TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Marlene Ibrus

Konfliktivabade dubleeritud andmetüüpide formaliseerimine

Bakalaureusetöö (9 EAP)

Juhendaja:
Danel Ahman, PhD

Tartu 2025

Konfliktivabade dubleeritud andmetüüpide formaliseerimine

Lühikokkuvõte:

Töös kirjeldatakse ja formaliseeritakse konfliktivabu dubleeritud andmetüüpe, mis on oluline komponent hajussüsteemides, kus andmete kooskõlalisus peab olema tagatud ilma keskse koordineerimiseta. Samuti formaliseeritakse ja analüüsitakse kolme erinevat konfliktivabade dubleeritud andmetüüpide disaini, kasutades nende matemaatilise täpsuse tagamiseks teoreemitõestajat Agda.

Võtmesõnad: Hajussüsteemid, CRDT, Agda, formaliseerimine

CERCS: P175 (Informaatika, süsteemiteooria)

Formalising Conflict-free Replicated Data Types

Abstract:

The thesis describes and formalises conflict-free replicated data types, which are an important component in distributed systems where data consistency must be ensured without the need for centralised coordination. Additionally, three different designs of conflict-free replicated data types are formalised and analysed, using the theorem prover Agda to ensure their mathematical precision.

Keywords: Distributed systems, CRDT, Agda, formalising

CERCS: P175 (Informatics, systems theory)

Sisukord

1. Sissejuhatus

Hajussüsteemid on tänapäeval laialt levinud lahendus, sest need võimaldavad mitmes serveris või seadmes paralleelselt andmeid töödelda [1]. Seeläbi parandavad hajussüsteemid süsteemide töökindlust ja jõudlust. Mitmetes hajussüsteemides kasutatakse käideldavuse, töökindluse ja latentsuse parandamiseks andmete dubleerimist [2]. Andmeid võib dubleerida mitme erineva geograafilise asukoha vahel (ingl *geo replication*), et suurendada süsteemi rikketaluvust [2]. Sellises keskkonnas peab arvestama sellega, et duplikaadid võivad teha kohalikke muudatusi eri järjekorras, mis võivad põhjustada konflikte. Selliste konfliktide vältimiseks on vaja mehhanismi, mis aitab säilitada andmete koherentsust.

Konfliktivabad dubleeritud andmetüübid on hajussüsteemides kasutatavad abstraktsed andmetüübid, mis tagavad andmetes konfliktide puudumise, ilma et süsteemi töösse peaks sekkuma [3]. Teisisõnu võimaldavad need andmetüübid hajutatud andmete automaatset ja deterministlikku sünkroniseerimist ilma keskse koordineerimiseta [3]. Nende tugevad matemaatilised omadused, nagu kommutatiivsus ja assotsiatiivsus, tagavad selle, et kõik süsteemi osad jõuavad lõpuks samasse olekusse, olenemata muudatuste järjekorrast.

Bakalaureusetöö eesmärk on formaalselt kirjeldada ja tõestada konfliktivabade dubleeritud andmetüüpide omadusi, kasutades selleks programmeerimiskeelt ja tõestusassistenti Agda. Töö raames formaliseeritakse Agdas kolm konfliktivabade dubleeritud andmetüüpide näidet, milleks on loendurid, registrid ja hulgad. Agda sõltuvalt tüübitud süsteemi abil modelleeritakse nende struktuuride olekud, operatsioonid ja käitumisreeglid ning tõestatakse nende olulisemad omadused, nagu näiteks konfliktide puudumine samaaegsete sõnumite rakendamisel. Töö eesmärgiks on pakkuda usaldusväärne alus CRDT-de korrektsuse formaalseks kontrolliks. Töö tulemusena valminud Agda teek on avaldatud Zenodos [4].

Bakalaureusetöö on jagatud neljaks suuremaks peatükiks. Peatükis 2 antakse ülevaade hajussüsteemidest, konfliktivabadest dubleeritud andmetüüpidest ja nende matemaatilistest omadustest. Peatükis 3 tutvustatakse programmeerimiskeelt Agda ja kirjeldatakse selle keele olulisemaid omadusi. Peatükis 4 kirjeldatakse Agdas formaliseeritud konfliktivabade dubleeritud andmetüüpide üldist struktuuri. Peatükis 5 esitatakse Agdas formaliseeritud näited.

2. Teoreetiline taust

Selles peatükis tutvustatakse konflikivabu replikeeritud andmetüüpe ja nendega seotud olulisi mõisteid, mis on aluseks nende andmestruktuuride ülesehitusele. Lisaks on need mõisted kasutuses formaliseerimise käigus tehtud valikute seletamiseks.

2.1 Hajussüsteemid

Hajussüsteem (ingl *distributed system*) on füüsiliselt lahutatud arvutite kogum, mille komponendid suhtlevad omavahel andmeside võrgu kaudu [1]. Protsessid jaotatakse hajussüsteemi füüsiliste võrgusõlmede (ingl *node*) või hostide (ingl *host*) vahel, et kasutada mitme arvuti ressursse [1]. Hajussüsteemi võrgusõlmed võivad olla teiste võrgusõlmedega osaliselt või täielikult ühendatud [1]. Selles töös eeldatakse, et hajussüsteemi võrgusõlmed on omavahel täielikult ühendatud ja et võrgukiht garanteerib sõnumite kausaalse kohalejõudmise.

2.2 Andmete dubleerimine

Üks viis andmete käideldavuse ja liiasuse (ingl *redundancy*) saavutamiseks hajussüsteemides on hoida süsteemi sõlmedes andmekogumi koopiaid [5]. Andmete dubleerimine (ingl *replication*) on viis andmete sünkroniseerimiseks süsteemi erinevate hostide vahel, tõstes seeläbi andmete liiasust süsteemis [1]. Liiasus saavutatakse andmete kopeerimisega mitmesse asukohta. Duplikaatide olemasolu aitab süsteemis tagada elastsust, käideldavust ja veatõrjet. Lisaks suurendab see süsteemi usaldusväärsust, sest andmetele pääseb ühe hosti rikke puhul teiste hostide kaudu järjepidevalt ligi.

2.3 Lõppkokkuvõttes tugev kooskõlalisus

Shapiro jt kohaselt on andmete duplikaadid lõppkokkuvõttes kooskõlalised (ingl *eventually consistent*) (lüh EC) juhul, kui nad vastavad kolmele tingimusele [3]. Need kolm tingimust on järgmised:

- Lõppkokkuvõttes kohaletoimetamine (ingl *eventual delivery*): kui suvaline uuendus jõuab ühe veavaba duplikaadini, siis jõuab see uuendus kunagi kõigi veavabade duplikaatideni.
- Koonduvus (ingl *convergence*): kui veavabad duplikaadid on edasi andnud sama uuenduse, siis jõuavad need duplikaadid lõpuks ekvivalentsesse olekusse.
- Lõppemine (ingl *termination*): kõik duplikaatidele rakendatavate funktsioonide rakendamised lõppevad kunagi.

Seega tähendab EC omadus seda, et kui uuendused peatuvad, siis jõuavad kõik duplikaadid kunagi kindlalt samasse olekusse [3]. EC omadus ei välista konfliktide tekkimist juba tehtud uuenduste vahel [3]. Konfliktide vältimiseks kasutatakse omaduse tugevamat versiooni, lõppkokkuvõttes tugeva kooskõlalisuse omadust (ingl *strong eventual consistency*) (lüh SEC), mis tagab, et sama uuenduse saanud veavabad duplikaadid jäävad samasse olekusse [3].

2.4 Konfliktivabad dubleeritud andmetüübid

Konfliktivabad dubleeritud andmetüübid (ingl *Conflict-free Replicated Data Types*) (lüh CRDT-d) on lõppkokkuvõttes tugevalt kooskõlaliste andmete klass, kus lahendatakse duplikaatide vahelisi konflikte automaatselt. CRDT-d on sobilikud kasutamiseks hajus- ja geo-dubleeritud süsteemides, kuna need põhinevad lihtsatel matemaatilistel omadustel, mis tagavad andmete käideldavuse ja usaldusväärsuse [3]. Näiteks kasutab CRDT-sid koos töötamist võimaldav disainitööriist Figma[1]. Kaks peamist CRDT-de klassi on olekupõhised CRDT-d ja operatsioonipõhised CRDT-d. Olekupõhiste CRDT-de kasutamine ei nõua võrgukihi idempotentsust, kuid vajab suuremat võrgukihi läbilaskevõimet. Operatsioonipõhiste CRDT-de kasutamine eeldab, et iga uuendus jõuab kohale ühe korra, kuid ei koorma võrku nii palju. Kahe CRDT-de klassi eeliste ühendamiseks on olemas ka hübriidlahendused, näiteks δ-CRDT-d [6]. Operatsioonipõhiste CRDT-de ja olekupõhiste CRDT-de kasutamine on samaväärne, kuna esimest tüüpi CRDT käitumist saab imiteerida teist tüüpi CRDT-ga ja vastupidi [3]. Selles töös keskendutakse operatsioonipõhiste CRDT-de formaliseerimisele.

2.5 Operatsioonipõhised konfliktivabad dubleeritud andmetüübid

Operatsioonipõhised konfliktivabad dubleeritud andmetüübid (ingl *Operation-Based Conflict-free Replicated Data Type*) (lüh operatsioonipõhised CRDT-d) on CRDT-de klass, kus duplikaadid jagavad sõnumitega omavahel enda olekule jadamisi rakendatud muudatusi, mitte muutunud olekuid [2]. Kui operatsioonid on omavahel kommutatiivsed, siis ei ole operatsioonide järjekorda vaja määrata [2]. Vastasel juhul tuleb konkurentsete operatsioonide jaoks määrata konfliktide lahendamise algoritm. Erinevalt olekupõhistest CRDT-dest ei vaja operatsioonipõhised CRDT-d nii suurt arvutivõrgu läbilaskevõimet [6]. Üheks olekupõhiste CRDT-de kasutamise miinuseks on see, et süsteem peab tagama, et üks sõnum jõuab kohale ainult ühe korra, mis on osades süsteemides keeruline.

[1] https://www.figma.com/blog/how-figmas-multiplayer-technology-works/

3. Agda

Selles peatükis tutvustatakse programmeerimiskeelt Agda ja selle põhifunktsionaalsust. Agda on sõltuvalt tüübitud funktsionaalne programmeerimiskeel ja tõestusassistent, mis põhineb Martin-Löfi tüübiteoorial ja Curry-Howardi vastavusel, mille kohaselt loogilised väited vastavad tüüpidele ning programmid nende tõestustele [7]. Agda sobib nii programmide kirjutamiseks kui ka formaalsete tõestuste esitamiseks. Keel toetab moodulitena organiseerimist, UTF-8 sümboleid ning eraldi kompileerimist, mis teeb suuremate projektide haldamise lihtsamaks [7]. Rohkem teavet leiab Agda veebilehelt [2].

