

TAMPEREEN TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

MIKA KATARA

**SPESIFIKAATIOIDEN STRUKTUROINNISTA
TLA⁺:SSA JA DISCOSSA**

Diplomityö

Aihe hyväksytty osastoneuvoston kokouksessa
17.4.1996
Tarkastaja: Prof. Reino Kurki-Suonio

Alkulause

Tämä työ on tehty Tampereen teknillisen korkeakoulun Ohjelmistotekniikan laitoksessa. Työn ohjaajana ja tarkastajana on toiminut professori Reino Kurki-Suonio, jota haluan kiittää sekä ystävällisestä ohjauksesta että mielenkiinnosta työtäni kohtaan. Kiitokset myös toisena tarkastajana toimineelle apulaisprofessori Hannu-Matti Järviselle.

Kiitän myös muita DisCo-projektiryhmän jäseniä ja Ohjelmistotekniikan laitoksen henkilökuntaa kannustavan työilmapiirin luomisesta. Lämpimät kiitokset erityisesti tekn. yo. Risto Pitkäselle, jonka kanssa olen työni edetessä saanut käydä monta valaisevaa keskustelua ja jolta olen saanut paljon rakentavaa kritiikkiä. Olen saanut vastauksia työssäni ilmenneisiin kysymyksiin myös monilta muilta henkilöiltä. Erityisen kiitoksen ansaitsevat ainakin yliassistentti Pertti Kellomäki ja tekn. yo. Timo Aaltonen.

Lämmin kiitos Leslie Lamportille siitä, että hän on kehittänyt TLA:n ohella myös \LaTeX in, jonka avulla tämäkin dokumentti on valmistettu. Työtäni on helpottanut suuresti se, että hän tarjoaa kirjoittamansa TLA- ja TLA⁺-dokumentit myös \LaTeX -muodossa.

Tämä diplomityö olisi paljon ikävämpi lukea, ellei fil. yo. Essi Lammi olisi tarkastanut kieliasua. Kiitokset kuuluvat siis myös hänelle.

Tampereella 4.9.1996

Mika Katara
Opiskelijankatu 4 B 80
FIN-33720 TAMPERE
Puh. 317 1057
E-mail: clark@iki.fi

Tiivistelmä

TAMPEREEN TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto
Ohjelmistotekniikka
KATARA, MIKA: Spesifikaatioiden strukturoinnista
TLA⁺:ssa ja DisCossa
Diplomityö, 53 s.
Tarkastajat: Prof. Reino Kurki-Suonio ja
apul.prof. Hannu-Matti Järvinen
Rahoittaja: DisCo-projekti
Syyskuu 1996
Avainsanat: reactive systems, formal specification,
structuring of specifications

DisCo (Distributed Cooperation) on Tampereen teknillisessä korkeakoulussa kehitetty formaali menetelmä, joka on tarkoitettu reaktiivisten järjestelmien spesifointiin. Suoritettavan DisCo-kielen semantiikka on määritelty TLA:lla, joka on Leslie Lamportin kehittämä logiikka rinnakkaisten ja reaktiivisten järjestelmien spesifointiin. TLA⁺ on TLA:han perustuva spesifointikieli, johon on otettu mukaan ohjelmointikielistä tuttuja rakenteita ja nimeämiskäytäntöjä.

Sekä DisCo että TLA⁺ sopivat siis reaktiivisten järjestelmien spesifointiin ja niillä on yhteinen semanttinen perusta. Menetelmät eroavat kuitenkin toisistaan monissa suhteissa, esimerkiksi syntaksien osalta. DisCo-kielen syntaksi muistuttaa ohjelmointikieltä, mutta TLA⁺:n syntaksi on yhdistelmä matemaattisia merkintöjä ja ohjelmointikielimäisiä rakenteita. Tästä syystä vertailu näiden kahden kielen välillä on paikallaan.

Tässä diplomityössä tarkastellaan DisCon ja TLA⁺:n eroja, mitä tulee niiden tarjoamiin keinoihin spesifikaation strukturoinnissa. Esimerkkeinä käytetään yksinkertaisia spesifikaatioita, jotka esitetään molemmilla kielillä. Lisäksi esitellään näiden kahden formaalin menetelmän tarjoamia mahdollisuuksia määrittelijän näkökulmasta katsottuna.

Tämän työn tarkoituksena oli myös etsiä TLA⁺:sta ominaisuuksia, joita voitaisiin lisätä DisCon tuleviin versioihin. Sellaisia ei kuitenkaan löydetty. TLA⁺ on selvästi matemaatikon työkalu ja siinä otettu vapauksia, jotka helpottavat spesifointia korkealla abstraktiotasolla, mutta joiden kuvaaminen toteutukseksi on erittäin hankalaa. Myös DisCo-kielessä on mahdollisuus käyttää lausekkeita, joita ei pystytä toteuttamaan, mutta niitä on oleellisesti vähemmän kuin TLA⁺:ssa. Niiden lisäämistä on kuitenkin vaikea perustella.

Abstract in English

TAMPERE UNIVERSITY OF TECHNOLOGY
Department of Information Technology
Software Engineering
Katara, Mika: On Structuring of Specifications
in TLA+ and DisCo
Master of Science Thesis, 53 pages
September, 1996

DisCo (Distributed Cooperation) is a formal specification method for reactive systems developed at the Tampere University of Technology. The semantics of the executable DisCo language is defined in terms of Temporal Logic of Actions. TLA introduced by Leslie Lamport, is a logic for specifying and reasoning about concurrent and reactive systems. TLA⁺ is a specification language based on TLA. TLA⁺ contains structuring and naming conventions familiar from programming languages.

Both DisCo and TLA⁺ suit well for specification of reactive systems and share the common semantics. They differ in many ways though, for example the syntax of the DisCo language is like the one of a programming language and the syntax of TLA⁺ is more like a combination of mathematical notations and structures of programming languages. Therefore we should compare these two languages.

The topic of this master's thesis is to compare the differences between DisCo and TLA⁺ in structuring of specifications. The same simple examples are presented in both languages and also the benefits of both methods are introduced from the specifier's point of view.

Another topic was to find features of TLA⁺ which could be included in the future versions of DisCo. However, such features were not found. TLA⁺ is a tool designed for a mathematician and it allows some liberties, which help to specify on a high level of abstraction, but which are very difficult to implement. Also in the DisCo language, there are statements which could not be implemented, but the number of these is considerably lower. There should be no reason to add statements of this kind.

Sisällysluettelo

Alkulause	i
Tiivistelmä	ii
Abstract in English	iii
Sisällysluettelo	iv
Määritelmät ja lyhenteet	vi
1 Johdanto	1
1.1 Reaktiiviset järjestelmät	1
1.2 Formaali spesifointi	1
1.3 Työn tarkoitus	1
1.4 Yleiskatsaus työhön	2
2 Teoriaa ja käsitteitä	3
2.1 Virheitä on vaikea välttää	3
2.2 Formaalien menetelmien perusteet	4
2.3 Formaalien menetelmien käyttö	5
2.4 Spesifoinnin tarkoitus	5
2.5 Spesifikaation laatiminen	6
2.6 Strukturointi	7
3 TLA	9
3.1 Logiikka rinnakkaisuuden hallintaan	9
3.2 Aktioiden logiikka	9
3.3 Aikalogiikka	10
3.4 Turvallisuus- ja elävyysominaisuudet	11
3.5 TLA-spesifikaatio	12
3.6 Spesifointi TLA:lla	13
4 TLA⁺	15
4.1 Uuden spesifointikielen tarkoitus	15
4.2 Kielen käsitteet ja rakenteet	15
4.3 Operaattorit	17
4.4 Strukturoinnista	17
4.5 Moduulit	19
4.6 The Game of Musical Chairs	21
4.7 TLA ⁺ spesifointimenetelmänä	25
5 DisCo	26
5.1 Menetelmä	26
5.2 Yhteisaktioteoria	26
5.3 Objektit	27
5.4 Spesifikaation rakenne	28

