av sikre komponenter. Dette kan være en mulig tilnærming for resonnering rundt sikkerhetsarkitekturer.

Generering av forklaring og abstraksjon. Generering av gode forklaringer fra modeller er en egenskap som er svært ønskelig, og et område det hadde vært veldig interessant å utvide våre teknikker med. Her har vi fokusert på å gi forklaring i form av å vise når resonneringen bryter sammen[35]. Ved å analysere dette kan man identifisere mangler. I tillegg kunne vi ha brukt resonnering til å forklare hvorfor noe holder, samt å forklare hvilke mellomliggende komponenter som kreves (ved å være en del av resonneringen) og hvilke som ikke kreves (ved ikke å være en del av resonneringen). Det er her viktig å vise en slik forklaring på et nivå som gir mening for arkitekten, som betyr å kunne oversette forklaringen fra logikken til notasjonen som brukes for modelleringen.

Ved bruk av andre resonneringsteknikker, som for eksempel «modellsjekking», kan man også gi forklaring, i form av et «moteksempel», på hvorfor en egenskap ikke holder. Det er verdt å merke at Jürjens & Shabalin (2004) bruker modellsjekking for å verifisere sikkerhetsaspekter i UMLSec (med en lignende systemarkitektur som oss).

En annen interessant tilnærming er å kunne spekulere og generere forslag til endring av modellen (eller egenskapen i seg selv) som gjør egenskapene sanne for de tilfeller der resonneringen bryter sammen. Dette kan oppnås med en form for resonnering kalt *abduksjon*. Ireland et al. (2011) har for eksempel utviklet en metode kalt *reasoned modelling critics* som kombiner modellerings- og bevismønstre for å spekulere endringer. Et eksempel på dette er fra avsnitt 5.1, hvor en alarm oppstod siden vi bare kunne finne brannmur for «system1» og «system2». Her kunne man spekulere at en endring i modellen som viste at «system1» og «system2» var de eneste (relevante) systemene, eller eventuelt dersom andre systemer fantes, foreslo prosjekter som må levere disse. Det siste forslaget kunne også vært tilfelle i avsnitt 5.3, hvor en tjeneste var merket med en `planlagt status` uten at et prosjekt var allokert til det. Her kunne det blir foreslått at «et prosjekt leverer tjeneste T» dersom T hadde `planlagt status`.

[34] Forskjellen mellom verifisering og feilsøking er når en alarm ikke oppstår: når man feilsøker betyr ikke mangel på alarm at modellen nødvendigvis er korrekt, men bare at man ikke fant noen feil. I en verifiseringssetting vil mangel på alarm bety at den er korrekt.

[35] Merk at siden problemstillingen er *undecidable* vil det være tilfeller hvor man ikke kan bevise noe som faktisk holder, og man kan ikke alltid skille mellom egenskaper som ikke er sanne og egenskaper man ikke klarer å bevise.

Generering av views & grafomskriving. I tillegg til å generere en forklaring vil det være ønskelig å kunne genere nye views (automatisk). Et opplagt eksempel på generering av views er gapanalysen i avsnitt 5.2.2, hvor arkitekten modellerer nåsituasjon og målbildet, og et view for gapet automatisk genereres. Et annet eksempel er å «løfte» resonneringen til modellnivået ved å implementere «*change impact analysis*» gjennom å generere mer abstrakte modeller, som beskrevet av Sunkle et al. (2013) og van Buuren et al. (2004). Generering av et «sikkerhets-view», som bare viser sikkerhetsrelatert informasjon, vil også vært et mål med et slikt arbeid. Merk dog at vi ikke har kjennskap til noe modelleringsspråk for sikkerhetsarkitektur som har definert et eget sikkerhetsview.

Når man generer et view som abstraherer bort detaljer vil man også kunne oppdatere dette viewet. Disse endringene burde da reflekteres i den underliggende modellen på en konsistent måte. Slik konsistens mellom ulike representasjoner kan oppnås gjennom en tilnærming kalt «*bidirectional transformations*» (Abou-Saleh, Cheney, Gibbons, McKinna, & Stevens, 2018).

Et relatert problem er at man ofte kan modellere diagrammer på ulik måte, men som er semantisk ekvivalent. Det er da ønskelig å kunne oversette mellom disse to representasjonene, eventuelt oversette til en slags «normalform». Vi har diskutert et eksempel på slik oversettelse hvor sikkerhetsattributter enten kan brukes til å merke eksisterende bokser, eller de kan være egne bokser. Slik oversettelse mellom ulike semantisk ekvivalent diagrammatiske representasjoner kan oppnås gjennom teknikker for *grafomskrivning* (Ehrig, Ehrig, & Prange, 2006).

Spørringer vs. attributter. I vårt rammeverk spesifiseres egenskapene som attributter i selve diagrammene og blir derfor en del av modellen. Et alternativ til å spesifisere dem er å ha et eget spørrespråk, med støtte for sikkerhetsrelaterte spørringer. I en ontologibasert tilnærming kan dette baseres på såkalte SPARKQL spørringer (som i Sunkle et al. (2013)), mens for andre teknikker må vi nok utvikle et slikt språk selv.

Tid/temporale egenskaper. Vi har gitt eksempler på at et prosjekt kanselleres og vi har vist hvordan man kan resonnere rundt konsekvenser av slike enderinger (se avsnitt 5.3). En annen problemstilling er at leveransene til et prosjekt endrer seg (for eksempel ved at en tjeneste ikke lenger vil leveres) eller at prosjektet/leveranser utsettes. Endring av en leveranse vil allerede være håndtert siden alarmer kommer som følge av mangel på nødvendige relasjoner: i eksemplene fra avsnitt 5.3 ville det ikke ha gjort noe forskjell om en relasjon fra prosjektet ble fjernet eller om hele prosjektet ble fjernet. Håndtering av en forsinkelse vil kreve modellering av tid, enten direkte eller indirekte gjennom en slags abstraksjon av tid (som for eksempel fremtidige tilstander). Vi kan da inkludere tidsperspektivet når vi resonnerer rundt avhengigheter, for eksempel for å passe på at en tjeneste ikke forutsettes brukt før den er levert. Dersom levering utsettes slik at en slik egenskap ikke holder kan man gi en alarm uten at nye sikkerhetsattributter er nødvendige.

Integrering av risiko- og trusselsmodeller. Mye sikkerhet baserer seg på risiko- og trusselbilder, og det vil være interessant å utforske om dette kan integreres/brukes i resonneringen. Vi har allerede nevnt arbeid med en ArchiMate-utvidelse med konsepter for

sårbarheter, trusler og risiko (van den Bosch, 2014), samt *UMLSec* (Jürjens, 2002), *abuse cases* (McDermott & Fox, 1999), *misuse cases* (Guttorm & Opdahl, 2005) og *accident scenarios* (Murali, Ireland, & Grov, 2015) for UML samt CORAS-tilnærmingen (Lund, Solhaug, & Stølen, 2010). Disse kan danne utgangspunktet for et slikt arbeid, som vil være spesielt viktig for å kunne resonnere rundt spørsmål S5 og S7 i introduksjonen.