Agda on funktsionaalne programmeerimiskeel, mis sarnaneb osaliselt teiste funktsionaalsete programmeerimiskeeltega nagu Haskell ja Coq [7]. Seega saab Agdas funktsioone argumentidena anda teistele funktsioonidele ja koostada puhtaid funktsioone, millel puuduvad kõrvalefektid. Programmeerimiskeele funktsionaalsus võimaldab andmetüüpide rekursiivset defineerimist, mustrituvastusel põhinevat funktsioonide määratlemist ja sõltuvate tüüpide kasutamist tõestuste kirjutamiseks [8]. Üks näide Agda funktsionaalsusest on naturaalarvude definitsioon [9].

```
data ℕ : Set where
  zero : ℕ
  succ : ℕ → ℕ
```

Selline definitsioon näitab, kuidas Agdas defineeritakse andmetüüpe rekursiivselt. Funktsioonid, mis opereerivad selliste andmetega, on määratud mustrituvastuse ja rekursiooni abil, järgides funktsionaalse programmeerimise põhimõtteid [7]. Üks selline rekursiivne funktsioon on naturaalarvude liitmine [9].

```
_+_ : ℕ → ℕ → ℕ
zero + n = n
(succ m) + n = succ (m + n)
```

Funktsiooni baasjuhuks on nullile naturaalarvu `n` liitmine, mis tagastab naturaalarvu `n`. Kui liidetakse naturaalarvust `m` järgmine arv `succ m` naturaalarvuga `n`, siis on tulemuseks `m` ja `n` summast ühe võrra suurem naturaalarv `succ (m + n)`.

[2] https://wiki.portal.chalmers.se/agda/pmwiki.php

Agda pakub sõltuvate tüüpide kasutamist, mis võimaldavad kirjutada paindlikke ja täpselt määratletud funktsioone. Sõltuvad tüübid annavad võimaluse luua polümorfseid funktsioone ja andmestruktuure, kus funktsiooni käitumine sõltub sisendi tüübist [8]. Levinud näide sõltuvate tüüpidega andmestruktuurist Agdas on fikseeritud pikkusega loend ehk vektor [8].

```
data Vec (A : Set) : ℕ → Set where
  [] : Vec A zero
  _::_ : {n : ℕ} → A → Vec A n → Vec A (succ n)
```

Iga `Vec A n` kirjeldab loendit, mis sisaldab täpselt `n` elementi tüübist `A` [8]. Argument `{n : Nat}` on kaudne (ingl *implicit*), ehk seda ei pea andmetüübi konstruktorile ette andma, vaid Agda määrab selle teiste argumentide ja nende tüüpide põhjal automaatselt.

Tüüp _≡_ defineerib Agdas väärtuste omavahelise võrdusrelatsiooni.

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

Agdas on võrdus ekvivalentsiseos ja seega peab rahuldama refleksiivsuse, sümmeetrilisuse ja transitiivsuse omadusi [9]. Seda, et võrdus on sümmeetriline, on võrdlemisi lihtne tõestada.

```
sym : ∀ {A : Set} {x y : A}
  → x ≡ y → y ≡ x
sym refl = refl
```

Tõestus `sym` võtab argumendiks ekvivalentsuse `x ≡ y`, seega `x` ja `y` on samaväärsed ja argument taandub refleksiivsusele [9]. Jääb tõestada, et `x ≡ x`, mis on refleksiivsus.

Üks kasulik Agda konstruktsioon on `with` lause [9]. Märksõnale `with` järgneb avaldis ja vähemalt üks rida, mille lõppu tuleb püstkriipsuga `|` eraldatud avaldise mustrituvastus. Lihtne näide `with` lause kasutamisest on loendi filtreerimine.

```
filter : {A : Set} → (A →Bool) → List A → List A
filter p [] = []
filter p (x :: xs) with p x
...              | true = x :: filter p xs
...              | false = filter p xs
```

`with` lause kasutamine avaldise mustrituvastuse tegemiseks on samaväärne mustrituvastuse abil defineeritud abifunktsiooni kasutamisega.

4. CRDT-d ja nende formalseerimine

Töö tulemusena valmis Agda teek [4], mis sisaldab CRDT-de alusfaili, mille põhjal formaliseeriti kolm näidis-CRDT-d. Selles peatükis antakse ülevaade formaliseeritud CRDT-de alusfailist, mida kasutatakse näidis-CRDT-de formaliseerimiseks, ja seletatakse kõige olulisemaid formaliseerimise käigus tehtud otsuseid.

4.1 CRDT-de struktuur

Weidneri jt sõnul [2] võib operatsioonipõhiste CRDT-de klassi esitada ennikuna $(\Sigma, \sigma^0, \mathit{prepare}, \mathit{effect}, \mathit{eval})$, kus Σ on andmetüübi olekute hulk, σ^0 on andmesõlme duplikaadi algolek, $\mathit{prepare}\ o\ \sigma\ r$ on meetod, mis valmistab ette sõnumi m vastavalt operatsioonile o, andmesõlme hetkeolekule σ ja duplikaadile r, $\mathit{effect}\ m\ \sigma$ on osaline funktsioon, mis rakendab ettevalmistatud sõnumit m andmesõlme hetkeolekule σ ja $\mathit{eval}\ \sigma\ q$ teostab olekul σ päringu q, mis tagastab andmesõlme hetkeolekule muudatusi tegemata päringu q tulemuse.

Joonis 1 illustreerib Agdas formaliseeritud CRDT andmetüübi struktuuri `CRDT`, mis põhineb Weidneri jt definitsioonil [2]. Formaliseerimine toimub Agda `record` tüüpi andmiku abil, mis võimaldab koondada omavahel seotud tüübid ja funktsioonid ühtsesse struktuuri. CRDT-de kontekstis tähistab see andmik mitme duplikaadiga hajussüsteemi mudelit, kus igal duplikaadil on oma lokaalne olek ja identifikaator. Andmik `CRDT` hoiab endas suvalist tüüpi andmeid `Data`, mille elementide võrdus $\mathtt{d} \equiv^{\mathtt{d}} \mathtt{d'}$, peab olema ekvivalents ja tuleb igat tüüpi andmete jaoks eraldi tõestada. CRDT-des hoitavatele andmetele `Data` kehtib nõue, et andmete hulgas leidub algne element. Hilisemate tõestuste lihtsustamiseks on andmesõlme olek `State` defineeritud tuginedes duplikaadi identifikaatorit sisaldavale olekule `RState`. Niimoodi sisaldab iga olek $\mathtt{s}$ `at` $\mathtt{r}$ andmete väärtust $\mathtt{s}$ ja andmesõlme identifikaatorit $\mathtt{r}$. Lisaks on defineeritud abifunktsioonid `replica-of` ja `state-of`, mis tagastavad vastavalt andmesõlme identifikaatori $\mathtt{r}$ ja oleku väärtuse $\mathtt{s}$. Samuti on defineeritud kahe oleku võrdsus $\mathtt{s} \equiv^{\mathtt{s}} \mathtt{s'}$, mille jaoks peavad kahe oleku $\mathtt{s}$ ja $\mathtt{s'}$ väärtused ja andmesõlme identifikaatorid võrdsed olema. Kahe oleku võrdsus $\mathtt{s} \equiv^{\mathtt{s}} \mathtt{s'}$ on ekvivalentsusseos.

Andmik `CRDT` sisaldab lisaks ka duplikaatide oleku muutmise operatsioone `Operation` ja duplikaatide vahel saadetavaid sõnumeid `Message`, mis tuleb iga formaliseeritud CRDT jaoks eraldi defineerida. `initial` loob iga andmesõlme jaoks algoleku, mis koosneb varem mainitud andmete algsest väärtusest ja andmesõlme identifikaatorist. `prepare` vastab allikas [2] kirjeldatud funktsioonile $\mathit{prepare}\ o\ \sigma\ r$ ja valmistab ette sõnumi operatsiooni `Operation`,

```
data RState (S : Set) : Set where
  _at_ : (s : S) → (r : Replica) → RState S

replica-of : ∀ {S} → RState S → Replica
replica-of (s at r) = r

state-of : ∀ {S} → RState S → S
state-of (s at r) = s

-- Raw CRDTs (that do not have to be well-behaved)

record CRDT : Set₁ where
  constructor
    crdt
  field
    Data : Set
    _≡ᵈ_ : Data → Data → Set
    isEquivalence-≡ᵈ : IsEquivalence _≡ᵈ_

  State : Set
  State = RState Data

  _≡ˢ_ : State → State → Set
  s ≡ˢ s' =
    state-of s ≡ᵈ state-of s' × replica-of s ≡ replica-of s'

  isEquivalence-≡ˢ : IsEquivalence _≡ˢ_
  isEquivalence-≡ˢ = record {
    refl  =
      ≈-refl isEquivalence-≡ᵈ , refl ;
    sym   =
      λ { (p , q) → ≈-sym isEquivalence-≡ᵈ p , sym q } ;
    trans =
      λ { (p , q) (p' , q') → ≈-trans isEquivalence-≡ᵈ p p' , trans q q' } }

  field
    Operation : Set
    Message : Set
    initial : Replica → State
    prepare : Operation → State → Message
    _·_  : Message → State → Maybe State
    Query : Set
    View : Set
    eval : Query → State → View

    property1 : ∀ {r} → replica-of (initial r) ≡ r
    property2 : ∀ {m s s'} → m · s ≡ just s' → replica-of s ≡ replica-of s'

  _≡ᵐ_ : Maybe State → Maybe State → Set
  just s ≡ᵐ just s' = s ≡ˢ s'
  just s ≡ᵐ nothing = ⊥
  nothing ≡ᵐ just s' = ⊥
  nothing ≡ᵐ nothing = ⊤

  ≡ᵐ-refl : ∀ (s : Maybe State) → s ≡ᵐ s
  ...

  ≡ᵐ-sym : ∀ {s s' : Maybe State} → s ≡ᵐ s' → s' ≡ᵐ s
  ...

  ≡ᵐ-trans : ∀ {s s' s''} → s ≡ᵐ s' → s' ≡ᵐ s'' → s ≡ᵐ s''
  ...

  isEquivalence-≡ᵐ : IsEquivalence _≡ᵐ_
  isEquivalence-≡ᵐ = record {
    refl = λ {s} → ≡ᵐ-refl s ;
    sym = λ {s} {s'} eq → ≡ᵐ-sym {s} {s'} eq ;
    trans = λ {s} {s'} {s''} eq₁ eq₂ → ≡ᵐ-trans {s} {s'} {s''} eq₁ eq₂ }
```

Joonis 1. Agdas formaliseeritud CRDT-de struktuur ilma korrektsuse omadusteta

andmesõlme hetkeoleku `State` ja andmesõlme identifikaatori `Replica` põhjal. Olekule σ sõnumi m rakendamise funktsiooni `m · s` tulemuse tüübiks on `Maybe State`, ehk funktsioon `m · s` on osaline funktsioon. Funktsiooni `m · s` formaliseerimine osalise funktsioonina peegeldab olukorda, kus igat duplikaatide vahel saadetavat sõnumit m ei pruugi olla võimalik rakendada olekule σ. `Query` vastab päringu q tüübile ja `View` vastab päringu tulemuse tüübile. Päringu sooritamiseks on nõutud funktsioon `eval`, mis vastab funktsioonile $eval\ \sigma\ q$.