5.5	Superpositio	29
5.6	Strukturoinnista	30
5.7	Musical Chairs DisColla	31
5.8	Peräkkäinen suoritusmalli	34
5.9	Työkalu	35
5.10	Todistamisesta	36
6	DisCo-spesifikaation muuntaminen TLA⁺:ksi	38
6.1	Muunnos DisCosta TLA ⁺ :ksi	38
6.2	Hajautettu lajittelu DisColla	38
6.3	Hajautettu lajittelu TLA ⁺ :lla	40
6.4	Muunnoksen yleistäminen	42
7	TLA⁺ ja DisCo spesifointimenetelminä	45
7.1	Menetelmien erot	45
7.2	Kielten erot	46
7.3	Uusi DisCo-kieli	47
7.4	Menetelmien kehittämisestä	48
8	Yhteenveto	50
8.1	Vertailun mielekkyys	50
8.2	Työn arviointi	51
	Lähdeluettelo	52

Määritelmät ja lyhenteet

\square eli box eli always on TLA:n operaattori. Jos operandina on tilapredikaatti, on predikaatti voimassa jokaisessa käyttäytymisen tilassa. Jos taas operandina on aktio, esimerkiksi \mathcal{A} , on jokainen käyttäytymisen askel \mathcal{A} -askel.

\diamond eli diamond eli eventually on TLA:n operaattori. Tarkoittaa samaa kuin $\neg\square\neg$. Jos operandina on tilapredikaatti, on predikaatti voimassa jossakin käyttäytymisen tilassa. Merkintä $\diamond\langle\mathcal{A}\rangle$ tarkoittaa, että jokin käyttäytymisen askel on $\langle\mathcal{A}\rangle$ -askel.

\triangleq -symboli luetaan englanniksi “is defined as” tai “equals by definition”. Suomeksi sen voi lukea “määritellään” tai “on määritelmän mukaan”.

\equiv -symboli merkitsee, että sen molemmilla puolilla olevilla logiikan lausekkeilla on sama totuusarvo. Määritelmän mukaan $A \equiv B$ tarkoittaa samaa kuin $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Samoin määritellään myös looginen ekvivalenssi \iff , joka luetaan “jos ja vain jos”.

Aktio on TLA:ssa ja DisCossa atominen tapahtuma, jossa järjestelmän tila muuttuu.

Alhaalta ylös (bottom-up) -spesifointitapa tarkoittaa sitä, että järjestelmän spesifointi aloitetaan pienistä itsenäisesti toimivista osista, jotka yhdessä muodostavat toimivan kokonaisuuden. [Setä92]

Assertio on totuusarvoinen lauseke, jonka avulla DisCo-spesifikaatiossa esitetään jokin turvallisuusvaatimus. Jos lauseke saa arvon epätosi, on vaatimusta rikottu.

Atomisen aktio on sellainen, joka kerran alkuun päästyään tapahtuu loppuun asti siten, että mikään muu aktio ei voi häiritä sen tapahtumista.

Disjunktio on totuusarvoinen lauseke, joka on muodostettu propositioista yhdistämällä ne loogisella operaattorilla “tai” eli **OR** eli \vee .

Elävyysominaisuudet kertovat sen, että “jotain hyvää tapahtuu joskus”. Jos systeemillä on esimerkiksi elävyysominaisuus $\diamond(x = 10)$, tarkoittaa se sitä, että muuttujan x arvo on joskus kymmenen.

Heikko reiluus (weak fearness) takaa sen, että jos tapahtuma on jostakin ajanhetkestä eteenpäin jatkuvasti mahdollinen, se tapahtuu äärettömän usein.

Importointi tarkoittaa DisCossa jonkin aiemmin määritellyn systeemin tuomista osaksi toista systeemiä. [Aal96]

Invariantti eli pysyväisväittäjä on aina tosi. Esimerkiksi $x = 10$ on invariantti, jos muuttujan x arvo on aina kymmenen.

Konjunktio on totuusarvoinen lauseke, joka on muodostettu propositioista yhdistämällä ne loogisella operaattorilla “ja” eli **AND** eli \wedge .

Kvantifiointi on matematiikan ja logiikan operaatio, joka voidaan tehdä joko kaikkikvanttorin tai olemassaolokvanttorin avulla. Kaikkikvanttori \forall tarkoittaa “kaikilla arvoilla” ja olemassaolokvanttori \exists “on olemassa jokin arvo”. Esimerkiksi lausekkeiden $\forall x \in \text{Nat} : x < x + 1$ ja $\exists y \in \text{Nat} : y^2 = 4$ arvot ovat tosia kun joukko Nat , jonka yli kvantifoidaan, on luonnollisten lukujen joukko.

Käyttäytyminen tarkoittaa TLA:ssa ääretöntä tilaketjua.

Luokka kuvaa siihen kuuluvien olioiden ominaisuudet.

Olio on luokan ilmentymä. DisCossa olioita kutsutaan objekteiksi.

Pakotettu reiluus (impartiality) takaa sen, että kaikki tapahtumat tapahtuvat äärettömän usein. Tämä voi johtaa ristiriitaan turvallisuusvaatimusten kanssa.

Perintä tarkoittaa sitä, että perivä luokka saa kaikki perittävän luokan ominaisuudet.

Propositio on logiikan lauseke, jonka arvo on tosi tai epätosi.

Superpositio on DisCon tapa toteuttaa vaiheittainen tarkentaminen. Superpositio säilyttää turvallisuusominaisuudet.

Tila tarkoittaa DisCossa sekä objektin paikallista tilaa, joka koostuu tilakoneista ja muuttujista, että systeemin tilaa, joka koostuu objektien tiloista.

Turvallisuusominaisuudet kertovat sen, että “jotain pahaa ei koskaan tapahdu”. Jos systeemillä olisi turvallisuusominaisuus $\square(x \leq 10)$, tarkoittaisi se sitä, että muuttujan x arvo ei ikinä ylitä arvoa kymmenen. Tämä ominaisuus rikkoutuisi jos jossakin tilassa x :n arvo olisi yli kymmenen, eli “jotain pahaa tapahtuisi”.

Vahti on DisCon aktiossa totuusarvoinen lauseke, jonka ollessa tosi aktio voi tapahtua.

Vahva reiluus (strong fairness) takaa sen, että jos tapahtuma on mahdollinen äärettömän usein, se myös tapahtuu äärettömän usein.

Vaiheittainen tarkentaminen (stepwise refinement) tarkoittaa sitä, että spesifikaatiota tarkennetaan pienin askelin ylhäältä alas kohti vaadittua tarkkuustasoa. Spesifikaatioon tuodaan kerralla vain vähän uusia ominaisuuksia siten, että vanhat ominaisuudet säilyvät. [Jär92]

Ylhäältä alas (top-down) -spesifiointitapa tarkoittaa sitä, että spesifiointi aloitetaan mahdollisimman yleisestä spesifikaatiosta, joka kuvaa systeemin hyvin korkealla abstraktiotasolla. Tästä edetään tarkentamalla aina toteutukseen asti. [Setä92]

1 Johdanto

1.1 Reaktiiviset järjestelmät

Perinteisesti tietotekniset järjestelmät ovat automatisoineet jonkin tehtävän, jossa voidaan erottaa toisistaan syötteet ja tulosteet. Järjestelmä on ottanut syötteet sisäänsä, prosessoinut ne ja tuottanut tulosteet. Monet nykypäivän järjestelmät eivät kuitenkaan sovi tähän kategoriaan, sillä ne toimivat periaatteessa taukoamatta, reagoiden jatkuvasti ulkoapäin tuleviin ärsykkeisiin. Tällaista järjestelmää sanotaan reaktiiviseksi. Reaktiiviset järjestelmät ovat usein sulautettuja, eli niissä on elektroniikan ohella myös ohjelmisto, joka on oleellinen osa järjestelmää. Esimerkki sulautetusta reaktiivisesta järjestelmästä on matkapuhelin, joka ottaa vastaan käyttäjän antamat komennot ja pyrkii säilyttämään yhteyden tukiasemaan.