Konsistens mellom operativt nivå og systemarkitektur. Vi har håndtert en form for sporing mellom det (abstrakte) operative nivået og den (mer konkrete) systemarkitekturen gjennom resonnering rundt sikkerhetsattributter som begge nivåene ble merket med. I såkalte *refinement*-baserte metoder beviser man at abstrakte representasjoner virkelig spesifiserer konkrete representasjoner, som betyr at alt som holder på abstrakt nivå også holder på det konkrete nivå (se for eksempel Back & Wright (2012)). Slike metoder fokuserer hovedsakelig på *oppførsel* som ikke kan brukes for alle aspekter av en arkitektur, men det hadde vært interessant å utforske om man kunne resonnere rundt sammenhengene mellom disse nivåene uten å måtte annotere med spesifikke (sikkerhets)attributter.

Integrasjon av kjøremiljø og angrepsdeteksjon. I arbeidet beskrevet her tar vi for oss «modeller» av de faktiske systemer og virksomheten. Ved å integrere det faktiske kjøremiljøet (for eksempel analyse av diverse logger) åpner det opp nye muligheter. Et eksempel er å kunne validere at den tekniske modellen overensstemmer med det faktiske systemet. Dette kan også gjøres i sanntid for å spore endringer i selve systemet til det mer operative nivået. For eksempel, dersom et system eller en tjeneste tas ned (for eksempel av en angriper eller for å håndtere et angrep) vil man da kunne spore dette mot operative konsekvenser av dette (det vil si hvilke evner man eventuelt mister og konsekvenser av å miste dem). SecGuru (Jayakaraman, Bjørner, Outhred, & Kaufmann, 2014) kan for eksempel overvåke nettverkstrafikk og bruker Z3 for å sjekke at policyer er overholdt.

Ved å modellere aktører og prosesser, kan man også resonnere om de rette prosessene er fulgt, også på et ikke-teknisk nivå (for eksempel har de riktige beredskapsplanene blitt iverksatt). Ved å inkludere trusselmodeller (for eksempel i form av angrepstrær (Schneider, 1999)) kan man bruke dette til deteksjon og reaksjon mot angrep. Ved å sammenligne situasjonen med modellerte angrepssteg kan man støtte deteksjon samt resonnere rundt mulige mottiltak. Man kan også se for seg hvordan dette kan brukes til å oppdatere trusselmodellen for angrep som delvis følger et kjent mønster, men med noen unntak (eventuelt kombiner ulike mønstre). Trusselmodellen kan da oppdateres med slike nye variasjoner og brukes for å håndtere fremtidige angrep, samt å resonnere om man har evner til å håndtere dem.

Referanser

Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., & Stevens, P. (2018). Introduction to Bidirectional Transformations. *Bidirectional Transformations: International Summer School, Oxford, UK, July 25-29, 2016, Tutorial Lectures* (ss. 1-28). Springer.

Ajer, A. K., & Olsen, D. H. (2018). Enterprise Architecture Challenges: a Case Study of Three Norwegian Public Sectors. *Twenty-Sixth European Conference on Information Systems (ECIS 2018).*

Back, R.-J., & Wright, J. (2012). *Refinement calculus: a systematic introduction.* Springer Science & Business Media.

Bjørner, N., & Jayaraman, K. (2015). Checking Cloud Contracts in Microsoft Azure. *International Conference on Distributed Computing and Internet Technology* (ss. 21-32). Springer.

Bloebaum, T. H., Lund, K., Grov, G., & Kristiansen, P. (2018). *Innspill til innretting av investeringsporteføljen for programområdet INI.* FFI-rapport 18/02076, BEGRENSET.

Broy, M., & Cengarle, M. V. (2011). UML formal semantics: lessons learned. *Software and Systems Modeling, 10*(4), 441-446.

Brucker, A. D., Doser, J., & Wolff, B. (2006). A model transformation semantics and analysis methodology for SecureUML. *International Conference on Model Driven Engineering Languages and Systems* (ss. 306-320). Springer.

Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model checking.* MIT Press.

Clocksin, W. F., & Mellish, C. S. (1994). *Programming in Prolog* (Fourth Edition. utg.). Berlin: Springer.

Cook, B. (2018). Formal reasoning about the security of AWS (Invited paper). *CAV'2018*.

Dahl, H., Hogganvik, I., & Stølen, K. (2007). *Structured Semantics for the CORAS Security Risk Modelling Language.* SINTEF ICT Report no. A970.

Dalton, G. C., Colombi, J., & Mills, R. (2006). Modeling Security Architecture for the Enterprise. *Proceedings of the 11th International Command and Control Research and Technology Symposiusm (ICCRTS)-- Coalition Command and Control in the Networked Era* .

De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. *TACAS 2008*, ss. 337-340.

Ehrig, H., Ehrig, K., & Prange, U. (2006). *Fundamentals of Algebraic Graph Transformations.* Monographs in Theoretical Computer Science, Springer.

Fisher, K., Launchbury, J., & Richards, R. (2017). The HACMS program: using formal methods to eliminate exploitable bugs. *Phil. Trans. R. Soc. A, 375*(2104), 20150401.

Forsvarsstaben. (2018). *Digitaliseringsstrategi for Forsvaret* .

Guttorm, S., & Opdahl, A. L. (2005). Eliciting security requirements with misuse cases. *Requirement Engineering, 10*(1), 34-44.

Hallingstad, G., Gay, S., S. Gay, J. -F., & Virvilis-Kollitiris, N. (2014). *CIS SECURITY CAPABILITY BREAKDOWN – COMPREHENSIVE APPROACH VERSION 2.0.* NATO Technical Report 2014/NCB009779/05.

Hansbø, M., Jørgensen, H., & Markussen, R. (2013). *Arkitekturarbeid i Forsvaret med forenklet bruk av NATO Architecture Framework (NAF).* FFI-rapport 2013/01069.

Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning.* Cambridge University Press.

Ireland, A., Grov, G., Llano, M. T., & Butler, M. (2011). Reasoned Modelling Critics: Turning Failed Proofs into Modelling Guidance. *Science of Computer Programming, 78*(3), 293–309.

Jayakaraman, K., Bjørner, N., Outhred, G., & Kaufmann, C. (2014). *Automated Analysis and Debugging of Network Connectivity Policies.* Microsoft Research Technical Report MSR-TR-2014-102.

Jill, L. H., & Simon, H. A. (1987). Why a diagram is (sometimes) worth ten thousands words. *Cognitive Science, 11*(1), 65-100.

Jürjens, J. (2002). UMLsec: Extending UML for secure systems development. *International Conference on The Unified Modeling Language* (ss. 412-425). Springer.

Jürjens, J., & Shabalin, P. (2004). Automated verification of UMLsec models for security requirements. *International Conference on the Unified Modeling Language* (ss. 365-379). Springer.

Lin, Y., Grov, G., & Arthan, R. (2016). Understanding and maintaining tactics graphically OR how we are learning that a diagram can be worth more than 10K LoC. *Journal of Formalized Reasoning , 9*(2), 69-130.

Lund, M. S., Solhaug, B., & Stølen, K. (2010). *Model-driven risk analysis: the CORAS approach.* Springer.

Mancini, F., Farsund, B., & Lillevold, F. (2017). *Sikkerhetsarkitektur for Forsvaret - en innledende studie av rammeverk og begreper*. FFI-rapport 17/01169.