Formaliseeritud CRDT-de veatuks töötamiseks sisaldab andmik `CRDT` kahte lisanõuet, mida on kujutatud joonisel 1. Need lisanõuded on `property1` ja `property2`. Lisanõue `property1` nõuab, et iga CRDT `r` duplikaadi identifikaator ei muutu duplikaadi olekule sõnumeid rakendades. Teisisõnu, iga andmesõlme identifikaator `r` on sama mis selle andmesõlme algne identifikaator: `replica-of (initial r)`$\equiv$ `r`. Lisanõue `property2` näeb ette, et kui andmesõlmes on ettevalmistatud sõnum, siis selle sõnumi rakendamine antud andmesõlme olekule ei muuda andmesõlme duplikaadi identifikaatorit. Need kaks omadust tagavad selle, et süsteemi töö käigus ei muutu ühegi andmesõlme identifikaator, mis omakorda tähendab, et duplikaate saab omavahel järjestada.

4.2 Sõnumite ja olekute ajalood

Weidneri jt sõnul [2] tähendab osaline järjestus $\prec$ kahe sõnumi m_1 ja m_2 puhul seda, et kui $m_1 \prec m_2$, siis sõnumi m_2 saatnud duplikaat on varasemalt kätte saanud sõnumi m_1 või ise eelnevalt sõnumi m_1 teistele süsteemi duplikaatidele saatnud. Selle põhjal on kaks sõnumit m_1, m_2 konkurentsed (ingl *concurrent*) ehk samaaegsed, juhul kui $m_1 \not\prec m_2$ ja $m_2 \not\prec m_1$ [2]. Järelikult peavad sõnumid m_1 ja m_2 olema saadetud erinevate duplikaatide poolt. Lisaks ei tohi sõnumi m_1 saatnud duplikaat sisaldada endas sõnumist m_2 saadud informatsiooni ja vastupidi. Selleks, et määrata, kas mingi duplikaat r on suvalise sõnumi m oma olekule rakendanud või selle ise saatnud, on vaja määrata, millisest duplikaadist sõnum m pärineb ja milliseid sõnumeid sõnumi m saatnud duplikaat enda olekule rakendanud on.

Sõnumite konkurentsuse ja samaaegse rakendamise kommutatiivsuse väljendamiseks on CRDT-de struktuuris defineeritud sõnumite ja olekute ajalood, mille formaliseerimist Agdas illustreerib joonis 2. Ajalugude defineerimiseks kasutatakse Agda `mutual` süntaksit, mis võimaldab ühiselt määratleda andmetüüpe ja funktsioone[3]. Sõnumi ajalugu `MessageHistory` sisaldab duplikaati

[3] https://agda.readthedocs.io/en/latest/language/mutual-recursion.html

```
MessageHistory : Set
MessageHistory = Σ[ r ∈ Replica ] (Operation × StateHistory r)

MessageWithHistory : Set
MessageWithHistory = Message × MessageHistory

data StateHistory (r : Replica) : Set where
  hemp : StateHistory r
  heff : (m : MessageWithHistory)
       → (sh : StateHistory r)
       → (heq : mreplica-of (mhistory-of m) ≡ r
         → shistory-of (mhistory-of m) ≡  sh)
       → StateHistory r
```

Joonis 2. Sõnumite ja olekute formaliseeritud ajalood

`r`, operatsiooni ja selle duplikaadi oleku ajalugu. Sõnum koos ajalooga `MessageWithHistory` koosneb sõnumist ja selle ajaloost. Kui sõnum rakendatakse duplikaadi olekule, tekib uus oleku ajalugu `StateHistory`. Selle formaliseerimine järgib allikas [2] kirjeldatud Algoritm 1 struktuuri, mille kohaselt rakendab duplikaat enda loodud sõnumi kohe iseenda olekule. Kui olek on algolek, on ajalugu tühi. Vastasel juhul, kui sõnumi ja oleku duplikaadid langevad kokku, peavad sõnumi ja oleku ajalood olema kooskõlas.

Joonisel 2 on kasutatud abifunktsioone `message-of`, `mhistory-of`, `mreplica-of` ja `shistory-of`, millega eraldatakse sõnumi ajaloost oleku ajaloo formaliseerimiseks vajalikud osad. Need funktsioonid tagastavad vastavalt sõnumi, selle ajaloo, duplikaadi, operatsiooni ja oleku ajaloo. Abifunktsioonid on defineeritud analoogselt joonisel 1 kujutatud funktsioonidele `replica-of` ja `state-of`.

CRDT-de struktuuri andmikus on defineeritud induktiivsed andmetüübid `m` $\bowtie^{s}$ `mh` ja `s` $\bowtie^{s}$ `[s , sh]`, et siduda sõnumeid ja olekuid vastavalt nende ajalugudega. Joonis 3 kujutab relatsiooni `m` $\bowtie^{m}$ `mh`, mis seob sõnumi `m` talle vastava ajalooga `mh`. Juhul kui sõnum `m` on valminud joonisel 1 kujutatud `prepare` funktsiooni tulemusena, siis sõnumi `m` ajalugu koosneb oleku `s` duplikaadist `replica-of s`, operatsioonist `o` ja oleku `s` ajaloost `sh`. Joonis 3 kujutab lisaks ka induktiivset andmetüüpi `s` $\bowtie^{s}$ `[s , sh]`, mis seob duplikaadi oleku `s` oleku ajalooga `sh`. Juhul kui olek `s` on duplikaadi `r` algolek, siis on olek `initial r` seotud tühja oleku ajalooga. Kui olek `s` ei ole algolek, siis olek `s`, mis on saadud sõnumi `m` rakendamisel ajalooga seotud olekule `s'`, vastab `s'` ajaloole, enne sõnum `m` lisamist.

```
data _⋈ᵐ_ : Message → MessageHistory → Set where
  ⋈ᵐ-prepare : {m : Message}
             → {o : Operation}
             → {s : State}
             → {sh : StateHistory (replica-of s)}
             → prepare o s ≡ m
             → s ⋈ˢ[ replica-of s , sh ]
             ----------------------------------
             → m ⋈ᵐ (replica-of s , o , sh)

data _⋈ˢ[_,_] : State → (r : Replica) → StateHistory r → Set where
  ⋈ˢ-emp : {r : Replica}
         ----------------------------------
         → initial r ⋈ˢ[ r , hemp ]
  ⋈ˢ-eff : {s s' : State}
         → {r : Replica}
         → {sh : StateHistory r}
         → {m : MessageWithHistory}
         → {heq : mreplica-of (mhistory-of m) ≡ r
           → shistory-of (mhistory-of m) ≡  sh}
         → (message-of m · s') ≡ just s
         → message-of m  ⋈ᵐ mhistory-of m
         → s' ⋈ˢ[ r , sh ]
         ----------------------------------
         → s ⋈ˢ[ r , heff m sh heq ]
```

Joonis 3. Sõnumite ajaloo sidumine sõnumiga ja oleku ajaloo sidumine olekuga

4.3 Hästi käituvad CRDT-d

Weidneri jt [2] seavad operatsioonipõhistele CRDT-dele nõuded, et kehtiks järgmised kaks omadust:

i) Kõikide sõnumite m puhul, kui $m = prepare(o, \sigma, r)$, siis $m \cdot \sigma \neq \bot$. Teisisõnu ettevalmistatud sõnumit m olekule σ rakendades ei teki konflikti.

ii) Kõikide olekute σ ja kahe sõnumi m_1, m_2 puhul, kui olek σ võib esineda kopeeritud olekuna ja sõnumid m_1, m_2 võivad rakenduda olekule σ samaaegselt, siis juhul kui $m_1 \cdot \sigma \neq \bot$ ja $m_2 \cdot \sigma \neq \bot$, kehtib $m_1 \cdot (m_2 \cdot \sigma) = m_2 \cdot (m_1 \cdot \sigma) \neq \bot$. Siin on $m \cdot \sigma$ olekule σ sõnumi m rakendamise tulemus. Teisisõnu on kahe konkurentse sõnumi m_1, m_2 rakendamine olekule σ alati konfliktivaba ning nende sõnumite rakendused kommuteeruvad.

Olekupõhiste CRDT-de niimoodi defineerimine tagab selle, et samaaegselt toimunud operatsioonide sõnumid saadetakse üheaegselt. Seega on omadustele i) ja ii) vastav CRDT konfliktivaba, kuna konkurentsete sõnumite puhul on nõutud, et need oleksid duplikaadi olekule rakendatavad. Sellist CRDT-d saab kutsuda hästi käituvaks CRDT-ks. Selleks, et lisada nõuded i) ja ii) Agdas formaliseeritud CRDT struktuurile, peab struktuuri lisama predikaadi sõnumite ajalugude määramiseks.

```
data _∉_ (mh : MessageHistory) : ∀ {r'} → StateHistory r' → Set where

  ∉-hEmp : {r' : Replica}
         → mreplica-of mh ≢ r'
         ------------------------------------
         → mh ∉ hemp {r'}
  ∉-hEff : {r' : Replica}
         → {sh' : StateHistory r'}
         → {m' : MessageWithHistory}
         → {heq : mreplica-of (mhistory-of m') ≡ r'
           → shistory-of (mhistory-of m') ≡   sh'}
         → mreplica-of mh ≢ mreplica-of (mhistory-of m')
         → mh ∉ shistory-of (mhistory-of m')
         → mh ∉ sh'
         ------------------------------------
         → mh ∉ heff m' sh' heq
```

Joonis 4. Sõnumi ajalugu ei kuulu oleku ajalukku

```
[_,_,_]||[_,_,_] : (m₁ : Message)
                 → (mh₁ : MessageHistory)
                 → (p : m₁ ⋈ᵐ mh₁)
                 → (m₂ : Message)
                 → (mh₂ : MessageHistory)
                 → (q : m₂ ⋈ᵐ mh₂)
                 → Set

[ m₁ , (r₁ , o₁ , h₁) , p ]||[ m₂ , (r₂ , o₂ , h₂) , q ] =
  ((r₂ , o₂ , h₂) ∉ h₁) × (((r₁ , o₁ , h₁) ∉ h₂))
```

Joonis 5. Sõnumite konkurentsuse predikaat

Sõnumite konkurentsuse formaliseerimiseks on vaja formaliseerida induktiivne predikaat `mh ∉ sh` mis tähendab seda, et sõnumi ajalugu `mh` ei sisaldu oleku ajaloos `sh`. Predikaadi definitsiooni baasiks on juht, kui oleku ajalugu `sh` on tühi ja sõnumi ajaloo duplikaat `mreplica-of mh` ei ole võrdne oleku duplikaadiga `r`. Kui oleku ajalugu `sh` ei ole tühi, siis koosneb oleku ajalugu `sh` sõnumist `m'` ja eelmise oleku ajaloost `sh'`. Predikaat `mh ∉ sh` kehtib juhul, kui sõnumi `mh` ajaloo duplikaat `mreplica-of mh` ei ole võrdne sõnumi `m'` ajaloo `mreplica-of (mhistory-of m)'` duplikaadiga ja sõnumi ajalugu `mh` ei sisaldu sõnumi `m'` ajaloo oleku ajaloos `shistory-of (mhistory-of m)` ja sõnumi ajalugu `mh` ei kuulu oleku ajalukku `sh'`.