Reaktiivinen järjestelmä voi olla myös hajautettu, jolloin suoritus tapahtuu erillisissä, keskenään kommunikoivissa yksiköissä. Suoritus voi edetä myös rinnakkaisesti, mikä usein vaikeuttaa järjestelmän hallintaa.

1.2 Formaali spesifointi

Järjestelmää kehitettäessä tuotetaan spesifikaatioita, joiden tarkoituksena on kuvata järjestelmän toiminta joko kokonaan tai oleelliselta osalta. Kehitystyön edetessä spesifikaatioiden abstraktiotaso alenee siirryttäessä esimerkiksi luonnollisella kielellä esitetyistä vaatimuksista valmiiseen ohjelmakoodiin tai piirikaavioon. Hyvän spesifikaation ominaisuuksia ovat täydellisyys, tarkkuus, virheettömyys, ymmärrettävyys, testattavuus ja jäljitettävyys [HaiMär95].

Reaktiivisten järjestelmien käyttäytyminen on usein monimutkaista. Niiltä vaaditaan kuitenkin erityistä vikasietoisuutta ja robustisuutta. Järjestelmä voi olla myös turvallisuuskriittinen, jolloin sen vikaantumisesta saattaa aiheutua vaaraa ihmisille. Tällaisen järjestelmän suunnittelussa vaaditaan sellaista tarkkuutta, johon ohjelmistotuotannossa perinteisesti käytetyillä menetelmillä on vaikea päästä. Formaalit menetelmät ovat matematiikkaan pohjautuvia menetelmiä, joilla vaadittava tarkkuus ja yksiselitteisyys voidaan saavuttaa.

1.3 Työn tarkoitus

Tässä diplomityössä on kiinnostuksen kohteena reaktiivisten järjestelmien formaali määrittely. Määrittelyvaiheen spesifikaatio syntyy useimmiten informaalisti esitettyjen, järjestelmän toiminnallisten vaatimusten pohjalta ja toimii järjestelmän suunnittelun lähtökohtana. Käyttäessään formaalia spesiointimenetelmää määrittelijä rakentaa spesifikaation menetelmän tarjoamin keinoin. Tässä työssä verrataan toisiinsa kahden menetelmän tarjoamia spesifikaatioiden strukturoinnin rakenteita ja tarkastellaan menetelmien eroja. Lisäksi esitetään

esimerkkejä saman järjestelmän spesifikaatiosta strukturoituna kummallakin menetelmällä.

1.4 Yleiskatsaus työhön

Tämän dokumentin **toisessa luvussa** kerrotaan työhön liittyvästä teoriasta ja käsitteistä eli formaaleista menetelmistä, spesifikaatioista ja niiden strukturoinnista. **Kolmannessa luvussa** esitellään aktioiden aikalogiikka, TLA, sen pohjana olevan aikalogiikan perusteet, operaattorit ja muut myöhempien lukujen kannalta keskeiset asiat. **Neljännessä luvussa** esitellään TLA^+ , spesifointikieli, jonka tarkoituksena on helpottaa TLA-spesifikaatioiden ymmärrettävyyttä. Luvussa käsitellään niitä lisäominaisuuksia, joita TLA^+ tuo TLA:han, ja esitetään yksinkertainen esimerkki TLA^+ -spesifikaatiosta. Luku perustuu lähteeseen [Lam95]. **Viidennessä luvussa** kerrotaan DisCo-menetelmästä, valotetaan sen teoreettista pohjaa ja tarkastellaan kielen spesifikaatioiden strukturointiin tarjoamia rakenteita. Luvun loppupuolella käsitellään DisCo-menetelmää yleisellä tasolla.

Kuudennessa luvussa pohditaan DisCo-spesifikaation muuntamista TLA^+ :ksi ja muunnetaan yksinkertainen esimerkkispesifikaatio. **Seitsemännessä luvussa** vertaillaan TLA^+ :aa ja DisCoa spesifointimenetelminä ja niiden tarjoamia strukturointimahdollisuuksia. Luvussa pohditaan myös menetelmien kehittämistä ja arvioidaan, voitaisiinko joitakin TLA^+ :n ominaisuuksia ottaa mukaan DisCo-kielen tuleviin versioihin. **Kahdeksannessa luvussa** tehdään yhteenveto, jossa arvioidaan menetelmien vertailun mielekkyyttä ja työn tekemistä yleensä.

2 Teoriaa ja käsitteitä

2.1 Virheitä on vaikea välttää

Kun pankkiautomaatti syö kortin ja veloittaa tiliä antamatta kuitenkaan seteleitä, on varsin ymmärrettävää, että käyttäjä syyttää tapahtuneesta konetta, jonka kanssa hän asioi. Pankkiautomaattien verkko on esimerkki rinnakkaisesta, hajautetusta, sulautetusta ja reaktiivisesta järjestelmästä, jonka kehitystyön eri vaiheissa on lähes mahdotonta varmistaa, että se toimii kaikissa mahdollisissa tilanteissa oikein. Testauksesta huolimatta virheitä jää huomaamatta ja ne aiheuttavat vikoja, jotka käyttäjä havaitsee häiriöinä [HaiMär95].

Laajat tietotekniset järjestelmät ovat yleensä monimutkaisia ja niissä on paljon virheitä. Järjestelmän koon kasvaessa kasvaa myös virheiden määrä, kuten niin sanottu lentokonesääntö asian ilmaisee: "Kaksimoottorisessa lentokoneessa on kaksi kertaa enemmän moottoriongelmia kuin yksimoottorisessa".

Usein virheet ovat onneksi varsin harmittomia ja niistä voidaan selvittää esimerkiksi käynnistämällä järjestelmä uudelleen. Jotkin järjestelmät ovat luonteeltaan sellaisia, että tämä ei käy päinsä. Kesäkuussa 1996 Ariane 5 -raketti tuhoutui pian nousun jälkeen. Tuhon syyksi paljastui virhe raketin ohjelmistossa, jonka aiheuttama vika sai raketin harhautumaan radaltaan ja tuhoamaan itsensä [ESA96]. Taloudelliset menetykset olivat valtavat ja Euroopan avaruusjärjestön avaruusohjelma koki vakavan takaiskun.

Sovellusta, jonka häiriöstä voi koitua vaaratilanteita, kutsutaan kriittiseksi. Kriittisten sovellusten toteuttaminen ohjelmistoina jakaa mielipiteitä. Toisaalta ohjelmistotekniikkaa ei alana pidetä kyllin kehittyneenä, toisaalta nähdään ohjelmistojen mahdollisuudet [Par96]. Euroopan avaruusjärjestö ei voi edes harkita ohjelmistoista luopumisen kaltaista ylellisyyttä.

Parnas [Par96] luettelee kolme seikkaa, joihin kriittistä ohjelmistoa kehitettäessä on kiinnitettävä huomiota:

1. tarkka, hyvin organisoitu ja matematiikkaa hyväksi käytävä dokumentaatio ja systemaattiset katselmukset
2. laaja testaus
3. pätevien ihmisten ja hyväksytyin tuotantoprosessin käyttö.

Ariane 5 -rakettia kehitettäessä tehtiin varmasti paljon työtä virheiden välttämiseksi, siinä kuitenkaan täysin onnistumatta. Parhaimpienkaan oppien soveltaminen ei aina auta, sillä suunnittelijat ovat vain ihmisiä ja ihmiset tekevät inhimillisiä erehdyksiä. Parnas toteaaakin: "One can only avoid errors by avoiding humans."

2.2 Formaalien menetelmien perusteet

Virheiden välttämiseen on kuitenkin aina pyrittävä, olipa kyseessä sitten millainen järjestelmä tahansa. Kaikki menetelmät, jotka auttavat välttämään virheitä tai ainakin havaitsemaan ne mahdollisimman aikaisessa vaiheessa, on otettava vakavasti.