McDermott, J., & Fox, C. (1999). Using abuse case models for security requirements analysis. *Computer Security Application Conference (ACSAC'99)*, ss. 55-64.

Murali, R., Ireland, A., & Grov, G. (2015). A Rigorous Approach to Combining Use Case Modelling and Accident Scenarios. *NASA Formal Methods 2015*, ss. 253-278.

Nasjonal sikkerhetsmyndighet. (2006). Veiledning i grunnleggende sikkerhetsarkitektur og -funksjonalitet for PARTISJONERT operasjonsmåte.

NATO Consultation, Command and Control Board. (u.d.). *NATO Architecture Framework Version - Enabling NNEC for NATO.* NATO/EAPC UNCLASSIFIED.

Rushby, J., & DeLong, R. (2007). Compositional security evaluation: the MILS approach. *International Common Criteria Conference.*

Schneider, B. (1999, December). Attack trees: Modelling security threats. *Dr. Dobb's Journal of Software Tools, 24*(12), 21-29.

Sherwood, J., Clark, A., & Lynas, D. (2005). *Enterpise Security Architecture - A Business-Driven Approach.* CRC Press.

Shostack, A. (2014). *Threat Modelling: Designing for Security* . Wiley.

Sunkle, S., Kulkarni, V., & Roychoudhury, S. (2013). Analyzing Enterprise Models Using Enterprise Architecture-based Ontology. *International Conference on Model Driven Engineering Languages and Systems* (ss. 622-638). Springer.

Thorbruegge, M., Hallingstad, G., & Gay, S. (2017). *CIS Security Shortfall Assessment 2017.* NATO Technical Report 2017/NCB011774/01, NATO RESTRICTED.

Turing, A. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London mathematical society, 2*(1), 230-265.

van Buuren, R., Jonkers, H., Iacob, M.-E., & Strating, P. (2004). Composition of relations in enterprise architecture models. *International Conference on Graph Transformation* (ss. 39-53). Springer.

van den Bosch, S. (2014). Designing Secure Enterprise Architectures - A comprehensive approach: framework, method, and modelling language. Master thesis, University of Twente.

Warmer, J., & Kleppe, A. (1998). *The Object Constraint Language - Precise Modelling with UML.* Addison-Wesley.

Wenzel, M., & Chaieb, A. (2007). SML with antiquotations embedded into Isabelle/Isar. *Workshop on Programming Languages for Mechanized Mathematics (satelite of CALCULEMUS 2007).*

A Forslag til formell beskrivelse av semantikken

Her gis en formell beskrivelse av semantikken som ble diskutert på overordnet nivå i kapittel 3, og illustrert med noen av eksemplene i kapittel 5. Merk at dette bare er et forslag til semantikk for videre diskusjon, og man må påberegne endringer av det som er beskrevet her. En del forenklinger er gjort i presentasjonen, men det vil fortsatt være en fordel med noe basiskunnskap om logikk.

Selv om eksemplene som er vist er enkle så er de generelle slutningene av interesse *undecidable*. Dette betyr at det ikke finnes en algoritme som garanterer at den finner et svar på om setningen er sann eller usann (Turing, 1937). Ulike automatiserte resonneringsteknikker bruker derfor en kombinasjon av søk, heuristikk og abstraksjoner for å oppnå de generelle slutninger. Disse garanterer derimot ikke at de finner et svar: dersom en teknikk ikke klarer å bevise at en setning er sann, betyr ikke dette nødvendigvis at setningen er usann. Det kan også skyldes at teknikken ikke var god nok. Et positivt svar vil derimot bety at setningen holder (noe som kalles *soundness*). I noen tilfeller gis det også et *moteksempel* til setningen som skal bevises. Det vil i så fall indikere at setningen er usann.

Her fokuserer vi på en resonneringsteknikk kalt *deduksjon*, som betyr at vi bruker logisk resonnement for å nå slutningen. En alternativ teknikk, som også er nevnt i hovedteksten, er *modelsjekking* som bruker algoritmiske teknikker til å søke gjennom alle mulige tilstander. Denne teknikken vil gi moteksempler når setningen ikke holder.

I avsnitt A.1 til A.3 ser vi på hvordan en modell kan formaliseres logisk, mens i A.4 og A.5 gir vi detaljer på hvordan selve resonneringen fungerer. I A.6 har vi en generell diskusjon om logisk tolkning. Merk at resoneringen vil være automatisert i et eksternt verktøy som for eksempel Z3[36].

A.1 Interne attributter

Semantikken representeres ved å kode diagrammene i en logikk, som det da kan resonneres rundt. For å støtte en slik koding introduserer vi noen attributter som brukes internt i resonneringen og som genereres automatisk av AREA. De er:

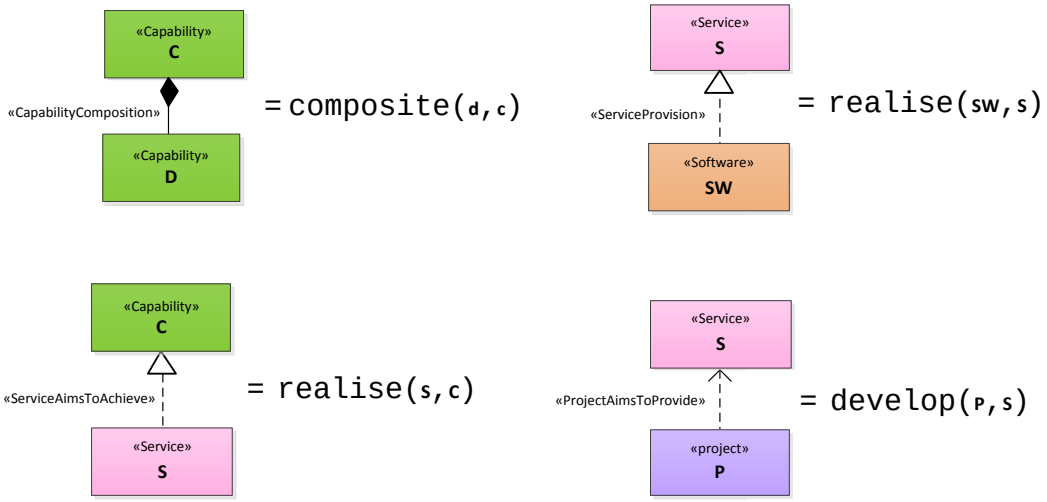
- **`service(E)`** indikerer at entitet **`E`** er en tjeneste.
- **`software(E)`** indikerer at entitet **`E`** er en programvare.
- **`capability(E)`** indikerer at entitet **`E`** er en kapabilitet.
- **`project(E)`** indikerer at entitet **`E`** er et prosjekt.

[36] Z3 bruker en ulik tilnærming for resonneringen enn vi bruker her, men vi vil ikke gå inn i detaljer på hvordan Z3 fungerer. For slike detaljer referer vi til (De Moura & Bjørner, 2008).

A.2 Formel modell av diagrammer

For å kode diagrammene logisk, definerer vi et sett med oversettelsesregler som mapper modellen til AREAs interne logiske representasjon.