Predikaat `mh ∉ sh` aitab formaliseerida kahe sõnumi konkurentsust läbi nende ajalugude. Joonis 5 kujutab Agdas formaliseeritud kahe sõnumi konkurentsuse predikaati `[m₁ , mh₁ , p]||[m₂, mh₂ , q]`. Antud on sõnum m_1, selle juurde kuuluv ajalugu mh_1 ja predikaat `p`, mille kohaselt $m_1 \bowtie^m mh_1$. Samuti on antud sõnum m_2, tema ajalugu mh_2 ja predikaat `q`, mille

```
Property3 : Set
Property3 = ∀ {o : Operation} {s : State}
          → Σ[ s' ∈ State ] ((prepare o s · s) ≡ᵐ just s')

Property4 : Set
Property4 = ∀ {m₁ m₂ : Message} {s s₁ s₂ s₁₂ s₂₁ : State}
              {mh₁ mh₂ : MessageHistory} {p : m₁ ⋈ᵐ mh₁} {q : m₂ ⋈ᵐ mh₂}
          → ([ m₁ , mh₁ , p ]||[ m₂ , mh₂ , q ])
          → (m₁ · s ≡ just s₁)
          → (m₂ · s ≡ just s₂)
          → Σ[ s₂₁ ∈ State ]
            Σ[ s₁₂ ∈ State ]
            ( m₂ · s₁ ≡ just s₂₁ ×  m₁ · s₂ ≡ just s₁₂ × s₂₁ ≡ˢ s₁₂ )
```

Joonis 6. Formaliseeritud CRDT lisanõuded

kohaselt $m_2 \bowtie^m mh_2$. Predikaat `[m₁ , mh₁ , p]||[m₂, mh₂ , q]` kehtib juhul, kui kumbki ajalugu ei sisaldu teineteises. Seda kontrollitakse predikaatide $mh_1 \notin mh_2$ ja $mh_2 \; mh\notin_1$ abil.

Weidneri jt [2] defineeritud omaduste i) ja ii) formaliseerimist Agdas illustreerib joonis 6. Eelnevalt defineeritud lisanõuete `property1` ja `property2` nummerdamise tõttu on formaliseeritud omaduse i) nimi `Property3` ja omaduse ii) nimi `Property4`. `Property3` on predikaat, mis nõuab, et duplikaadis ettevalmistatud sõnumi rakendamine olekule `s` ei tekita konflikti. `Property4` nõuab, et kahe konkurentse sõnumi m_1 ja m_2 rakendamine olekule `s` annab sama tulemuse sõltumata sõnumite rakendamise järjekorrast. Teisisõnu, konkurentsete sõnumite m_1 ja m_2 olekule `s` on defineeritud ja ei tekita konflikte.

```
record WBCRDT : Set₁ where
  constructor
    wbcrdt

  field
    D : CRDT

  open CRDT D
  open WellBehavedCRDT D

  field
    property3 : Property3
    property4 : Property4
```

Joonis 7. Hästi käituv CRDT

Tänu joonisel 6 formaliseeritud omadustele `Property3` ja `Property4` saab Agdas formaliseerida hästi käituvate CRDT-de andmiku, mida on kujutatud joonisel 7. Hästi käituvate CRDT-de andmik `WBCRDT` koosneb joonisel 1 kujutatud CRDT-st `CRDT D` ja allikas [2] kirjeldatud lisanõuetest i) ja ii). Niimoodi formaliseeritud CRDT-de struktuur vastab CRDT-de matemaatilistele nõuetele ja garanteerib sõnumite rakendamise kommutatiivsuse.

5. Näited

Selles peatükis tutvustatakse mõningaid levinumaid CRDT-de konstruktsioone ning nende vastavat formaliseerimist programmeerimiskeeles ja tõestusassistendis Agda. Täpsemalt käsitletakse loendureid, registreid ja hulki, tuues välja iga konstruktsiooni struktuur, tööpõhimõtted ning omadused, mis võimaldavad konfliktivabu uuendusi hajussüsteemides.

5.1 Loendurid

Üks tuntud CRDT-de liike on loendurid, mis lasevad duplikaatidel iseseisvalt ja koordineerimata ühise loenduri väärtust muuta [10]. Selleks kasutatakse loenduri CRDT andmeteks kommutatiivseid monoide, mida muudetakse liitmise operatsioonidega. Kui loenduril defineerida ka andmete eemaldamise operatsioon, siis on loenduris hoitavad andmed Abeli rühm. Üks lihtne näide on täisarvude peal töötav loendur, mille jaoks on defineeritud andmete lisamise ja eemaldamise operatsioonid. Kuna täisarvude liitmine ja lahutamine on kommutatiivsed, siis ei teki duplikaatide uuendamisel konflikte. CRDT loendureid kasutatakse sageli hajusarhitektuuriga rakendustes (ingl *peer-to-peer application*), kus lokaalsed uuendused peavad levima viivitamatult ja usaldusväärselt [11]. Üks erijuht CRDT loenduritest on monotoonsed loendurid, kus väärtust saab ainult suurendada, mis muudab nad eriti sobivaks näiteks sündmuste loendamiseks või kordumatu identifikaatori genereerimiseks.

```
crdtCounter : CRDT
crdtCounter = record {
  Data             = N;
  _≡ᵈ_             = _≡_;
  isEquivalence-≡ᵈ = isEquivalence;
  Operation        = Operation;
  Message          = Message;
  initial          = λ r → 0 at r;
  prepare          = λ { (add n) _ → add-msg n };
  _·_              = λ { (add-msg n) (s at r) → just ((s + n) at r) };
  Query            = N;
  View             = N;
  eval             = λ { _ (s at _) → s };
  property1        = refl;
  property2        = λ { {add-msg x} {s at r} {.(s + x) at .r} refl → refl} }
```

Joonis 8. Monotoonne loendur CRDT

Joonis 8 kujutab Agdas formaliseeritud monotoonset loendurit `crdtCounter`. Selle CRDT andmete `Data` tüübiks on naturaalarvud, kus andmete võrdus `d ≡ᵈ d'` on naturaalarvude võrdsus `n ≡ n'`. Naturaalarve hoidev loendur CRDT on lihtsustatud versioon monotoonsest loendur CRDT-st. Sellise lihtsustamise eesmärk on vältida Agdas täisarvudega töötamist. `Operation` on operatsioonide tüüp, kus iga naturaalarvu `n` puhul on operatsioon liitmine `add n` `Message` on

```
wb-crdtCounter : WBCRDT
wb-crdtCounter = record {
  D         = crdtCounter;
  property3 = property3;
  property4 = property4 }
```

Joonis 9. Hästi käituv loendur CRDT

tüüp, mis koosneb sõnumitest `add-msg n`. `crdtCounter` algolek on iga duplikaadi `r` puhul `0`, ehk loendur algab nullist. Sõnumid `add-msg n` valmistatakse ette `add n` põhjal ja `m · s` rakendab sõnumis `add-msg n` saadetud liitmisoperatsioonid olekule `s`. `Property1` on refleksiivsus ning `Property2` tõestab, et `m · s` uuendab korrektselt olekut `s`.

Joonis 9 visualiseerib hästi käituvat loendur CRDT-d, kus `property3` ja `property4` on peatükis 4 kirjeldatud predikaatide i) ja ii) tõestused naturaalarvulise monotoonse loenduri jaoks.

```
property3 : Property3
property3 {add n} {s at r} = (((s + n)) at r) , refl , refl
```

`Property3` tõestamiseks rakendatakse sõnumit `add n` olekule `s at r`, mille tulemuseks on `(s + n) at r`, millest tuleneb refleksiivselt, et sõnumi `add n` rakendamisel saadud olek on võrdne uue olekuga. `Property4` tõestus tugineb naturaalarvude liitmise omadustele. Selle predikaadi jaoks tõestatakse, et kahe konkurentse sõnumi rakendamine olekule ei põhjusta konflikti. Teisisõnu, kahe konkurentse sõnumi järjestikkuse rakendamise tulemus ei sõltu sõnumite rakendamise järjekorrast. Lisanõudest `Property3` teame, et andmesõlme identifikaator ei muutu, kui andmesõlme olekule operatsiooni rakendatakse. Järelikult peame Agdas näitama ainult seda, et kehtib $(s+n_1)+n_2 = (s+n_2)+n_1$. Alustades sõnumi $\texttt{m}_1$ esimesena rakendamisest, kasutades liitmise assotsiatiivsust (Agdas `+-assoc s n₁ n₂`) $(s + n_1) + n_2 = s + (n_1 + n_2)$. Kasutades võrdluse kongruentsi omadust ja naturaalarvude liitmise kommutatiivsust (Agdas `cong₂ _+_ {s} refl (+-comm n₁ n₂)`) saame, et $s + (n_1 + n_2) = s + (n_2 + n_1)$. Kasutades uuesti liitmise assotsiatiivsust (Agdas `+-assoc s n₂ n₁`) saame, et $s+(n_2+n_1) = (s+n_2)+n_1$. Järelikult kehtib $(s+n_1)+n_2 = (s+n_2)+n_1$. Kuna `Property4` tõestus tugineb naturaalarvude liitmise omadustele, siis ei kasutata selle CRDT korrektsuse tõestamiseks konkurentsuse eeldust. Järelikult on monotoonnse loendur CRDT `Property4` triviaalne, kuna kahe duplikaadi vahel ei saa konflikte tekkida. Agdas formaliseeritud lisaomaduse `Property4` tõestus on esitatud lisas 6.1.