Insinööritieteet ovat perinteisesti perustuneet matematiikkaan, joka antaa keinot taistella monimutkaisuutta vastaan. Kun ongelma saadaan kuvattua matematiikan kielellä, voidaan sen ratkaisussa käyttää apuna matematiikan järeää koneistoa. Menetelmiä, jotka matematiikan avulla pyrkivät vähentämään inhimillisten erehdysten mahdollisuutta ohjelmistojen ja laitteistojen kehitystyössä, kutsutaan formaaleiksi menetelmiksi. Ne yhdistävät matematiikan täsmällisen semantiikan ohjelmointikielistä tuttuun täsmälliseen syntaksiin.

Formaalin menetelmän ideana on siirtää suunnittelijan huomio *miten*-tasolta *mitä*-tasolle. Menetelmät pyrkivät kuvaamaan ongelmat sellaisessa muodossa, jossa toteutuksen tekniset yksityiskohdat eivät häiritse päättelyä. Matematiikan koneisto mahdollistaa myös järjestelmän ominaisuuksien varmentamisen todistamalla. Menetelmien suurimmat edut ovat tulosten yksiselitteisyys ja tarkkuus, ja niitä käytetään pääasiassa järjestelmän vaatimusasettelun, määrittelyn ja suunnittelun apuna. Osa menetelmistä tukee myös järjestelmän toteutus- ja testausvaiheita. Ideaalitapauksessa koko järjestelmän kehitys aloitetaan tekemällä informaalilla tavalla esitettyjen vaatimusten pohjalta formaali spesifikaatio, jota tarkennetaan formaalisti aina toteutukseen asti ja testataan menetelmän tuottamalla testiaineistolla. Menetelmiä, jotka tekisivät kaiken tämän automaattisesti, ei ainakaan vielä ole näköpiirissä, ja vaikka sellainen olisikin käytettävissä, jää suunnittelijan itsensä pohdittavaksi aina kuitenkin se, vastaavatko lähtökohdaksi annetut vaatimukset sittenkään todellisuutta. [Val94]

Se matematiikan osa-alue, jota formaalit menetelmät yleensä soveltavat, on logiikka. Tämä on luonnollista, koska tietoteknisen järjestelmän kehittämisessä suunnittelijalla on äärimmäisen looginen vastustaja, tietokone.

Formaaleista määrittelymenetelmistä tunnetuimmat ovat VDM ja Z, jotka perustuvat joukko-oppiin ja tavalliseen logiikkaan. Tärkeitä määrittelykeinoja ovat myös kieliopit, tyyppitetyt algebrat ja muunnossäännöt sekä aikalogiikka, joka sopii hyvin rinnakkaisten järjestelmien määrittelyyn. [Val94]

Kriittisten sovellusten ohella formaalien menetelmien käyttökohteita ovat hankalasti korjattavat järjestelmät ja ohjelmat, jotka kirjastoidaan ja käytetään uudelleen [Val94]. Tämä diplomityö tutkii reaktiivisten järjestelmien määrittelyä aikalogiikkaan pohjautuvien formaalien menetelmien avulla. Järjestelmät ovat usein sulautettuja, eikä määrittelyvaiheessa aina edes vielä tiedetä, mitkä osat tullaan toteuttamaan ohjelmistona ja mitkä piireinä.

2.3 Formaalien menetelmien käyttö

Käytännössä on havaittu, että formaalien menetelmien käyttö vähentää virheitä ja parantaa ohjelmistojen huollettavuutta. Niiden käyttö pakottaa ajattelemaan asioita tavallista tarkemmin, ja vaikka mitään erityistä menetelmää ei käyttäisikään, usein jo ajattelutavan muutos formaalimpaan suuntaan vähentää virheitä.

Formaalien menetelmien käyttö voi olla huomattavan työlästä. Menetelmien soveltajilta vaaditaan myös uusia taitoja esimerkiksi ohjelmistotuotannossa perinteisesti käytettyihin menetelmiin verrattuna. Tänä päivänä eräät standardit kuitenkin jo vaativat formaalien menetelmien käyttöä, joten sekä menetelmien kehittämiseen että alan koulutukseen on syytä panostaa. [Val94]

Järjestelmän voi testaamisella osoittaa toimimattomaksi, mutta ei täysin toimivaksi. Monimutkaista järjestelmää ei voi mitenkään testata täydellisesti. Vaikka formaalien menetelmien käyttö ei olisikaan kivutonta, on täydellinen testaaminen yksinkertaisesti mahdotonta. Formaalit menetelmät eivät kuitenkaan ole ratkaisu kaikkiin ongelmiin, vaikka tiukat formalistit ehkä näin ovat joskus uskoneet. Teollisuus ei ole ollut laajassa mittakaavassa kiinnostunut formaalien menetelmien käyttöönotosta ohjelmistotuotannossa. Tällä hetkellä vaikuttaa siltä, että formaalit menetelmät ja ohjelmistotuotannossa perinteisesti käytetyt menetelmät ovat ottamassa askelia toisiaan kohti. Monet formalistit ovat ymmärtäneet informaalien menetelmien edut ja kiinnostus virheiden vähentämiseen formaalien menetelmien avulla kasvaa. Menetelmälle, joka yhdistäisi formaalien ja informaalien menetelmien edut, olisi varmasti kysyntää.

Nopeaa siirtymistä formaalien menetelmien käyttöön ei ole näköpiirissä. Tämä ei johdu pelkästään menetelmien puutteista. Käytäntö on osoittanut, että paljonkin virheitä sisältävät järjestelmät toimivat oikeastaan uskomattoman hyvin.

2.4 Spesifioinnin tarkoitus

Ohjelmistojen tuottamisen sanotaan olevan dokumenttien tuottamista. Tuotetut dokumentit ovat spesifikaatioita, jollaisia syntyy ohjelmiston tuotantoprosessin jokaisessa vaiheessa. Tuotantoprosessi etenee useimmiten luonnollisella kielellä lausutuista vaatimuksista valmiiseen, ohjelmointikielellä esitettyyn toteutukseen. Kussakin vaiheessa tehty spesifikaatio kertoo miten ohjelmisto tällä tasolla toteutetaan ja samalla mitä seuraavan tason pitää tehdä. Jos esimerkiksi määrittelydokumentissa lukee: "Yhtäaikaisten käyttäjien suurin sallittu määrä on kymmenen", voivat suunnittelijat kirjoittaa suunnitteludokumenttiin: "Vakio `MAX_USERS` kertoo yhtäaikaisten käyttäjien suurimman sallitun määrän. Vakio on tyyppiä `integer`, se alustetaan arvoon kymmenen ja sen arvo on aina 0 tai suurempi." Laajemman kuvan spesifikaation roolista ohjelmistotyössä antaa [HaiMär95].

Spesifioinnin tärkein tavoite on laatia spesifikaatio, jossa määritellään vaatimukset mahdollisimman täydellisesti, tarkasti ja virheettömästi. Ohjelmistot pyritään rakentamaan siten, että muutokset johonkin ohjelmiston osaan voidaan tehdä ilman, että muita osia täytyisi muuttaa. Tämä auttaa yleensä sellaisten virheiden korjaamista, jotka on tehty tuotantoprosessin loppupuolella. Prosessin alkupäässä tehtyjen virheiden korjaaminen voi pahimmillaan johtaa ohjelman arkkitehtuurin muuttumiseen. Jos ohjelma on jo toteutettu, kun tällainen perustavaa laatua oleva virhe havaitaan, tulee sen korjaaminen hyvin kalliiksi.