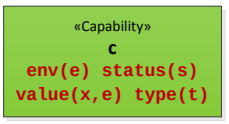
Først defineres reglene for relasjonene. AREA støtter fire typer relasjoner, med følgende oversettelsesregler:



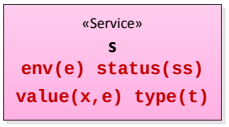
Dekomponering av kapabiliteter til deres delkapabiliteter gjøres gjennom `composite`-relasjonen.[37] Relasjoner som «konkretiserer» gjennom å realisere entiteter fra et mer abstrakt nivå kodes gjennom en realiseringsrelasjon (`realise`). Dette gjelder både tjenester som realiser kapabiliteter, og programvare som realiser tjenester. Til slutt brukes `develop`-relasjonen på prosjekter som anskaffer eller utvikler en tjeneste.

AREA må også kode egenskapene til attributtene for alle entitetene. Fire forskjellige bokser støttes. Disse har følgende oversettelsesregler:

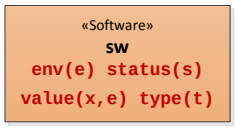
[37] AREA støtter bare dekomponering av kapabiliteter siden delen av NAF som støttes her er basert på de eksemplene vi har gitt. Tjenester, programvare- og systemkomponenter vil kunne dekomponeres med samme relasjon, men det er ikke støttet.



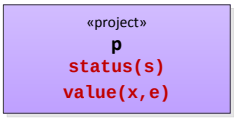
$$= (env(c,e) \wedge status(c,s) \rightarrow capability(c) \wedge value(x,eval(e)) \wedge type(c,s))$$



$$= (env(s,e) \wedge status(s,ss) \rightarrow service(s) \wedge value(x,eval(e)) \wedge type(s,t))$$



$$= (env(sw,e) \wedge status(sw,s) \rightarrow software(sw) \wedge value(x,eval(e)) \wedge type(sw,s))$$

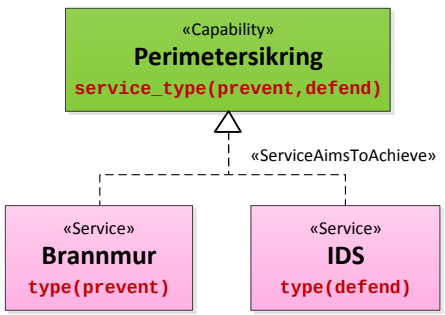


$$= status(p,s) \rightarrow project(p) \wedge value(x,val(e))$$

Merk at vi har forenklet reglene ved bare å gi ett attributt av hver type. Det er enkelt å utvide dette til å støtte flere attributter av hver type, samt å støtte utelatelse av attributter gjennom ikke å inkludere de attributtene i oversettelsen.

Hver entitet kan ha attributter som begrenser omfanget av entiteten: `env` begrenser entiteten til et gitt miljø eller en gitt kontekst, mens `status` kan gi ytterligere krav til inngående relasjoner. For eksempel vil `status(planlagt)` kreve at et prosjekt er allokert til å levere entiteten (dvs. det må være en inngående `develop`-relasjon fra et prosjekt). Disse må holde for at de andre attributtene i entiteten kan brukes. Logisk representeres dette som en implikasjon. Merk også at det ikke gir noe mening å ha `env` og `type` for et prosjekt (i alle fall med vår tolkning av dem), og disse er derfor ignorert.

For å illustrere oversettelsen brukes modellen fra Figur 3.1, som vi gjengir her:



Denne har to realiseringsrelasjoner mellom to ulike tjenester og én kapabilitet. Dette gir følgene representasjon[38]:

```
realise(brannmur,perimetersikring)
realise(iDS,perimetersikring)

capabilitity(perimentersikring)
service(brannmur)
type(brannmur,prevent)
service(iDS)
type(iDS,defend)
```

Disse har ingen begrensninger og implikasjonene har derfor blitt fjernet. Videre splitter vi opp konjunksjonene (siden $A \wedge B$ er det samme som A og B).

Vi har nå kodet modellen som da kan brukes til å resonere rundt egenskapsattributtene.

A.3 Støtteattributter og visualiseringsattributter

Støtte- og visualiseringsattributter har en standard semantikk i forhold til vanlige programmeringsspråk så vi vil ikke gå dypt inn i detaljene.

Visualiseringsattributtene (`scb_color(N)` og `gap_color(N)`) vil evaluere argumentene og fargelegge de tilknyttede entitetene i forhold til resultatet av denne evalueringen. Fargeleggingen er beskrevet i avsnitt 3.2.4.

For `type(T)`, `env(P)`, `status(S)` gjøres det ingen evaluering[39]. For `eal(N)` må `N` evalueres, og denne verdien må være mellom 1 og 7. `avg(Vs)`, `sum(Vs)` og `sub(A,B)` evaluerer først argumentene og returnerer deretter resultatet av den aktuelle utregningen: `avg(Vs)` regner ut gjennomsnittet av argumentene, `sum(Vs)` regner ut summen av argumentene, mens `sub(A,B)` trekker `B` fra `A`. Disse vil typisk være som argument for `value`, `scp_color` eller `gap_color`. Udefinerte konstanter «ignoreres» ved å sette dem til 0.[40]

`value(E,V)` evaluerer verdien til `V` og lagrer den i posisjon `E` av en global tabell, som betyr at det ikke har tilknytning til entiteten hvor den er deklarert. `E` må være unik, som betyr at

[38] I logikken har variabler stor forbokstav og konstanter liten forbokstav. Dette er samme syntaks som sikkerhetsattributtspråket fra kapittel 3. Siden selve navnet på boksene skal tolkes som en konstant gis de liten forbokstav i oversettelsen.

[39] Man burde fortsatt sjekke at argumentene har lovlig verdi, for eksempel for `status(S)` så må `S` være `planlagt` eller `støttet`.

[40] Vi har valgt en forenklet løsning for udefinerte konstanter som nok ikke er tilstrekkelig og må forbedres i fremtiden. For eksempel så skilles det ikke mellom konstanter som er «gjemt» grunnet begrenset omfang (gjennom `env-` eller `status`attributter) og ikke-eksisterende konstanter. Løsningen fungerer for eksemplene i denne rapporten, men ville for eksempel ikke ha fungert i sammenheng med `gap_color`, hvor målet er en så lav verdi som mulig. Det er verdt å nevne at slike udefinerte verdier generelt er problematiske å håndtere i formell logikk og gjøres på ulike måter i ulike logikker.

`value(E,V)` kan kun eksistere én plass for hver `E`. Dette er trolig noe som må revurderes for større modeller. Det kan heller ikke være sirkulær avhengighet mellom forskjellige `E`. For eksempel så kan man ikke ha både `value(x,y)` og `value(y,x)`.

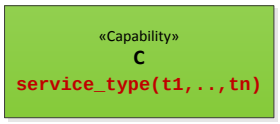
A.4 Verifisering av egenskapsattributter

Egenskapsattributtene må verifiseres gjennom et formelt bevis. Dette gjøres ved å generere en såkalt *bevisforpliktelse*[41] (*proof obligation*) som er en matematisk setning som AREA må bevise. Formen på en slik bevisforpliktelse er:

$A \vdash P$

Dette betyr fra (antagelsene) `A` må vi vise at `P` holder. Ofte har vi ikke noen antagelser (utenom representasjon av modellen som er implisitt) og da skriver bare $\vdash P$.