5.2 Registrid

Register CRDT-d on hajussüsteemides laialt kasutusel ja moodustavad olulise osa CRDT-de disainist [10]. Register hoiab ühte väärtust, mida saab kirjutusoperatsioonidega uuendada ja lugeda. Kui kaks kirjutust ei ole konkurentsed, kirjutab uuem kirjutus vanema üle. Levinumad tüübid on viimane kirjutus võidab register (lüh LWW register) ja mitme väärtusega register (lüh MV register) [10]. Selles töös formaliseeritakse näitena ainult LWW register. LWW registris kaasneb iga kirjutusega globaalselt unikaalne identifikaator, milleks on tihti loogiline kell (nt Lamporti kell), mille põhjal määratakse uuem kirjutus. See võimaldab konflikte deterministlikult lahendada ilma keskse koordineerimiseta. LWW registrid sobivad hästi süsteemidesse, kus võib esineda kirjutuste võistlust või replikatsiooniviivitusi. Nende formaliseerimine Agdas võimaldab täpselt modelleerida ja tõestada registrite konfliktivaba käitumist.

Selleks, et Agdas formaliseerida LWW register CRDT-d, on vaja lisaks modelleerida sündmuste ajalist järjestust, kuna hajussüsteemis ei saa sündmuste toimumise aega määrata usaldusväärselt füüsiliste kellade abil. Järelikult tuleb ajalise järjestuse määramiseks kasutada loogilisi kelli. Lamporti kellad on loogilised kellad, mis loovad süsteemi sündmustel osalise järjestuse [12]. Teisisõnu, Lamporti kellad järjestavad süsteemi sündmusi selle põhjal, milline sündmus juhtus enne teist. Lamporti kellad koosnevad kohalikust aja loendurist ja andmesõlme identifikaatorist, mida võrreldakse leksikograafiliselt. Seega on iga Lamporti kell globaalselt unikaalne. Eeldades, et andmesõlmede identifikaatorid on fikseeritult järjestatud, siis saab Lamporti kellasid aja loendurite ja andmesõlmede identifikaatorite põhjal leksikograafiliselt võrrelda. Iga kord, kui protsess tekitab uue sündmuse, näiteks uuendab registri väärtust, suurendatakse lokaalselt selle protsessi Lamporti kellaaega [12]. Lisaks saadetakse iga sõnumiga kaasa saatja kohalik loendur, ning vastuvõtja uuendab oma loendurit maksimaalselt: $\max(t_{\text{oma}}, t_{\text{saatja}}) + 1$ [12]. LWW registri puhul kasutatakse neid kellasid selleks, et määrata, milline kirjutus on uuem, kas kohalik või sõnumist saadud, ning milline väärtus peaks registrisse jääma.

```
data LClock : Set₀ where
    lclock : Time → Replica → LClock
```

Agdas formaliseeritud Lamporti kell `LCLock` koosneb naturaalarvulisest aja loendurist `Time` ja duplikaadi identifikaatorist `Replica`. Selline Lamporti kella definitsioon tagab selle, et iga süsteemi ajatempel on unikaalne, kuna süsteemi duplikaadid on unikaalsed ja aja loendur suureneb pärast igat duplikaadis tehtud operatsiooni. Formaliseeritud Lamporti kella jaoks on defineeritud ka abifunktsioonid `replica-of-clock` ja `time-of-clock`, mis

```
crdtLWW : CRDT
crdtLWW = record {
  Data             = LWWValue;
  _≡ᵈ_             = _≡_;
  isEquivalence-≡ᵈ = isEquivalence;
  Operation        = Operation;
  Message          = Message;
  initial          = initialLWW;
  prepare          = prepareLWW;
  _⊙_              = applyLWW;
  Query            = ⊤;
  View             = S;
  eval             = evalLWW;
  property1        = refl;
  property2        = λ {m} {s} {s'} → property2 {m} {s} {s'} }
```

Joonis 10. LWW register CRDT

on sarnased joonisel 1 kujutatud abifunktsioonidega `replica-of` ja `state-of`. Lisaks on Agdas formaliseeritud Lamporti kellade `lclock` t_1 r_1 ja `lclock` t_2 r_2 võrdlustehe `lclock` t_1 r_1 $<^l$ `lclock` t_2 r_2, mis on defineeritud induktiivselt järgmise kahe juhu kaudu.

```
data _<ˡ_ : LClock → LClock → Set where
  time<  : ∀ {t₁ t₂ : Time} {r₁ r₂ : Replica}
         → t₁ < t₂
         → lclock t₁ r₁ <ˡ lclock t₂ r₂

  replica< : ∀ {t : Time} {r₁ r₁ : Replica}
         → r₁ <ʳ r₂
         → lclock t r₁ <ˡ lclock t r₂
```

Juhul, kui Lamporti kellade naturaalarvulised aja loendurid t_1 ja t_1 on erinevad, siis väiksema loenduriga Lamporti kell on varasem, kui suurema loenduriga Lamporti kell. Juhul, kui kahe kella aja loendurid on võrdsed, siis määratakse kellade järjestus duplikaatide r_1 ja r_2 järjestuse r_1 $<^r$ r_2 järgi. Lisaks on Lamporti kellade võrdlemiseks defineeritud ka otsustusalgoritm $<^l$`-comp`, mille kohaselt on kaks Lamporti kella `lclock` t_1 r_1 ja `lclock` t_2 r_2 kas võrdsed või on üks kell teisest suurem. Otsustusalgoritmi $<^l$`-comp` definitsioon on esitatud lisas 6.2.

Joonis 10 kujutab Agdas formaliseeritud LWW register CRDT-d. Selles CRDT-s hoitakse `LWWValue` tüüpi andmeid.

```
record LWWValue : Set where
  constructor lwwvalue
  field
    value : S
    clock : LClock
```

`LWWValue` on defineeritud andmikuna, millel on kaks andmevälja `value` ja `clock`. Andmeväljas `value` hoitakse tegelikku salvestatud andmeväärtust, mis on suvalist tüüpi `S`. Andmeväljas `clock` hoitakse Lamporti kella tüüpi `LClock`, mis tähistab ajatemplit, millal salvestatud andmeväärtust `value` viimati muudeti. `LWWValue` jaoks on defineeritud ka kaks abifunktsiooni `ltime-of` ja `lstate-of`, mis eraldavad olekust `lwwvalue` vastavalt Lamporti kella naturaalarvulise aja loenduri või andmesõlme oleku hetkeväärtuse. Need abifunktsioonid on defineeritud sarnaselt joonisel 1 kujutatud abifunktsioonidele `replica-of` ja `state-of`. Andmete ekvivalentsuseks `d` $\equiv^{d}$ `d'` kasutatakse Agda standardteegi vaikimisi määratletud võrduspredikaati `_≡_`.

LWW register CRDT-l on üks operatsioon `write val`, mis kirjutab võimalusel andmesõlme hetkeoleku üle. Selle operatsiooni edastamiseks on LWW register CRDT-l ühte tüüpi sõnum `write-msg val cl`, kus `cl` on Lamporti kell ja `val` on väärtus, millega andmesõlme hetkeolekut võimalusel uuendatakse.

`LWWValue` andmetüübis hoitava `S` tüübi kohta eeldatakse, et sellele on määratud mingisugune algväärtus `initial`^{s}. Järelikult saab LWW register CRDT-le määrata algväärtuse `initialLWW`.

```
initialLWW : Replica → RState (LWWValue)
initialLWW r = (lwwvalue initial^s (lclock 0 r)) at r
```

LWW register CRDT algväärtus `initialLWW` koosneb tüübi `S` algväärtusest `initial`^{s} ja Lamporti kellast, mille aja loendur on 0.

Sõnumite ettevalmistamiseks kasutatakse funktsiooni `prepareLWW`.

```
prepareLWW : Operation → RState (LWWValue) → Message
prepareLWW (write v) (lwwvalue _ (lclock t oldRepl) at r)
   = write-msg v (lclock (1 + t) r)
```

`prepareLWW` koostab operatsioonist `write v` ja duplikaadi `r` olekust `lwwvalue _ (lclock t oldRepl) at r` uue sõnumi `write-msg v (lclock (1 + t) r)`, kus `v` on väärtus, millega soovitakse vana väärtust üle kirjutada ja `oldRepl` on duplikaat, mis ajal `t` duplikaadi `r` väärtust üle kirjutas. Funktsiooni `prepareLWW` rakendamine duplikaadile `r` on süsteemi mõttes sündmus, seega sõnumis `write-msg v (lclock (1 + t) r)` on Lamporti kell ühe võrra suurenenud. Lisaks on tulemuseks saadud sõnumi `write-msg v (lclock (1 + t) r)` Lamporti kella duplikaat `r`, kuna sõnum valmistati selles duplikaadis ette. Sõnumi rakendamiseks duplikaadi olekule on defineeritud funktsioon `applyLWW`.

```
applyLWW : Message → RState (LWWValue) → Maybe (RState (LWWValue))
applyLWW (write-msg v' cl') (lwwvalue v cl at r)
   with <ˡ-comp cl' cl
applyLWW (write-msg v' cl') (lwwvalue v cl at r) | tri< p ¬p ¬p'
   = just (lwwvalue v cl at r)
applyLWW (write-msg v' cl') (lwwvalue v cl at r) | tri≈ ¬p p ¬p'
   = just (lwwvalue v cl at r)
applyLWW (write-msg v' cl') (lwwvalue v cl at r) | tri> ¬p ¬p' p
   = just (lwwvalue v' cl' at r)
```

Funktsioon `applyLWW` rakendab Lamporti kellade põhjal olekule `lwwvalue v cl at r` sõnumit `write-msg v' cl'`, kus `v` ja `v'` on `S` tüüpi väärtused, `cl` ja `cl'` on Lamporti kellad ja `r` on duplikaat, mille olekut soovitakse üle kirjutada. Sõnumit `write-msg v' cl'` saab rakendada juhul, kui Lamporti kell `cl'` on suurem kui duplikaadi oleku Lamporti kell `cl`. Kellade `cl'` ja `cl` võrdlemiseks kasutatakse otsustusalgoritmi `<ˡ-comp cl' cl`. Juhul, kui saabunud sõnumi Lamporti kell `cl'` on suurem kui andmesõlme kohalik kell `cl` siis kirjutatakse duplikaadi `r` olek üle sõnumi sisuga ja duplikaadi `r` uueks olekuks saab `lwwvalue v' cl' at r`. Muidu jääb andmesõlme olek samaks.