Hyvä spesifikaatio pyrkii esittämään järjestelmän toiminnan asianmukaisella tarkkuustasolla mahdollisimman yksiselitteisesti. Spesifikaatio voidaan kirjoittaa luonnollisen kielen ja ohjelmointikielen lisäksi myös erityisesti spesifointiin tarkoitettulla kielellä. Tällaiset kielet ovat yleensä formaaleja, eli niillä on tarkkaan määritelty kielioppi. Luonnollisella kielellä esitetty spesifikaatio voi olla oikeastaan mitä vain ja yksiselitteisyys jää yleensä saavuttamatta. Toteutuksen perusteella tehtävää päättelyä taas hankaloittaa liiallinen yksityiskohtaisuus. Spesifointikieli tarjoaa tarkan syntaksin ja mahdollisuuden kuvata järjestelmä korkealla abstraktiotasolla. Kielet on yleensä suunniteltu tietäntyyppisiä ongelmia varten ja niihin voi liittyä työkaluja, jolloin niitä voidaan kutsua spesifointimenetelmiksi. Työkalut voivat auttaa määrittelijää esimerkiksi järjestelmän ominaisuuksien todistamisessa tai spesifikaation toteutuksessa. Spesifointikielten ohjelmointikieliä korkeampi abstraktiotaso saavutetaan esimerkiksi ottamalla käyttöön rajoittamattomia lukualueita.

Järjestelmän kehittäminen alkaa mahdollisimman yleisen spesifikaation laadimisesta, josta ylhäältä alas -menetelmän mukaisesti edetään pieniin yksityiskohtiin asti. Usein halutaan kuitenkin spesifioida jo olemassa oleva järjestelmä, jotta siinä mahdollisesti olevat virheet pystyttäisiin paikantamaan ja korjaamaan. Tällöin kannattaa useimmiten edetä alhaalta ylös, spesifioimalla ensiksi pieniä itsenäisesti toimivia kokonaisuuksia, joista spesifioinnin loppuvaiheessa kootaan kokonainen järjestelmä. Näin toimitaan myös, jos halutaan tehdä kirjastoitavia, uudelleenkäytettäviä komponentteja. [Setä92]

2.5 Spesifikaation laatiminen

Spesifikaation laadimisessa voi tehdä samat virheet kuin ohjelman kirjoittamisessa, sillä sen kirjoittamisen voi aloittaa ilman mitään kokonaiskuvaa siitä, mitä on spesifioimassa. Tällaisen lähestymistavan motiivi on yleensä ajan säästö, mutta todellisuudessa se johtaa vain siihen, että sitä mukaa kun käsitys järjestelmästä paranee, spesifikaatio joudutaan kirjoittamaan uudelleen, ja kallista aikaa kuluu hukkaan.

Spesifikaatio on määritelmä, eikä syntaksiltaan oikeaoppisesta määritelmästä voida sanoa onko se oikea vai väärä. Spesifikaatio kuitenkin ilmaisee jotain tarkoitusta, ja jos se tässä epäonnistuu, voidaan sen sanoa olevan väärä. Jos sekä spesifikaatio että alunperin asetetut vaatimukset lausutaan logiikan kielellä, voi-

daan aina todistamalla tarkastaa implikoiko spesifikaatio vaatimukset. Jos näin on, voidaan sen sanoa ilmaisevan tarkoitustaan. [Lam93]

Systeemiä sanotaan suljetuksi, jos sen toimintaa voidaan tarkkailla ulkopuolelta, mutta siihen ei voida vaikuttaa. Avoin systeemi sitä vastoin on sellainen, joka on jatkuvassa vuorovaikutuksessa ympäristönsä kanssa. Järjestelmää spesifioitaessa sen ympäristö voidaan yleensä ottaa mukaan spesifikaatioon. Näin saadaan suljettu systeemi, joka mahdollistaa myös sellaisten ominaisuuksien todistamisen, joiden olemassaolo edellyttää ympäristön toimintaa. Suljettu systeemi kuitenkin toteutetaan avoimena systeiminä. [Mik95]

Spesifioinnin voisi sanoa olevan oma taiteen lajinsa, joka vaatii sekä kykyä ymmärtää järjestelmän toiminta kokonaisuutena että tarkkaa yksityiskohtien hallintaa. Ennen kaikkea spesifiointi vaatii kokemusta. On tärkeää, että määrittelijä löytää heti alussa järjestelmästä keskeiset käsitteet, ne oliot jotka ovat sen toiminnan kannalta keskeisiä. Määrittelijän on kiinnitettävä huomiota myös spesifikaation ymmärrettävyyteen, sillä spesifikaatiosta, jota vain sen kirjoittaja ymmärtää, ei ole mitään hyötyä. Spesifiointikielellä tehdyn spesifikaation ymmärrettävyyttä on aina syytä parantaa kommentoimalla se kattavasti luonnollisella kielellä. Kommenttien lisäksi tarvitaan usein dokumentaatiota, joka kertoo mitä järjestelmän on tarkoitus tehdä ja miten spesifikaatio on organisoitu [Lam93]. Nyrkkisääntönä voisi mainita sen, että mitä matemaattisempaa spesifiointikieltä käytetään, sen enemmän spesifikaatiota pitää kommentoida ja dokumentoida.

2.6 Strukturointi

Strukturoinnin tarpeellisuus havaittiin 1960-luvulla, kun huomattiin, että useimmat ohjelmat olivat täysin mahdottomia ylläpitää. Ohjelmien virheitä ei voitu korjata, koska dokumentoimattoman koodispaletin ymmärtäminen oli täysin mahdotonta jopa sen kirjoittajalle. Koska tarve yhä laajempien ohjelmistojen kehittämiseen kuitenkin kasvoi, haluttiin löytää keinoja, joilla ohjelmista saataisiin virheettömämpiä ja helpompia ymmärtää. Alettiin puhua rakenteisesta ohjelmoinnista (structured programming), joka toi mukanaan mm. ajatuksen siitä, että ohjelman lähdekoodista pitäisi nähdä suoraan miten ohjelman suoritus etenisi. Tämä helpotti olennaisesti ohjelmien toiminnan ymmärtämistä. Hyvä selostus rakenteisesta ohjelmoinnista on lähteessä [Seth96].

Pian havaittiin myös, että jos ohjelman hajottaa pienemmiksi kokonaisuuksiksi, moduuleiksi, sitä on helpompi ymmärtää. Tämä johtuu siitä, että ihminen hahmottaa kokonaisuuksia parhaiten kun ne koostuvat enintään 7 ± 2 palasta. Jokainen pala voi sitten sisältää kuinka paljon tietoa tahansa. Ominaisuutta kutsutaan Millerin laiksi [Mil56].

Mikäli moduulin rajapinta toteutetaan järkevästi ja pystytään rajoittamaan moduulin käyttö tapahtumaan vain tämän rajapinnan kautta, voidaan moduulin sisältöä korjata ilman, että sitä käyttäviä ohjelman osia täytyisi muuttaa.

Koodin käyttö uudelleen on myös mahdollista, sillä hyvin suunniteltua moduulia voidaan käyttää muissakin ohjelmissa. Modulaarisuus mahdollistaa lähes kaikki sellaiset ominaisuudet, joita nykypäivänä pidetään olio-ohjelmoinnin suurimpina etuina.

Strukturilla tarkoitetaan spesifoinnin yhteydessä niitä rakenteita, jotka muodostavat järjestelmän spesifikaation. Hyvin strukturoitu spesifikaatio ilmentää sen laatijoiden tarkoitusta ja antaa seuraavan vaiheen suunnittelijoille yksiselitteisen ja tarkan selvityksen siitä, mitä järjestelmän tulee tehdä. Tällaisen spesifikaation tulee mahdollistaa tarkastelu sekä yleisellä että yksityiskohtaisella tasolla. Mitä laajempi spesifikaatio on, sitä tärkeämpää on kiinnittää huomiota sen strukturiin. Hyvä strukturi tekee spesifikaatiosta helpommin ymmärrettävän ja ylläpidettävän.

Modulaarisuuden edut tulevat hyvin esille laajoissa spesifikaatioissa. Ymmärrettävyys yleensä paranee, kun suuri spesifikaatio pilkotaan moduuleihin, joita Millerin lain mukaan ei tulisi olla yli seitsemää kappaletta. Jos moduulit ovat useiden sivujen pituisia, ne kannattaa pilkkoa alimoduuleiksi. Näin spesifikaatio saa pikemminkin syvyys- kuin pituussuuntaisen struktuurin.