Vi støtter to egenskapsattributter: `service_type` og `software_type`. Hvert argument av `service_type` vil vi få en egen bevisforpliktelse[42]:



$$= \quad \vdash \texttt{service\_type(c,t1)} \quad \ldots \quad \vdash \texttt{service\_type(c,tn)}$$

`service_type` har følgende definisjon[43]:

(service_type_def): $\texttt{service\_type(C,T)} \equiv \exists\ \texttt{x.}\ \texttt{service(x)} \wedge \texttt{realise(x,C)} \wedge \texttt{type(x,T)}$

Dette betyr det må finnes en tjeneste (`x`) som realiserer `C` og er av type `T`.

For å kunne resonere brukes såkalte *inferensregler*. En slik regel har følgende format:

$$\frac{\vdash P_1 \quad \ldots \quad \vdash P_n}{\vdash P}$$

For å bevise forpliktelsen under streken (`P`), må vi bevise forpliktelsene over streken. Dette kan gjøres «fremover»: fra P_1 og … og P_n kan vi konkludere at `P` er sant, eller «bakover»: for å bevise `P` må vi bevise P_1 og … og P_n. Vi kommer tilbake til den praktiske betydningen av retningen nedenfor.

[41] I noen tilfeller bruker vi bare *forpliktelse* for *bevisforpliktelse*.
[42] Vi illustrerer her bare egenskapen til en kapabilitet, men man kan også tenke seg tjenester som inneholder dette attributtet. Dette vil gi de samme bevisforpliktelsene.
[43] Husk at stor forbokstav brukes for variabler, som kan instansers til andre verdier av resonneringen.

En viktig egenskap til `realise` er at relasjonen er *transitiv*:

(realise_trans):

$$\frac{\vdash \texttt{realise(X,Y)} \quad \vdash \texttt{realise(Y,Z)}}{\vdash \texttt{realise(Y,Z)}}$$

Dette betyr at for å vise at et element `Y` realiser `Z` så finner vi et mellomliggende element `Y` som realiser `X` og blir realisert av `Z`. En konsekvens av denne regelen er for eksempel at en tjeneste som realiser en kapabilitet `C` ikke trenger å være direkte tilkoblet til `C`. Her er eksempler på andre regler som kan brukes:

(status_planlagt):

$$\frac{\vdash \texttt{develop(Y,X)}}{\vdash \texttt{status(X,planlagt)}}$$

(status_støttet):

$$\frac{}{\vdash \texttt{status(X,støttet)}}$$

(case):

$$\frac{\vdash P \rightarrow Q \quad \vdash R \rightarrow Q \quad \vdash P \vee R}{\vdash Q}$$

(existI):

$$\frac{\vdash \texttt{P(X)}}{\vdash \exists\ \texttt{x. P(x)}}$$

Merk at denne listen av regler ikke er komplett, men er kun for illustrasjon. Reglene (case) og (existsI) er standardregler i førsteordens logikk, mens de andre er spesifikke for vår tolkning. Av praktisk betydning betyr dette at (realise_trans), (status_planlagt) og (status_støttes) må gis i Z3, mens (case) og (existsI) er innebygd. (status_planlagt) og (status_støttes) brukes hovedsakelig for å kunne vise at beviset er innenfor en gitt begrensing. (status_planlagt) er for det tilfellet der en entitet bare er planlagt og vi må vise at det er prosjekt som skal utvikle den, mens (status_støttes) er for det tilfellet der entiteten allerede er støttet og vi ikke trenger å resonnere videre for å «fjerne begrensningen».

Når resonneringen foregår «fremover» starter vi med eksisterende faktum og resonnerer frem til nye faktum. For eksempelet overfor vet vi at for denne modellen holder `realise(myIDS,iDS)`og `realise(iDS,perimetersikring)`. Vi kan derfor bruke *(realise_trans)* for å trekke konklusjonen at `realise(myIDS,perimetersikring)`. Dette kan da brukes videre i beviset.

Resonneringen kan også foregå «bakover» hvor en bevisforpliktelse reduseres til nye forpliktelser. For eksempel dersom vi må bevise $\vdash$ `status(myIDS,planlagt)` så kan vi jobbe bakover og redusere forpliktelsen til å måtte bevise $\vdash$ `develop(Y,myIDS)` ved å bruke *(status_planlagt)*. Vi må da finne en komponent som utvikler (`develop`) `myIDS`.

For å illustrere vil eksempelet fra Figur 1.1 resultere i følgende bevisforpliktelser:

```
├ service_type(perimetersikring,prevent)
├ service_type(perimetersikring,defend)
```

Disse kan gjennom (*service_type_def*) og (*existI*) utrulles til:

```
├ service(X) ∧ realise(X,perimetersikring) ∧ type(X,prevent)
├ service(Y) ∧ realise(Y,perimetersikring) ∧ type(Y,defend)
```

For den første forpliktelsen trenger vi bare å instansiere variabelen `X` med konstanten `brannmur`:

```
├ service(brannmur) ∧ realise(brannmur,perimetersikring) ∧
  type(brannmur,prevent)
```

Vi kan så bryte konjunktene ned i separate bevisforpliktelser:

```
├ service(brannmur)
├ realise(brannmur,perimetersikring)
├ type(brannmur,prevent)
```

Disse følger direkte fra kodingen av modellen ovenfor. Beviset av den andre bevisforpliktelsen er nesten identisk. Merk at Z3 vil gjøre en lignende resonnering automatisk.

Bevisforpliktelsene og definisjonen til `system_type` er veldig lik `service_type`:


```
=   ├ system_type(c,t1)
    ...
    ├ system_type(c,tn)
```

(system_type_def): `system_type(C,T) ≡ ∃ x. system(x) ∧ realise(x,C) ∧ type(x,T)`

For å illustrere både `system_type` og hvordan alarmer fremkommer bruker vi istedenfor eksempelet fra Figur 5.1. Modellen oversettes til logikk som følger:

```
realise(brannmur,perimetersikring)
realise(iDS,perimetersikring)
realise(brannmurA,brannmur)
realise(brannmurB,brannmur)
realise(myIDS,iDS)
capabilitity(perimentersikring)
service(brannmur)
service(iDS)
software(myIDS)
type(myIDS,defend)
```

```
env(brannmurA,system1) →
      software(brannmurA) ∧ type(brannmurA,prevent)
env(brannmurB,system2) →
      software(brannmurB) ∧ type(brannmurB,prevent)
```

Følgende bevisforpliktelser vil generes fra *Perimetersikring* entiteten:

```
├ system_type(perimetersikring,prevent)
├ system_type(perimetersikring,defend)
```

Beviset av den andre forpliktelsen er nesten identisk til eksemplene ovenfor. Hovedforskjellen er at vi må bruke *(realise_trans)* for å utlede `realise(myIDS,perimetersikring)` fra `realise(myIDS,iDS)` og `realise(iDS,perimetersikring)`.