Duplikaadi oleku väärtuse hindamiseks on defineeritud funktsioon `evalLWW`.

```
evalLWW : ⊤ → RState (LWWValue) → S
evalLWW _ (lwwvalue v _ at _) = v
```

Funktsioon `evalLWW` tagastab suvalises duplikaadis hoitava oleku väärtuse `v`, sõltumata duplikaadist endast või oleku Lamporti kellast.

Joonisel 10 kujutatud LWW register CRDT lisaomadus `Property1` tõestatakse refleksiivsusega, aga lisaomaduse `Property2` tõestamiseks tuleb läbi vaadata `<ˡ-comp cl' cl` tulemusel saadud juhud, sest `Property2` nõuab, et sõnumi `m` rakendamine duplikaadi `r` olekule `s` ei muuda duplikaati `r`.

```
property2 : {m : Message} {s s' : RState LWWValue}
       → applyLWW m s ≡ just s'
       → replica-of s ≡ replica-of s'
property2 {write-msg v' cl'} {(lwwvalue v cl) at r} eq
 with <ˡ-comp cl' cl
property2 {write-msg v' cl'} {lwwvalue v cl at r} refl
 | tri< p ¬p ¬p' =
   refl
```

```
property2 {write-msg v' cl'} {lwwvalue v cl at r} refl
  | tri≈ ¬p p ¬p' =
    refl
property2 {write-msg v' cl'} {lwwvalue v cl at r} refl
  | tri> ¬p ¬p' p =
    refl
```

Otsustusalgoritmi `<ˡ-comp cl' cl` igal võimalikul juhul taandub omaduse `Property2` tõestus võrduse `r ≡ r` tõestamisele, mistõttu on kõik kolm juhtu tõestatavad refleksiivsuse abil.

```
wb-crdtLWW : WBCRDT
wb-crdtLWW = record {
  D         = crdtLWW;
  property3 = property3;
  property4 = property4 }
```

Joonis 11. Hästi käituv LWW register CRDT

Joonis 11 kujutab Agdas formaliseeritud hästi käituvat LWW register CRDT-d, kuhu on joonisel 10 esitatud CRDT-le juurde lisatud lisanõuded i) ja ii). Lisanõude i) ehk `Property3` tõestamiseks on vaja läbi vaadata juht, kus duplikaadi olekut ei kirjutata üle, ja juht, kus olek kirjutatakse üle.

```
property3 : Property3
property3 {write v'} {(lwwvalue v (lclock t oldRepl)) at r}
  with <ˡ-comp (lclock (1 + t) r) (lclock t oldRepl)
property3 {write v'} {lwwvalue v (lclock t oldRepl) at r}
  | tri< p ¬p ¬p'
    = (lwwvalue v (lclock t oldRepl) at r) , refl , refl
property3 {write v'} {lwwvalue v (lclock t oldRepl) at r}
  | tri> ¬p ¬p' p
    = (lwwvalue v' (lclock (1 + t) r) at r) , refl , refl
```

Lisanõue `Property3` väidab, et värskelt loodud sõnumi rakendamine samas duplikaadis, kus see loodi, ei tekita konflikte. Selleks võrreldakse Lamporti kellasid otsustusalgoritmi `<ˡ-comp (lclock (1 + t) r) (lclock t oldRepl)`. Kui uus kell on väiksem, siis uuendust ei rakendata, aga kui uus kell on suurem, siis rakendatakse uuendus. Võrdsusjuht on välistatud, sest sama duplikaadi uus kell on alati suurem kui tema eelmine. Seega ei teki konflikti ning omadus kehtib.

Lisanõude `Property4` tõestamiseks on vaja samaaegselt läbi vaadata nelja kella võrdlemisel tekkivad juhud. Antud tõestuse võib täispikkuses leida formaliseeritud teegist [4]. Siin on esitatud lisanõude `Property4` tõestuse algus, kus on näha, milliseid kellasid võrreldakse.

```
property4 : Property4
property4 {write-msg v₁' cl₁'} {write-msg v₂' cl₂'}
        {lwwvalue v cl at r} {lwwvalue v₁ cl₁ at r₁}
        {lwwvalue v₂ cl₂ at r₂}
        {s₁₂} {s₂₁} {mh₁} {mh₂} {p} {q} c eq₁ eq₂
  with <ˡ-comp cl₂' cl | <ˡ-comp cl₁' cl
    | <ˡ-comp cl₂' cl₁ | <ˡ-comp cl₁' cl₂
...
```

Lisanõue `Property4` nõuab, et konkurentsete sõnumite `write-msg v₁' cl₁'` ja `write-msg v₂' cl₂'` järjestikkuse rakendamise tulemus ei sõltu sõnumite rakendamise järjekorrast. Olek `lwwvalue v cl at r` on duplikaadi `r` olek enne sõnumite rakendamist, olek `lwwvalue v₁ cl₁ at r₁` on saadud rakendades sõnumit `write-msg v₁' cl₁'` olekule `lwwvalue v cl at r` ja olek `lwwvalue v₂ cl₂ at r₂` on saadud rakendades sõnumit `write-msg v₂' cl₂'` olekule `lwwvalue v cl at r`. Seega on lisanõude `Property4` tõestamiseks vaja läbi vaadata otsustusalgoritmi väljakutsete tulemused `<ˡ-comp cl₂' cl`, `<ˡ-comp cl₁' cl`, `<ˡ-comp cl₂' cl₂` ja `<ˡ-comp cl₁' cl₁`. Tõestus põhineb juhtumipõhisel analüüsil.

Üks viis, kuidas `Property4` juhte tõestada, on otsene tõestus.

```
property4 {write-msg v₁' cl₁'} {write-msg v₂' cl₂'}
        {lwwvalue v cl at r} {lwwvalue v₁ cl₁ at r₁}
        {lwwvalue v₂ cl₂ at r₂}
        _ refl refl
  | tri< s ¬s ¬s' | tri< t ¬t ¬t' | tri< u ¬u ¬u' | tri< w ¬w ¬w'
    = (lwwvalue v cl at r) , (lwwvalue v cl at r) ,
       refl , refl , refl , refl
```

Kehtivad võrratused `cl₂' <ˡ cl`, `cl₁' <ˡ cl`, `cl₂' <ˡ cl₂` ja `cl₁' <ˡ cl₁`, seega algset olekut `lwwvalue v cl at r` ei saa üle kirjutada, sõltumata sõnumite rakendamise järjekorrast.

Teine viis, kuidas omaduse `Property4` juhte tõestada, on kasutada vastuolu välja viimist.

```
property4 {write-msg v₁' cl₁'} {write-msg v₂' cl₂'}
        {lwwvalue v cl at r} {lwwvalue v₁ cl₁ at r₁}
        {lwwvalue v₂ cl₂ at r₂}
```

```
          _ refl refl
    | tri< s ¬s ¬s' | tri< t ¬t ¬t' | tri< u ¬u ¬u' | tri> ¬w ¬w' w
     = ⊥-elim (¬w t)
```

Kehtivad võrratused $cl_2' <^l cl$, $cl_1' <^l cl$, $cl_2' <^l cl_2$ ja $cl_1' >^l cl_1$. Juhtude läbivaatamisest `¬w : cl₁' <ˡ cl → ⊥` tuletab võrratuse $cl_1' <^l cl$ kehtimisest vastuolu ja `t : cl₁' <ˡ cl` väidab, et selline võrratus kehtib. Kui rakendada eituse tõestust `¬w` tõestusele `t`, siis on tulemuseks vastuolu. Järelikult saab lisanõude `Property4` juhtusid tõestada vastuolu välja viimisega.

Kolmas viis, kuidas omaduse `Property4` juhte tõestada, on konkurentsuse omaduse kasutamine vastuolu tuletamiseks.

```
property4 {write-msg v₁' cl₁'} {write-msg v₂' cl₂'}
        {lwwvalue v cl at r} {lwwvalue v₁ cl₁ at r₁}
        {lwwvalue v₂ cl₂ at r₂}
        {s₁₂} {s₂₁} {mh₁} {mh₂} {p} {q} c refl refl
  | tri> ¬s ¬s' s | tri> ¬t ¬t' t | tri≈ ¬u u ¬u' | tri≈ ¬w w ¬w'
   = ⊥-elim (haveDifferentAuthors (proj₁ c)
    (begin
     mreplica-of mh₂ ≡⟨sym (⋈ᵐ→replica-eq q) ⟩
     replica-of-clock cl₂' ≡⟨cong replica-of-clock u ⟩
     replica-of-clock cl₁' ≡⟨⋈ᵐ→replica-eq p ⟩
     mreplica-of mh₁
    □))
```

Kehtivad võrratused $cl_2' >^l cl$, $cl_1' >^l cl$, $cl_2' \equiv^l cl_2$ ja $cl_1' \equiv^l cl_1$. Predikaat `c` ütleb, et sõnumid `write-msg v₁' cl₁'` ja `write-msg v₂' cl₂'` on konkurentsed, mistõttu $mh_1 \notin mh_2$ ja $mh_2 \notin mh_1$. Võrdus `u : cl₂' ≡ cl₁'` väidab, et konkurentsete sõnumite Lamporti kellad on võrdsed, mis ei saa kehtida, kuna sellisel juhul pärineksid mõlemad sõnumid samast duplikaadist ja ei oleks konkurentsed. Abilemma `haveDifferentAuthors` võtab sisendiks predikaadi, mis väidab, et sõnumi ajalugu ei sisaldu oleku ajaloos, ja tõestab, et sõnumi ajalugu ja oleku ajalugu pärinevad erinevatest duplikaatidest. Abilemma `haveDifferentAuthors` tõestus on esitatud lisas 6.4. Abilemma `⋈ᵐ→replica-eq` võtab sisendiks predikaadi, mis seob sõnumi tema ajalooga, ja tagastab tõestuse, et sellisel juhul on sõnumis sisalduv Lamporti kell võrdne sõnumi ajaloos sisalduva Lamporti kellaga. Abilemma `⋈ᵐ→replica-eq` tõestus on esitatud lisas 6.5. Kasutades abilemmat `⋈ᵐ→replica-eq` saab koostada võrrandite jada, mis tõestab, et `mreplica-of mh₂ ≡ mreplica-o mh₁`, sest

```
crdtOR : CRDT
crdtOR = record {
  Data      = ORSet;
  _≡ᵈ_      = _≡ᵒ_;

  isEquivalence-≡ᵈ = record {
    refl = λ {x}
      → refl , id-fun , id-fun ;
    sym = λ { (p , i , j)
      → sym p , j , i } ;
    trans = λ (p₁ , i₁ , j₁) (p₂ , i₂ , j₂)
      → trans p₁ p₂ , i₂ ∘ i₁ , j₁ ∘ j₂ };

  Operation = OROperation;
  Message   = ORMessage;
  initial   = initialOR;
  prepare   = prepareOR;
  _•_       = applyOR;
  Query     = S;
  View      = Bool;
  eval      = evalOR;
  property1 = refl;
  property2 = property2 }
```

Joonis 12. OR-hulk CRDT

sõnumite ajalugudes mh_1 ja mh_2 olevad Lamporti kellad on samaväärsed. Kasutades abilemmat `haveDifferentAuthors` predikaadil $mh_2 \notin mh_1$, mille tulemus on `mreplica-of` $mh_2 \not\equiv r_1$, ja eelnevalt tõestatud võrduset `mreplica-of` $mh_2 \equiv$ `mreplica-o` mh_1, saab tuletada vastuolu. Järelikult saab seda lisanõude `Property4` juhtu tõestada kasutades ära sõnumite konkurentsust.