3 TLA

3.1 Logiikka rinnakkaisuuden hallintaan

TLA (Temporal Logic of Actions) eli aktioiden aikalogiikka on Leslie Lamportin kehittämä logiikka, jonka avulla voidaan spesifioida rinnakkaisia järjestelmiä ja tehdä päättelyitä niiden ominaisuuksista [Lam94]. TLA:n ideana on tehdä päättely algoritmin ominaisuuksista mahdollisimman yksinkertaiseksi. Kun algoritmi lausutaan logiikan kielellä, saadaan sen toimintaa kuvaava TLA-kaava, jonka pohjalta päätelmät algoritmin ominaisuuksista tehdään. Päätelmien tekemisen helpottamiseksi on joukko valmiita todistussääntöjä. TLA:ssa ei eroteta toisistaan spesifikaatiota ja ominaisuutta, vaan molemmat ovat kaavoja. Näin saavutetaan se etu, että spesifikaation kaava implikoi loogisesti ominaisuuden kaavan.

TLA:n perusta on aikalogiikka, johon on lisätty aktio-käsite. Jokainen TLA-spesifikaatio on muutettavissa aikalogiikan spesifikaatioksi [Setä92]. Logiikka on haluttu pitää mahdollisimman yksinkertaisena, ja sen täydellinen formaali semantiikka mahtuukin tiivistettynä yhdelle sivulle. TLA ei tunne tyyppejä, joten muuttujat voivat saada mitä arvoja tahansa. Tämä on tietoinen valinta, sillä tyyppikäsite olisi tuonut logiikkaan ylimääräistä painolastia.

Huomattavasti täydellisempi esitys TLA:sta on [Lam94], johon tämä luku pitkälti perustuu.

3.2 Aktioiden logiikkaa

Oletetaan, että käytössämme on arvojen joukko. Joukon alkioita ovat esimerkiksi kokonaisluvut, kuten -5 , 0 ja 405 , sekä merkkijonot kuten "asdf". Lisäksi on ääretön määrä muuttujia, joihin näitä arvoja voi sijoittaa. Otamme käyttöön myös totuusarvot tosi ja epätosi. *Tila* on funktio, joka kuvaa muuttujan sen arvoksi. Esimerkiksi muuttujan x arvo tilassa s on $s(x)$, jota merkitään tästä eteenpäin $s[[x]]$.

Tilafunktio on muuttujista ja arvoista koostuva lauseke. Tilafunktiolla f on tilassa s arvo $s[[f]]$. Tilafunktion arvo tilassa s saadaan, kun sen muuttujat korvataan niiden arvoilla tilassa s . Esimerkiksi tilafunktio $x^2 - y + 10$ kuvautuu tilassa s arvoksi $(s[[x]])^2 - s[[y]] + 10$. *Tilapredikaatti* on totuusarvoinen tilafunktio. Olkoon tilapredikaatti¹ $P \triangleq x^2 = y - 10$ tosi tilassa s jos ja vain jos $s[[P]] \triangleq (s[[x]])^2 = s[[y]] - 10$ on tosi. Tällöin sanotaan, että tila s tyydyttää predikaatin P .

Aktio on relaatio minkä tahansa kahden tilan välillä. Se on tavallisista ja pilkutetuista muuttujista sekä arvoista koostuva totuusarvoinen lauseke. Aktion

¹Symboli \triangleq luetaan englanniksi "is defined as" tai "equals by definition". Suomeksi sen voi lukea "määritellään" tai "on määritelmän mukaan".

tavallisten muuttujien arvot lasketaan vanhassa tilassa ja pilkutettujen uudessa tilassa. Esimerkiksi aktio $y = x' + 1$, merkitään \mathcal{A} , on tosi tilojen s ja t välillä jos ja vain jos $s[[y]] = t[[x]] + 1$, merkitään $s[[\mathcal{A}]]t$, on tosi. Tällöin paria (s, t) sanotaan \mathcal{A} -*askeleeksi*. Aktion sanotaan olevan vireessä (enabled) tilassa s , jos on olemassa sellaiset arvot pilkutetuille muuttujille, että aktion lauseke tulee todeksi, kun tavallisten muuttujien arvot lasketaan tilassa s .

Tilapredikaattia voidaan siis pitää aktiona, jossa ei ole pilkutettuja muuttujia. Jokaiselle funktiolle f ja predikaatille P , merkinnät f' ja P' tarkoittavat lausekeita, jotka saadaan kun f :ssä ja P :ssä kaikki muuttujat pilkutetaan. Aivan kuten ohjelmissa, TLA-spesifikaatioissakin on tavallisten muuttujien (flexible variables) lisäksi sellaisia muuttujia, joiden arvot eivät muutu eli niillä on aina sama arvo tilasta riippumatta (rigid variables). Vain tällaisten muuttujien yli saa kvantifioida tilafunktioiden ja aktioiden lausekkeissa. Ohjelmointikielissä muuttujaa, jonka arvo pysyy vakiona kutsutaan vakioksi, mutta matematiikka ei tunne sellaista käsitettä.

3.3 Aikalogiikka

Aikalogiikassa algoritmin suorituksen ajatellaan olevan sekvenssi askeleita siten, että jokainen askel tuottaa uuden tilan muuttamalla yhden tai useamman muuttujan arvoa. Askeleen sanotaan olevan änkyttävä niiden muuttujien suhteen, joiden arvoja se ei muuta. Merkintä $[\mathcal{A}]_f$, jossa f on kaikkien muuttujien joukon osajoukko, tarkoittaa $\mathcal{A} \vee (f' = f)$, eli tapahtuu joko \mathcal{A} -askel tai *änkytysaskel*, jossa f :n muuttujien arvot eivät muutu. Usein merkitään vain $[\mathcal{A}]$, jos joukko f on selvä asiayhteydestä. Vastaavasti merkintä $\langle \mathcal{A} \rangle_f$ tarkoittaa $\mathcal{A} \wedge (f' \neq f)$, eli että askel on sellainen \mathcal{A} -askel, joka ei ole änkytysaskel. Askeleista syntyy tilaketju ja äärettömän pitkää tilaketjua kutsumme *käyttäytymiseksi*.

Käyttäytyminen vastaa algoritmin suoritusta, joten päättely algoritmeista siis edellyttää päättelyä käyttäytymisistä. Reaktiivisen järjestelmän kuvitellaan toimivan ikuisesti. Tällöin on tärkeää, että myös käyttäytymiset ovat äärettömän pitkiä tilaketjuja. Ongelmia ei kuitenkaan synny spesifioitaessa järjestelmiä, jotka pysähtyvät jossakin tilassa, sillä pysähtymisen jälkeen voidaan toistaa änkytysaktiota loputtomiin. Yleensä ei ollakaan kiinnostuneita siitä, voidaanko järjestelmän pysähtyminen todistaa, vaan siitä, että järjestelmä todella tekee jotain. On helppo tehdä järjestelmä, joka täyttää kaikki turvallisuusvaatimuksensa, nimittäin sellainen, joka vaatimukset täyttävän alkutilan jälkeen vain änkyttää.

Tilapredikaatti P on tosi käyttäytymisellä σ , jos se pätee käyttäytymisen σ ensimmäisessä tilassa. Jos tilapredikaatti P on tosi käyttäytymisen jokaisessa tilassa, merkitään $\Box P$, ja luetaan "aina P ". Vastaavasti aktio \mathcal{A} on tosi käyttäytymisellä σ , jos käyttäytymisen σ ensimmäinen askel on \mathcal{A} -askel. Aktiolle \mathcal{A} lauseke $\Box \mathcal{A}$ on tosi jos ja vain jos jokainen askel on \mathcal{A} -askel. Merkintä $\Box[\mathcal{A} \vee \mathcal{B}]$ siis tarkoittaa, että jokainen askel on joko \mathcal{A} -askel tai \mathcal{B} -askel tai änkytysaskel

joka ei muuta muuttujien arvoja. Käyttämällä lyhennysmerkintää $\diamond P$ vastamaan lauseketta $\neg \square \neg P$ saamme käyttöön operaattorin, joka kertoo, että P on tosi jossakin käyttäytymisen tilassa. Nämä operaattorit yhdistämällä saadaan operaattorit $\square \diamond$ ja $\diamond \square$, jotka tarkoittavat intuitiivisesti “äärettömän usein” ja “joskus jatkuvasti”. Merkintä $P \rightsquigarrow Q$ määritellään $\square(P \Rightarrow \diamond Q)$, joka tarkoittaa sitä, että aina kun P on tosi, on myös Q tosi joko samassa tai jossakin myöhemmässä tilassa.