For den første bevisforpliktelsen er det to ulike tilfeller som håndteres separert. Dette gjøres ved hjelp av *(case).* Etter litt forenkling har vi tre nye bevisforpliktelser:

```
P ├ system_type(Perimetersikring,prevent)
R ├ system_type(Perimetersikring,prevent)
├ P ∨ R
```

Vi instansierer `P` til `env(brannmurA,system1)` og `R` til `env(brannmurB,system2).` Dette resulterer i:

```
env(brannmurA,system1) ├ system_type(perimetersikring,prevent)
env(brannmurB,system2) ├ system_type(perimetersikring,prevent)
├ env(brannmurA,system1) ∨ env(brannmurB,system2)
```

Bevisene av de to første forpliktelsene følger samme struktur som vi har sett ovenfor. Forskjellen er at vi må bruke antagelsen for å få de rette faktaene. For eksempel så brukes `env(brannmurA,system1)` på

```
env(brannmurA,system1) →
      software(brannmurA) ∧ type(brannmurA,prevent)
```

Dette gjør at vi nå kan bruke både `software(brannmurA)` og `type(brannmurA, prevent)`.

Den siste bevisforpliktelsen er

```
├ env(brannmurA,system1) ∨ env(brannmurB,system2)
```

Denne kan vi ikke bevise og er grunnen til at vi får alarmen. En bruker må da bestemme om dette er noe som kan godtas. Forklaring på hvorfor resonneringen bryter sammen vil være en

viktig del i å forklare hva problemet er for brukeren, som vi har identifisert som mulig videre arbeid i hoveddokumentet.

A.5 Logisk tolkning av regler

I avsnitt 5.4 ble det illustrert hvordan regler kan formaliseres i logikk og automatisk sjekkes av AREA. Her detaljeres ytterligere hvordan en slik tilnærming kan foregå.

Regelen som skal formaliseres sier at *«all programvare som håndterer BEGRENSET informasjon skal ha et Evaluation Assurance Level (EAL) på minimum nivå 4»*. For at denne regelen skal være aktuell må det være en relasjon til en BEGRENSET entitet som brukes for å representere et BEGRENSET sikkerhetsdomene. For å oppnå dette så må AREA matche og instansiere en variabel `X` for:

```
realise(X,bEGRENSET)
```

Dersom AREA finner en slik match genereres følgende bevisforpliktelser for `X`:

```
├ software(X) → (eal(X,N) ∧ N ≥ 4)
├ ∀z. realise(z,X) ∧ software(z) → (eal(Z,N) ∧ N ≥ 4)
```

Oversatt til norsk sier den første forpliktelsen at dersom `X` en programvarekomponent, må den ha EAL-nivå 4 eller høyere. Den andre forpliktelsen håndterer indirekte relasjoner til komponenten `X` som er BEGRENSET. For eksempel, en BEGRENSET tjeneste som blir realisert av en programvareentitet. Merk at ∀ betyr at alle slike programvarekomponenter som (direkte eller indirekte) realiser at `X` må ha EAL-nivå 4 eller høyere.

Det spesifikke eksempelet brukt i avsnitt 5.4 er gitt i Figur 5.11. Relasjonen mellom entitetene blir i logikken representert av

```
realise(s,bEGRENSET)
```

Her matches og instansieres `X` derfor til `s`. I dette tilfellet holder `software(s)`. Vi må derfor bevise følgende forpliktelser:

```
├ eal(s,N)
├ N ≥ 4
```

Det kan ikke bevises siden vi ikke kan vise at `s` har det riktige EAL nivået. I det oppdaterte eksempelet fra Figur 5.12 vil sikkerhetsattributtet `eal(4)` representeres i logikken som

```
eal(s,4)
```

Det kan brukes direkte til å vise:

```
├ eal(s,4)
├ 4 ≥ 4
```

A.6 Diskusjon om logisk tolkning

Vi har gitt én logisk tolkning av semantikken for en liten del av NAF (versjon 3). Dette kan danne basis for diskusjon om hva hver entitet og relasjon egentlig betyr, som videre vil resultere i en enhetlig semantikk som arkitektene kan være enige i.

I tillegg til en omforent semantikk må man velge en passende logikk man kan representere semantikken i. Her har vi brukt en *klassisk* tolkning, hvor rollen til en logisk setning er å utrykke *sannhet*. En konsekvens av dette er for eksempel at

$$\texttt{service\_type(defend,defend)} \equiv \texttt{service\_type(defend)}$$

Med andre ord kan vi ikke direkte utrykke en form for «redundans» med to ulike `defend`-mekanismer, siden det samme argumentet kan brukes flere ganger i resonneringen. Dette betyr at det holder å etablere `service_type(defend)`en gang. Et alternativ er en *lineær* tolkning hvor rollen til setningen uttrykker *ressursbruk*. I dette tilfellet vil `service_type(defend,defend)` kreve at det er to `defend`-mekanismer (ressurser). Siden ressurser som brukes for å etablere en `defend`-mekanisme vil «brukes opp», og kan da ikke brukes for å etablere den andre mekanismen. Vi må vurdere om en lineær tolkning er mer passende for disse attributtene.

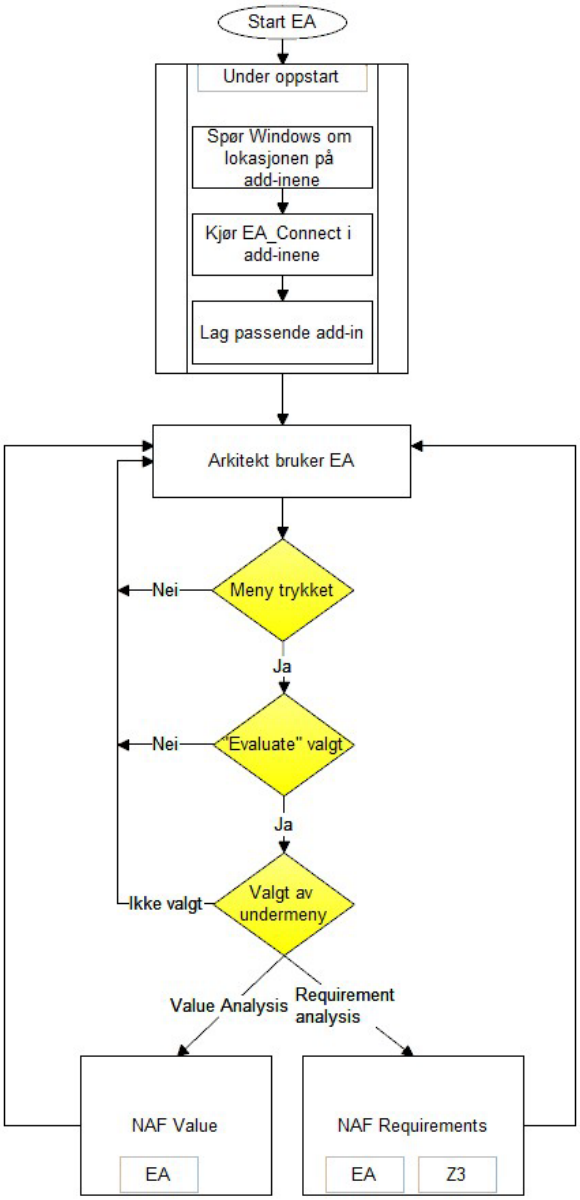
Logikken (som vi bruker) er binær: enten er en logisk setning sann eller usann. Det er også tilfeller hvor vi ikke kan bevise den (og vi kan derfor ikke si at den er sann) eller motvise den (og vi kan derfor ikke vise at den er usann); med andre ord vet vi ikke om den er sann eller usann. Et viktig spørsmål er hvordan vi skal tolke relasjoner eller komponenter som ikke er modellert. Prolog bruker en «*closed world assumption*», som i prinsippet betyr at alt som ikke er modellert ikke er sant. Hvis vi skal bevise at det ikke er en relasjon mellom A og B, og den ikke er modellert kan vi konkludere at en slik relasjon ikke eksisterer. Dersom vi derimot ikke følger prinsippet om en «*closed world assumption*», må man spesifikt modellere/kode at en slik relasjon ikke eksisterer. Vi har ikke brukt en «*closed world assumption*» her.