5.3 Hulgad

CRDT hulgad (ingl *CRDT Sets*) on andmetüübid, mis toetavad hajussüsteemides elementide lisamist ja eemaldamist. [10]. Tuntud näide on lisamist ja eemaldamist toetav hulk (ingl *Observe-Remove Set*) (lüh OR-hulk), mis võimaldab samaaegselt mitmes duplikaadis lisatud elemente hiljem kindlalt eemaldada [10]. Selle saavutamiseks seob OR-hulk iga lisamise kordumatu identifikaatoriga ning salvestab need eraldi [10]. Elementide eemaldamise korral eemaldatakse kõik sama väärtusega, aga erinevate identifikaatoritega lisamiskirjed, mis on eemaldamise hetkeks teada [10]. OR-hulk sobib rakendustesse, kus elementide lisamine ja eemaldamine peab olema turvaline ka võrgu viivituste või katkestuste korral.

Joonis 12 kujutab Agdas formaliseeritud OR-hulk CRDT-d. Duplikaatides hoitakse `ORSet` tüüpi andmeid.

```
record ORSet : Set where
  constructor orset
  field
    elements : List ORValue
```

```
        counter : ℕ
```

`ORSet` on andmestruktuur, mis koosneb väärtuste hulgast ja loendurist. Väärtuste hulk salvestab hulga elemente koos nende unikaalsete identifikaatoritega ning loendurit kasutatakse kordumatute identifikaatorite genereerimiseks iga uue lisamise korral. Agda kirjeldus defineerib OR-hulga andmiku `record` konstruktoriga `orset`, millel on kaks välja: `elements`, mis on OR-väärtuste loend, ja `counter`, mis on naturaalarvuline loendur. `ORSet` jaoks on defineeritud eraldi elementide võrdlusrelatsioon `s ≡ᵒ s'`.

```
_≡ᵒ_ : ORSet ↦ ORSet ↦ Set
orset elements₁ ₁counter₁ ≡ᵒ orset elements₂ counter₂ =
  counter₁ ≡ counter₂ ×
  (∀ {x} ↦ (x ∈ᵒᵛelements₁) ↦ (x ∈ᵒᵛelements₂)) ×
  (∀ {x} ↦ (x ∈ᵒᵛelements₂) ↦ (x ∈ᵒᵛelements₁))
```

Võrdlusrelatsioon `s ≡ᵒ s'` abil saab loendeid(Agdas `List`) kohelda kui hulkasid, sest `s ≡ᵒ s'` tähendab mõlemasuunalise elementide kuuluvuse nõudmist.

`ORValue` on andmiku tüüp, mis esindab üht elementi OR-hulgas.

```
record ORValue : Set where
  constructor orvalue
  field
    value : S
    id    : LClock
```

Iga `ORValue` koosneb väärtusest `value`, mis kuulub mingisse tüüpi `S`, ja unikaalsest identifikaatorist `id`, mis on Lamporti kell `LClock`. Sellise struktuuri loomiseks kasutatakse konstruktorit `orvalue`. Lisaks on defineeritud kaks abifunktsiooni. `value-of-orvalue`, mis tagastab `ORValue` sees oleva tegeliku väärtuse, ning `id-of-orvalue`, mis tagastab selle unikaalse identifikaatori. Joonisel 12 kujutatud OR-hulga CRDT elementide samasuse määramiseks on defineeritud eraldi võrdlusrelatsioon `v ≡ᵒᵛ v'`.

```
_≡ᵒᵛ_ : ORValue → ORValue → Set
(orvalue value id) ≡ᵒᵛ (orvalue value' id') =
  value ≡ value' × id ≡ id'
```

Kaks väärtust on võrdsed siis, kui nende sisu `value` ja identifikaator `id` on võrdsed. Funktsioon `dec-≡ᵒᵛ` otsustab, kas kaks `ORValue` tüüpi elementi on võrdsed, kasutades väärtuse tüübi

```
wb-crdtOR : WBCRDT
wb-crdtOR = record {
  D         = crdtOR;
  property3 = property3;
  property4 = property4 }
```

Joonis 13. Hästi käituv OR-hulk CRDT

otsustatavat võrdsust `deqS` ja Lamporti kellade võrduse kontrollimist `<ˡ-comp`. Funktsiooni `dec-≡ ᵒᵛ` definitsioon on leitav lisas 6.3.

OR-hulk CRDT-l on kaks võimalikku operatsiooni, `add` ja `remove`, mis lisavad või eemaldavad väärtusi andmestruktuurist. Lisaks on OR-hulk CRDT-l kahte tüüpi sõnumid, `write-add` ja `write-remove`, kus esimene lisab loendisse `ORSet` väärtuse ja viimane eemaldab väärtuse loendist. Funktsioon `initialOR` algatab duplikaadis `r ORSet` tüüpi oleku, määrates sellele tühja elementide loendi ja loenduri väärtuseks nulli. Funktsioon `findS` otsib antud väärtuse `x` olemasolu `ORSet` elementide loendis, samas kui `evalOR` hindab, kas väärtus `x` on `ORSet` olekus olemas. Funktsioon `removeORValue` eemaldab `ORSet` elementide loendist väärtuse `x`, samas kui `removeAll` eemaldab kõik määratud väärtused `ORSet` elementide loendist. `findAllS` leiab kõik `S` tüüpi väärtused elementide loendist. `prepareOR` valmistab ette sõnumi, mis kas lisab või eemaldab väärtuse, ja `applyOR` rakendab selle sõnumi olekule. `crdtOR` lisaomaduste `Property1` ja `Property2` tõestused taanduvad refleksiivsusele. Lisaomaduse `Property1` tõestus on näha joonisel 12. Järgnevalt on välja toodud `Property2` tõestuse.

```
property2 : {m : ORMessage} {s s' : RState ORSet}
        → applyOR m s ≡ just s'
        → replica-of s ≡ replica-of s'
property2 {write-add x} {orset elements counter at r} {s'} refl =
   refl
property2 {write-remove xs} {orset elements counter at r} {s'} refl =
   refl
```

Joonis 13 kujutab Agdas formaliseeritud hästi käituvat OR-hulk CRDT-d. Lisanõude `Property3` tõestamiseks tuleb varasemalt kirjeldatud abifunktsioone kasutades duplikaadi olekule operatsioone `add` ja `remove` rakendada.

```
property3 : Property3
property3 {add val} {orset elements counter at r}
 = (orset (orvalue val (lclock counter r) :: elements)
   (1 + counter) at r) ,
```

```
    (refl , id-fun , id-fun) ,
    refl
property3 {remove vals} {orset elements counter at r}
 = (orset (removeAll (findAllS vals elements) elements)
    (1 + counter) at r) ,
    (refl , id-fun , id-fun) ,
    refl
```

Kui lisatakse väärtus `val`, siis uus olek sisaldab väärtust koos värskendatud loenduri väärtusega ja originaalsed elemendid jäävad alles. Kui eemaldatakse väärtused `vals`, siis eemaldatakse need väärtused olemasolevate elementide loendist ning loendurit uuendatakse.

Lisanõude `Property4` tõestamiseks on vaja läbi vaadata, milliseid konkurentseid sõnumeid rakendatakse. Tõestuses kasutatakse mitut abilemmat, mille tõestused on avaldatud Zenodos [4].

```
property4 {write-add val₁} {write-add val₂}
        {orset elements counter at r}
        {orset elements₁ counter₁ at r₁}
        {orset elements₂ counter₂ at r₂}
        {s₁₂} {s₂₁} {mh₁} {mh₂} {p} {q} c refl refl =
  (orset (val₂ :: val₁ :: elements) (2 + counter) at r) ,
  (orset (val₁ :: val₂ :: elements) (2 + counter) at r) ,
  refl , refl ,
  (refl ,
   (λ {ov} t → writeAddInvariant t) ,
   λ {ov} t → writeAddInvariant t) ,
  refl
```

Kui kaks elemendi lisamise sõnumit `write-add val₁` ja `write-add val₂` on konkurentsed, siis on vaja tõestada, et väärtus `ov` kuulub tulemushulka, sõltumata esimese kahe elemendi järjekorrast. Seda tõestab abilemma `writeAddInvariant`. Kuna hulgad `orset (val₂ :: val₁ :: elements) (2 + counter) at r` ja `orset (val₁ :: val₂ :: elements) (2 + counter) at r` sisaldavad samu väärtuseid, seega saab selle `Property4` juhu tõestada abilemma `writeAddInvariant` abil.

```
property4 {write-add val₁} {write-remove val₂}
        {orset elements counter at r}
        {orset elements₁ counter₁ at r₁}
        {orset elements₂ counter₂ at r₂}
        {s₁₂} {s₂₁} {mh₁} {mh₂} {p} {q} c refl refl =
```

```
      (orset (removeAll val₂ (val₁ :: elements)) (2 + counter) at r) ,
      (orset (val₁ :: removeAll val₂ elements) (2 + counter) at r) ,
      refl , refl ,
      (refl ,
       (λ {ov} t → removeAllOne val₁ val₂ t) ,
       λ {ov} t → removeAllDistinct val₁ val₂
          (concurrentDistinct p q (proj₂ c) (proj₁ c)) t) ,
      refl
```