Koska kvantifioitaessa olemassaolo- tai kaikkikvanttorin avulla voidaan muuttuja sitoa vain yhdessä tilassa, on tarpeen ottaa käyttöön kvanttoreiden temporaaliversiot, jotka sitovat muuttujan tilaketjun jokaisessa tilassa. Temporaalista olemassaolokvanttoria merkitään \exists ja temporaalista kaikkikvanttoria \forall .

Sekä algoritmit että ominaisuudet esitetään kaavoina. Jos algoritmin toimintaa kuvaava TLA-kaava F on tosi jollakin käyttäytymisellä σ , kuvaa σ algoritmin mahdollisen suorituksen. Kaava F tyydyttää ominaisuuden G jos $F \Rightarrow G$ on tosi kaikilla käyttäytymisillä. Tällöin G on tosi jokaisella algoritmin suoritusta kuvaavalla käyttäytymisellä.

3.4 Turvallisuus- ja elävyysominaisuudet

Spesifioitavan järjestelmän ominaisuudet voidaan jakaa turvallisuus- ja elävyysominaisuuksiin. Turvallisuusominaisuudet kertovat intuitiivisesti, että “jotain paha ei koskaan tapahdu” ja elävyysominaisuudet, että “jotain hyvää tapahtuu joskus”. Järjestelmä saa turvallisuusominaisuutensa, kun sille asetetaan turvallisuusvaatimuksia. Ne kertovat intuitiivisesti, mitä järjestelmä *voi tehdä*. Elävyysominaisuudet toteutetaan asettamalla reiluusvaatimuksia, jotka kertovat mitä järjestelmän *tulee tehdä*.

Turvallisuusominaisuuksien rikkoutuminen on mahdollista havaita äärellisen pituisen tilaketjun perusteella. Esimerkiksi turvallisuusvaatimuksen $\square(x \leq 10)$ rikkoutuminen havaitaan välittömästi, jos muuttujan x arvo jossakin tilassa on yli kymmenen. Sama ei päde elävyysominaisuuksien tapauksessa. Esimerkiksi vaatimuksen $\diamond(x = 10)$ rikkoutumista ei missään tilassa todista se, että x ei vielä siihen mennessä ole saanut arvoa 10, sillä x voi saada sen jossakin tulevassa tilassa. TLA:n suurimpia etuja on se, että vaatimukset järjestelmän elävyydelle voidaan esittää eksplisiittisesti [Lad96]. Reaktiivisissä järjestelmissä ei elävyysominaisuuksien olemassaolo ole yleensä itsestäänselvyys, vaan se pitää todistaa.

Elävyysominaisuudet toteutetaan asettamalla reiluusvaatimuksia, koska niin pystytään varmistamaan, että turvallisuusominaisuuksia ei vahingossa samalla kiristetä. TLA tuntee kaksi kolmesta reiluuden muodosta, heikon (weak fairness) ja vahvan (strong fairness) reiluuden. Reiluus ilmaistaan aktioita kohtaan. Heikko reiluus ilmaistaan lausekkeella $WF_f(\mathcal{A})$, joka ilmoittaa, että mikäli aktio $\langle \mathcal{A} \rangle_f$ joskus tulee vireeseen ja pysyy siitä eteenpäin vireessä jatkuvasti, niin äärettömän monta $\langle \mathcal{A} \rangle_f$ -askelta tapahtuu. Vahva reiluus $SF_f(\mathcal{A})$ taas tarkoittaa

Predicate and Action Operators

p'	[p true in final state of step]
$[A]_e$	$[A \vee (e' = e)]$
$\langle A \rangle_e$	$[A \wedge (e' \neq e)]$
ENABLED A	[An A step is possible]
UNCHANGED e	$[e' = e]$
$A \cdot B$	[Composition of actions]

Temporal Operators

$\Box F$	[F is always true]
$\Diamond F$	[Eventually: $\neg \Box \neg F$]
$WF_e(A)$	[Weak fairness: $\Box \Diamond \langle A \rangle_e \vee \Box \Diamond \neg \text{ENABLED } A$]
$SF_e(A)$	[Strong fairness: $\Box \Diamond \langle A \rangle_e \vee \Box \Diamond \neg \text{ENABLED } A$]
$F \rightsquigarrow G$	[Leads to: $\Box (F \Rightarrow \Diamond G)$]
$\exists x : F$	[Temporal existential quantification (hiding).]
$\forall x : F$	[Temporal universal quantification.]

Kuva 1: TLA:n operaattorit. [Lam95]

taa sitä, että mikäli $\langle A \rangle_f$ on vireessä äärettömän monta kertaa, niin se myös suoritetaan äärettömän monta kertaa. Jos aktio on vireessä jatkuvasti, se on vireessä äärettömän usein. Vahva reiluus siis implikoi heikon reiluuden.

Kolmas reiluuden muoto, pakotettu reiluus (impartiality) tarkoittaa sitä, että kaikki tapahtumat tapahtuvat äärettömän usein. Pakotettu reiluus voi rikkoa turvallisuusvaatimuksia.

Kaikki TLA:n operaattorit on esitelty kuvassa 1.

3.5 TLA-spesifikaatio

TLA:ssa järjestelmän suoritus esitetään siis käyttäytymisenä, joka on äärettömän pitkä tilaketju. Tilat määräytyvät systeemin muuttujien perusteella ja siirtymää tilasta toiseen sanotaan aktioksi. Päättely perustuu tiloihin ja siirtymiin niiden välillä.

TLA:ssa aktioiden ajatellaan tapahtuvan yksi kerrallaan. Aktiot ovat atomisia, eli aktion kerran alettua se tapahtuu loppuun asti, ilman että mikään muu aktio voi häiritä sen tapahtumista. Mikäli aktioilla ei ole yhteisiä muuttujia, ne voivat toteutuksessa tapahtua myös samanaikaisesti.

TLA-kaavan, joka on muotoa

$$Init \wedge \Box[\mathcal{A}] \wedge F,$$

sanotaan olevan kanonisessa muodossa. Kaavassa $Init$ ilmaisee alkutilan, \mathcal{A} on

kaikkien aktioiden disjunktio ja F kaikkien reiluusvaatimusten konjunktio. Kanonisessa muodossa olevaa spesifikaatiota, josta reiluusvaatimukset on jätetty pois, sanotaan turvallisuusspesifikaatioksi.