Valg av resonneringsteknikk vil også ha innvirkning på egenskaper man kan uttrykke og hva man kan resonnere om. For eksempel, bruker ontologibaserte rammeverk hovedsakelig en logikk kalt *description logic*. Videre brukes de fleste logikker til å uttrykke såkalte *safety*-egenskaper, som brukes for å uttrykke at «noe som ikke skal skje heller ikke skjer». Modellsjekking støtter ofte såkalt *temporal logic*, hvor man kan uttrykke kausale egenskaper gjennom en abstraksjon av tid (f.eks. A er sant frem til B er sant). Her støttes ofte også såkalte *liveness*-egenskaper, som kan utrykke at «noe som skal skje faktisk skjer» (f.eks. en gang i

fremtiden blir A sann). Noen tilgjengelighetsaspekter er for eksempel naturlig å uttrykke som *liveness*-egenskaper.

B Operasjonell beskrivelse av AREA

Avsnitt 4.2 gav en beskrivelse av implementasjonen av vårt resonnerings-verktøy AREA for Enterprise Architect. Til høyre vises et flytdiagram som gir mer detaljer om hvordan verktøyet fungerer.



Den første delen/boksen skjer automatisk når EA er konfigurert riktig. Under oppstart vil Enterprise Architect starte alle add-in som er registrert. Disse vil kjøre i bakgrunnen helt til brukeren gjør noe som trigger en aktuell add-in til å utføre en handling. I prototypen tilsvarer dette at det er gjort et menyvalg spesifisert i programvaren. Som diskutert i hovedteksten vil dette trolig endres i fremtiden slik at en bruker ikke trenger å velge dette fra en meny.

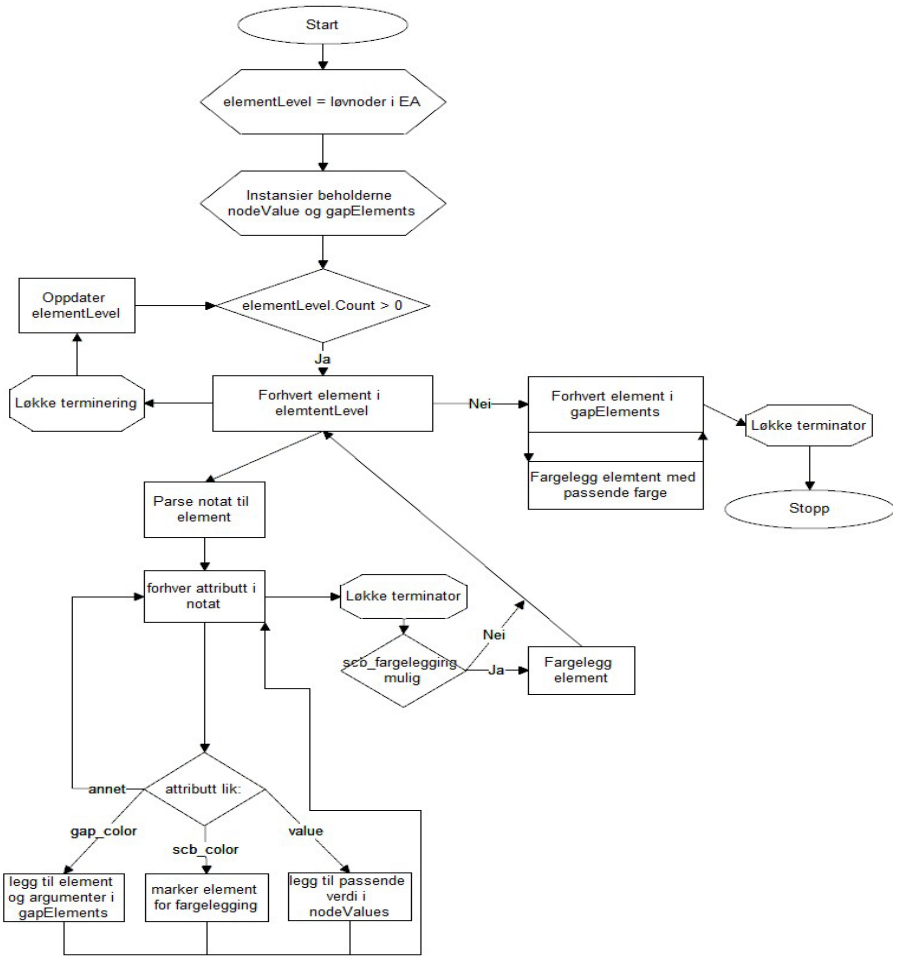
Den neste delen viser hva som skjer når en bruker starter AREA som vist i Figur 4.2. Detaljene for *«NAF value»* og *«NAF Requirements»* diskuteres i de neste avsnittene.

Vi har ikke prioritert effektiviteten til prototypen. Det betyr blant annet at all analyse repeteres hver gang en bruker ber om ny analyse. Her er det store forbedringspotensialer ved å analysere kilden til endringene og kun analysere de delene av modellen hvor endringen har potensielle konsekvenser. Det vil trolig bli viktigere for større modeller.

B.1 NAF Value

En illustrasjon av «NAF Value» under kjøring er gitt i grafen under. *NATO CIS Security Capability Breakdown* er organisert i nivåer hvor et nivå fargelegges (og analyseres) basert på nivået under. Disse traverseres «bottom up», og for hvert nivå gjennomgås nodene etter attributter som er nødvendige for å utføre en verdianalyse. For de elementene med en verdi

lagres verdien i en egen datastruktur. Elementer merket med `gap_color` lagres i en egen datastruktur for senere prosessering, mens elementer merket med `scp_color` fargelegges når alle verdiene som fargeleggingen baserer seg på er evaluert. Dette gjøres helt til toppen av hierarkiet er gjennomgått. Til slutt fargelegges alle elementene merket med `gap_color` i sine respektive farger. Merk at enkelte elementer ikke kan fargelegges med en gang grunnet avhengigheter til andre elementer som ikke har blitt evaluert.