Kui elemendi lisamise ja elementide eemaldamise sõnumid `write-add` $\texttt{val}_1$ ja `write-remove` $\texttt{val}_2$ on konkurentsed, siis ei viita need samale elemendile, sest nad ei pärine samast duplikaadist ning seetõttu ei ole elementide identifikaatorid võrdsed. Seda tõestab abilemma `concurrentDistinct`. Abilemma `removeAllOne` tõestab, et kui väärtus `ov` kuulub pärast teatud väärtuste eemaldamist, siis kuulus see väärtus sinna ka enne eemaldamist. Abilemma `removeAllDistinct` tagab, et kui väärtus `val` on eristatav eemaldatavatest väärtustest `vals` ning kuulub hulka `val' :: removeAll vals elements`, siis peab väärtus `vals` kuuluma ka hulka `removeAll vals (val':: elements)`. Olenemata mis järjekorras sõnumeid rakendatakse, saab abilemmadele tuginedes tõestada, et tulemus jääb samaks ja ei teki konflikti. `Property4` tõestus on analoogne ka juhul, kui väärtuste eemaldamise sõnum `write-remove` $\texttt{val}_1$ ja lisamissõnum `write-add` $\texttt{val}_1$ on konkurentsed.

```
    property4 {write-remove val₁} {write-remove val₂}
           {orset elements counter at r}
           {orset elements₁ counter₁ at r₁}
           {orset elements₂ counter₂ at r₂}
           {s₁₂} {s₂₁} {mh₁} {mh₂} {p} {q} c refl refl =
      (orset (removeAll val₂ (removeAll val₁ elements)) (2 + counter) at r) ,
      (orset (removeAll val₁ (removeAll val₂ elements)) (2 + counter) at r) ,
      refl , refl ,
      (refl ,
       (λ {ov} t → removeAllTwice val₁ ₂val₂ {val = ov} t) ,
       λ {ov} t → removeAllTwice val₂ val₁ {val = ov} t) ,
      refl
```

Kui kaks elementide eemaldamise sõnumit `write-remove` $\texttt{val}_1$ `write-remove` $\texttt{val}_2$ on konkurentsed, siis nende rakendamise tulemus ei sõltu järjekorrast. Abilemma `removeAllTwice` tõestab, et need eemaldamised on kommutatiivsed ja konflikte ei teki. Seega saab tõestada `Property4` viimase juhu.

6. Kokkuvõte

Lõputöö eesmärk oli Agdas formaliseerida konfliktivabade dubleeritud andmetüüpide ehk CRDT-de teek [4]. Töö keskseks saavutuseks oli CRDT-de struktuuri formaliseerimine. Lisaks tutvustati kolme levinud CRDT-de disainitüüpi, mis formaliseeriti tuginedes CRDT-de struktuurile. Lõputöös tutvustati ka hajussüsteemide ja CRDT-dega seotud olulisi mõisteid nagu andmete dubleerimine ja lõppkokkuvõttes tugev kooskõlalisus. Veel kirjeldati programmeerimiskeelt ja tõestusassistenti Agda.

Edasised uurimissuunad võiksid keskenduda konfliktivabade dubleeritud andmetüüpide omaduste täpsemale analüüsile ja tõestamisele. Üheks oluliseks järgmiseks sammuks oleks formaalselt tõestada lõppkokkuvõttes kooskõlalisuse saavutamine kõigis duplikaatides. See omadus on keskse tähtsusega hajussüsteemide usaldusväärsuse tagamisel. Lisaks võiks tööd laiendada ka keerukamate struktuuride formaliseerimisele, näiteks pool-otsekorrutisel põhinevate CRDT-de modelleerimisele ja nende korrektsuse tõestamisele [2]. Selline lähenemine võimaldaks erinevate CRDT-de kombineerimist uuteks CRDT-deks viisil, mis säilitab süsteemi determinismi ja kooskõla ilma keskse koordineerimiseta.

Viited

[1] Steen M. van ja Tanenbaum A. S. Distributed Systems. 4. väljaanne. distributed-systems.net, 2023. https://www.distributed-systems.net.

[2] Weidner M., Miller H. ja Meiklejohn C. Composing and decomposing op-based CRDTs with semidirect products. *Proc. ACM Program. Lang.* 4.ICFP (august 2020). DOI: 10.1145/3408976.

[3] Shapiro M., Preguiça N., Baquero C., and Zawirski M. Conflict-free Replicated Data Types. SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems. Vol. 6976. Springer, Oct. 10, 2011, p. 386. DOI: 10.1007/978-3-642-24550-3_29.

[4] Ibrus M. ja Ahman D. Formalising CRDTs in Agda. Mai 2025. DOI: 10.5281/zenodo.15425132. doi.org/10.5281/zenodo.15425132.

[5] Santos A. P. L. d. Protocols for Database Replication with Delta-based CRDT. magistritöö. Detsember 2023. https://run.unl.pt/handle/10362/167320 (11.05.2025).

[6] Blau T. Verifying Strong Eventual Consistency in δ-CRDTs. Bakalaureusetöö. Seattle, WA, 2020. https://arxiv.org/abs/2006.09823.

[7] Bove A., Dybjer P., and Norell U. A Brief Overview of Agda – A Functional Language with Dependent Types. *Theorem Proving in Higher Order Logics*. Ed. by Berghofer S., Nipkow T., Urban C., and Wenzel M. Berlin, Heidelberg: Springer, 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6.

[8] Norell U. ja Chapman J. Dependently Typed Programming in Agda. https://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf. (06.05.2025).

[9] Wadler P., Kokke W. ja Siek J. G. Programming Language Foundations in Agda. August 2022. https://plfa.inf.ed.ac.uk/22.08/.

[10] Shapiro M., Preguiça N., Baquero C., and Zawirski M. A comprehensive study of Convergent and Commutative Replicated Data Types. report. Inria – Centre Paris-Rocquencourt ; INRIA, Jan. 13, 2011. https://inria.hal.science/inria-00555588 (05/04/2025).

[11] Dolan S. Brief Announcement: The Only Undoable CRDTs are Counters. Virtual Event, Italy, 2020. DOI: 10.1145/3382734.3405749. https://doi.org/10.1145/3382734.3405749.

[12] Massachusetts Computer Associates, Inc. ja Lamport L. Time, clocks, and the ordering of events in a distributed system. VMware Research and Calibra. *Concurrency: the Works*

of Leslie Lamport. Toim. Malkhi D. Association for Computing Machinery, 9. oktoober 2019. DOI: 10.1145/3335772.3335934.

Lisad

6.1 Monotoonse loendur CRDT omaduse `Property4` tõestus

```
property4 : Property4
property4 {add-msg n₁} {add-msg n₂}
        {s at r} {s₂ at r₂} {s₁ at r₁}
        c refl refl =
          (((s + n₁ + n₂) at r)) ,
          ((s + n₂ + n₁) at r) ,
          refl ,
          refl ,
          (begin
            (s + n₁) + n₂ ≡⟨ +-assoc s n₁ n₂ ⟩
            s + (n₁ + n₂) ≡⟨ cong₂ _+_ {s} refl (+-comm n₁ n₂) ⟩
            s + (n₂ + n₁) ≡⟨ sym (+-assoc s n₂ n₁) ⟩
            (s + n₂) + n₁□
          ) ,
          refl
```

6.2 Omaduse `<ˡ-comp` tõestus

```
<ˡ-comp : Trichotomous _≡_ _<ˡ_
<ˡ-comp (lclock t₁ r₁) (lclock t₂ r₂)
  with <-cmp t₁ t₂
<ˡ-comp (lclock t₁ r₁) (lclock t₂ r₂) | tri< p ¬p ¬p' =
  tri< (time< p)
       (λ { refl → ¬p refl })
       λ { (time< s) → ¬p' s ; (replica< s) → ¬p refl }
<ˡ-comp (lclock t₁ r₁) (lclock t₂ r₂) | tri≈ ¬p refl ¬p'
  with <ʳ-cmp r₁ r₂
... | tri< q ¬q ¬q'
  = tri< (replica< q)
         (λ { refl → ¬q refl })
         λ { (time< s) → ¬p' s ; (replica< s) → ¬q' s }
... | tri≈ ¬q refl ¬q'
  = tri≈ (λ { (time< s) → ¬p' s ; (replica< s) → ¬q s })
         refl
         λ { (time< s) → ¬p' s ; (replica< s) → ¬q s }
... | tri> ¬q ¬q' q =
  tri> (λ { (time< s) → ¬p' s ; (replica< s) → ¬q s })
       (λ { refl → ¬q q })
       (replica< q)
<ˡ-comp (lclock t₁ r₁) (lclock t₂ r₂) | tri> ¬p ¬p' p =
  tri> (λ { (time< s) → ¬p s ; (replica< s) → ¬p p })
       (λ { refl → ¬p p })
       (time< p)
```

6.3 Funktsiooni dec-≡ᵒᵛ definitsioon

```
dec-≡ᵒᵛ : Decidable _≡ᵒᵛ_
dec-≡ᵒᵛ (orvalue value id) (orvalue value' id')
   with deqS value value' | <ˡ-comp id id'
... | yes p | tri< q ¬q' ¬q'' =
   no (λ { (refl , refl) → ¬q' refl })
... | yes refl | tri≈ ¬q refl ¬q'' =
   yes (refl , refl)
... | yes p | tri> ¬q ¬q' q'' =
   no (λ { (refl , refl) → ¬q' refl })
... | no ¬p | tri< q ¬q' ¬q'' =
   no (λ { (refl , refl) → ¬p refl })
... | no ¬p | tri≈ ¬q q' ¬q'' =
   no (λ { (refl , refl) → ¬p refl })
... | no ¬p | tri> ¬q ¬q' q'' =
   no (λ { (refl , refl) → ¬p refl })
```

6.4 Abilemma `haveDifferentAuthors` tõestus

```
haveDifferentAuthors : {r' : Replica}
                     → {mh : MessageHistory}
                     → {sh' : StateHistory r'}
                     → mh ∉ sh'
                     → mreplica-of mh ≢ r'
haveDifferentAuthors {r'} {r , op , sh} {sh'} (∉-hEmp x) eq = x eq
haveDifferentAuthors {r'} {mh} {sh'} (∉-hEff x x₁ eq) eq' =
  haveDifferentAuthors eq eq'
```

6.5 Abilemma ⋈ᵐ→**replica-eq** tõestus

```
⋈ᵐ→replica-eq : ∀ {v cl mh} → write-msg v cl ⋈ᵐ mh
              → replica-of-clock cl ≡ mreplica-of mh
⋈ᵐ→replica-eq {v} {lclock t r} {mh}
             (⋈ᵐ-prepare {o = write val}
              {s = lwwvalue _ (lclock t' oldRepl) at r'}
              refl q)
```

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Marlene Ibrus,

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose ”Konfliktivabade dubleeritud andmetüüpide formaliseerimine”, mille juhendaja on Danel Ahman, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada Tartu Ülikooli digitaalarhiivi kuni autoriõiguse kehtivuse lõppemiseni;
2. annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi kaudu Creative Commonsi litsentsiga CC BY NC ND 4.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni;
3. olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile;
4. kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Marlene Ibrus

15.05.2025