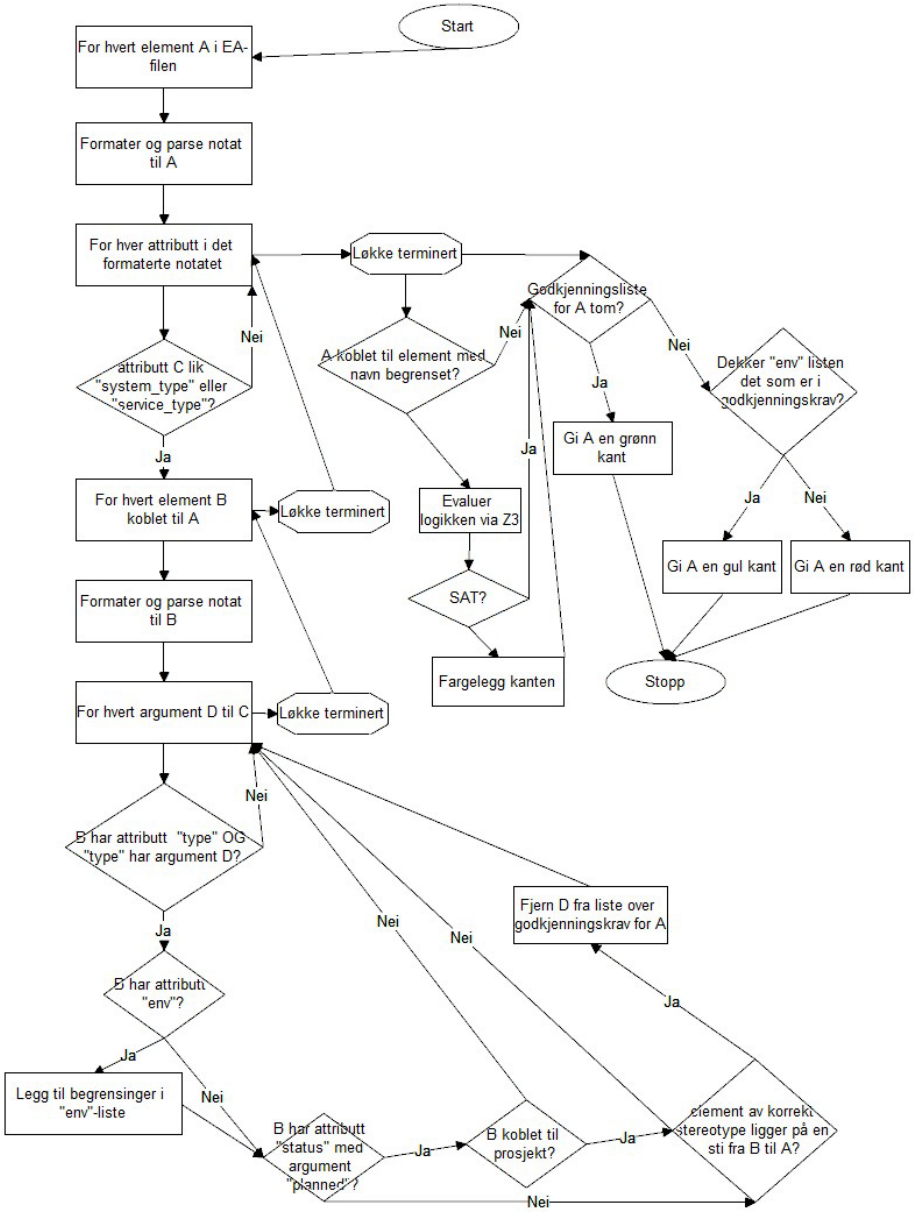
B.2 NAF Requirements

«*NAF Requirements*» er den største delen av AREA. Det er også her Z3 blir brukt. Resonneringen foregår derfor på to forskjellige måter. På sikt vil Z3 brukes for å støtte all resonnering, men dette er ikke ferdig for denne versjonen (0.15). Under vises et flytdiagram for dette valget.

AREA traverserer alle elementene i modellen og søker etter attributter av `system_type` eller `service_type`. Finner programmet et element A med et slikt attributt, vil den undersøke

argumentene og følge relasjonene etter elementet som tilfredsstiller attributtet i henhold til logikken tidligere beskrevet.[44]

Etter at alle elementene er evaluert vil programmet sjekke om en entitet er koblet til en tjeneste kalt *BEGRENSET*, som betyr at AREA må sjekke EAL nivå (se avsnitt 5.4). I dette tilfellet vil Z3 brukes for resonneringen. Avhengig av resultatet fra Z3 (og traverseringen av datastrukturen) vil elementene fargelegges i sine respektive farger. Det vil også gis en varselmelding til bruker om det er nødvendig.



[44] Resonneringen har med andre ord blitt hardkodet for de ulike attributtene, og deduksjonen fra vedlegg A er derfor ikke direkte implementert.

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's MISSION

FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's VISION

FFI turns knowledge and ideas into an efficient defence.

FFI's CHARACTERISTICS

Creative, daring, broad-minded and responsible.

Om FFI

Forsvarets forskningsinstitutt ble etablert 11. april 1946. Instituttet er organisert som et forvaltningsorgan med særskilte fullmakter underlagt Forsvarsdepartementet.

FFIs FORMÅL

Forsvarets forskningsinstitutt er Forsvarets sentrale forskningsinstitusjon og har som formål å drive forskning og utvikling for Forsvarets behov. Videre er FFI rådgiver overfor Forsvarets strategiske ledelse. Spesielt skal instituttet følge opp trekk ved vitenskapelig og militærteknisk utvikling som kan påvirke forutsetningene for sikkerhetspolitikken eller forsvarsplanleggingen.

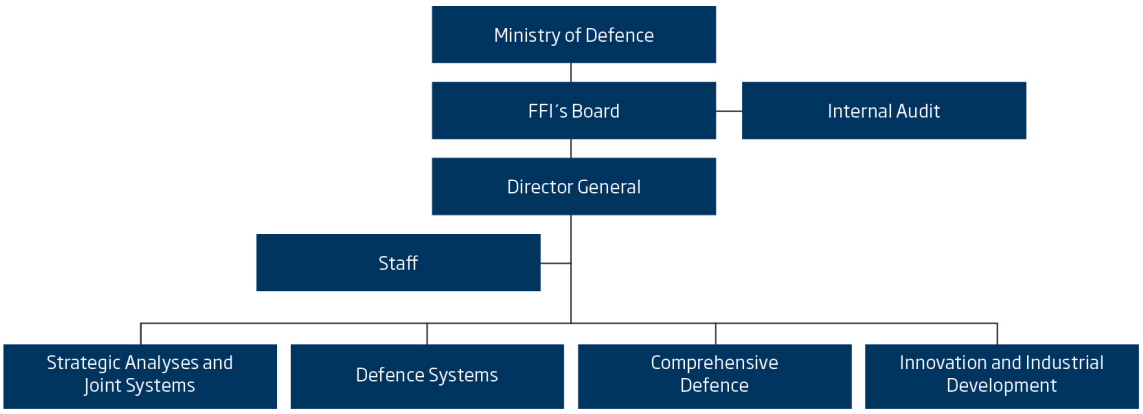
FFIs VISJON

FFI gjør kunnskap og ideer til et effektivt forsvar.

FFIs VERDIER

Skapende, drivende, vidsynt og ansvarlig.

FFI's organisation



Forsvarets forskningsinstitutt
Postboks 25
2027 Kjeller

Besøksadresse:
Instituttveien 20
2007 Kjeller

Telefon: 63 80 70 00
Telefaks: 63 80 71 15
Epost: ffi@ffi.no

Norwegian Defence Research Establishment (FFI)
P.O. Box 25
NO-2027 Kjeller

Office address:
Instituttveien 20
N-2007 Kjeller

Telephone: +47 63 80 70 00
Telefax: +47 63 80 71 15
Email: ffi@ffi.no

FFI Forsvarets forskningsinstitutt
Norwegian Defence Research Establishment