Kanonisessa muodossa olevan spesifikaation operationaalinen tulkinta on, että aina kun $Init$ on mahdollinen alkutila, voimme ajatella järjestelmän suorituksen etenevän niin, että alkutilan jälkeen suoritetaan \mathcal{A} :n aktioita, joita skeduloidaan siten että reiluusvaatimukset F täyttyvät. Jokainen järjestelmä, jolla on operationaalinen tulkinta, voidaan esittää kanonisessa muodossa. [Kur96a]

Spesifikaatio voi olla myös vain luettelo järjestelmälle asetettuja vaatimuksia. Tällaisella spesifikaatiolla ei kuitenkaan välttämättä ole operationaalista tulkintaa, sillä vaatimukset voivat olla ristiriidassa toistensa kanssa. Jos esimerkiksi järjestelmälle asetettavat turvallisuus- ja elävyysvaatimukset esitetään vain yksinkertaisesti muodossa $\Box P$ tai $P \rightsquigarrow \langle \mathcal{A} \rangle$, niin päädytään helposti spesifikaatioon, jota ei voida suorittaa. [Kur96a]

Esimerkiksi spesifikaatio

$$\begin{aligned} Init &\triangleq (x = 0) \wedge (y = 0) \\ \mathcal{A} &\triangleq (x' = x + 1) \wedge (y' = y) \\ \mathcal{B} &\triangleq (y' = x + 2) \wedge (x' = x) \\ Spec &\triangleq Init \wedge \Box[\mathcal{A} \vee \mathcal{B}]_{\langle x, y \rangle} \end{aligned}$$

kuvaa järjestelmän, jonka alkutila on $Init$ ja jossa on kaksi aktiota \mathcal{A} ja \mathcal{B} . Järjestelmässä on lisäksi kaksi muuttujaa x ja y , joiden arvo alussa on nolla. Aktion \mathcal{A} tapahtuessa muuttujan x arvo kasvaa yhdellä ja muuttujan y arvo pysyy samana. Aktion \mathcal{B} tapahtuessa muuttujan y arvoksi tulee muuttujan x vanha arvo lisättyinä kahdella ja muuttujan x arvo pysyy samana. Koko järjestelmän spesifikaatio $Spec$ kertoo alkutilan ja sen, että mahdolliset aktiot ovat \mathcal{A} , \mathcal{B} tai äänkytysaktio, joka ei muuta muuttujien x ja y arvoja. Tämä spesifikaatio on turvallisuusspesifikaatio, koska se ei aseta reiluusvaatimuksia. Koska reiluutta ei vaadita kummankaan aktion suhteen, kuvaa järjestelmän oikeaa suoritusta sellainenkin käyttäytyminen, jossa kumpikaan aktio ei koskaan tapahdu vaan alkutilaa toistetaan loputtomiin.

Jos aktiossa ei ilmoita eksplisiittisesti muuttujien arvoja uudessa tilassa, tulee arvoista määrittelemättömiä. Tämä tarkoittaa sitä, että aktiossa ilmoitetaan se, että kaikki muut muuttujat paitsi ne, joiden arvoa aktio muuttaa, pysyvät muuttumattomina. Tämä voidaan ilmoittaa eksplisiittisesti kuten yllä olevassa esimerkissä tai lausekkeella UNCHANGED e , jossa e on niiden muuttujien joukko joiden arvo ei muutu.

3.6 Spesifointi TLA:lla

TLA mahdollistaa spesifikaation kokoamisen alhaalta ylös. Todistukset voidaan tehdä pienemmistä kokonaisuuksista, jotka konjungoituna muodostavat koko

systemin spesifikaation [AbaLam95]. Abadi ja Lamport [AbaLam94] ovat käyttäneet tätä menetelmää reaaliaikavaatimuksia sisältävän järjestelmän spesifointiin siten, että spesifikaatioon, joka ei ota huomioon reaaliaikavaatimuksia, on ne lisätty. Vastaavasti laajan spesifikaation voi pilkkoa pienemmiksi paloiksi, joten myös eteneminen ylhäältä alas on mahdollista.

TLA:ssa toteutus merkitsee implikaatiota, eli toteutus osoitetaan oikeaksi kirjoittamalla sekä spesifikaatio että sen toteutus TLA-kaavoiksi ja todistamalla, että toteutus implikoi loogisesti spesifikaation.

Todistuksia tehtäessä on järkevää käyttää valmiita todistussääntöjä. Myös kaikki tavallisen logiikan ja aikalogiikan säännöt ovat käytettävissä. Formaaleja menetelmiä käytettäessä pitäisi todistukset pyrkiä tekemään mahdollisimman sokeasti, luottaen valmiisiin sääntöihin ja tunnettuihin tekniikoihin. Todistaminen on muutenkin tarpeeksi hankalaa, sitä ei kannata enää tehdä vaikeammaksi yrittämällä keksiä pyörän uudelleen.

Lauseketta, joka on aina tosi, kutsutaan invariantiksi. Jos haluamme esimerkiksi todistaa, että P on invariantti, voimme käyttää apuna valmista sääntöä:

$$\frac{\begin{array}{l} P \\ \square[\mathcal{A}] \\ \square[P \wedge \mathcal{A} \Rightarrow P'] \end{array}}{\square P}$$

Sääntö sanoo, että jos P pätee alkutilassa ja aina tapahtuu joko aktio \mathcal{A} tai äänkytysaktio ja \mathcal{A} säilyttää P :n voimassa, niin voimme päätellä, että $\square P$ pätee, eli P on invariantti.

Temporaalisen olemassaolokvanttorin avulla voidaan muuttujia piilottaa kvantifioimalla ne pois määritelmistä. Kun spesifikaatio annetaan kanonisessa muodossa, siitä yleensä piilotetaan apumuuttujat, joiden arvoilla ei järjestelmän ulkoisen käyttäytymisen kannalta ole merkitystä.

4 TLA⁺

4.1 Uuden spesifointikielen tarkoitus

Lamport on esittänyt väitteen, että TLA-spesifikaation pitäisi olla helpommin ymmärrettävä kuin vastaavan pituinen ohjelma kirjoitettuna Pascalilla. Hän perustelee väitettään sillä, että TLA:n semantiikka on yksinkertaisempi kuin Pascalin. Se, että matemaatikot eivät ole kehittäneet selkeitä esitysmuotoja pitkille matemaattisille kaavoille, on hänen mukaansa syy siihen, miksi kaavat vaikuttavat monimutkaistuvan pituuden kasvaessa. [Lam94]

Vaikka TLA onkin varsin yksinkertainen formalismi, tulee monimutkaisten järjestelmien spesifikaatioista väistämättä laajoja ja vaikeasti ymmärrettäviä. TLA⁺ on formaali spesifointikieli, joka on tarkoitettu TLA-kaavojen kirjoittamiseen ja väitösten esittämiseen niiden voimassaolosta. Uuden spesifointikielen tarkoituksena on tehdä spesifikaatiot helpommin ymmärrettäviksi. Kieleen on otettu mukaan monia ohjelmointikielistä tuttuja rakenteita, joiden avulla spesifikaatioille on mahdollista antaa selkeä struktuuri. Hyvin strukturoitu spesifikaatio ei helpota ainoastaan seuraavan vaiheen suunnittelijoiden työtä, vaan se tarjoaa usein myös määrittelijälle itselleen selkeämmän kuvan spesifioitavan järjestelmän toiminnasta. Päättyvä järjestelmän ominaisuuksista on yleensä sitä helpompaa mitä selkeämpi tämä kuva on.

TLA⁺ on TLA:n syntaktinen laajennus, joka on karkeasti arvioiden vain TLA, johon on lisätty syntaktista sokeria. Kieleen on lainattu rakenteita ja nimeämiskäytäntöjä mm. Modula-ohjelmointikielestä ja Z-spesifointikielestä. Tavoitteena on ollut helpottaa monimutkaisuuden hallintaa laajoissa spesifikaatioissa, mutta säilyttää samalla TLA:n semanttinen yksinkertaisuus. Suurin ja näkyvin ero TLA:han verrattuna on moduulien käyttöönotto, joka mahdollistaa laajojen spesifikaatioiden pilkkomisen pienemmiksi ja helpommin hallittaviksi kokonaisuuksiksi. Lisäksi kieli tarjoaa operaattoreita tietorakenteiden määrittelyyn ja käsittelyyn. TLA⁺ mahdollistaa myös dokumentaation tekemisen suoraan spesifikaatioon, joten erillistä dokumentaatiota ei enää välttämättä tarvita. TLA⁺ ei kuitenkaan ole hopealuoti, joka tekisi spesifioinnista ratkaisevasti helpompaa kuin mitä se on tähän asti ollut.

4.2 Kielen käsitteet ja rakenteet

TLA⁺:n syntaksi ei ole vielä tätä kirjoitettaessa vakiintunut, joten seuraavassa esitellään vain kielen tärkeimmät käsitteet ja rakenteet. Täydellisemmän kuvan kielestä saa lähteistä [Lam95], joka on tähän mennessä laajin esitys kielestä, ja [Lam96], jossa kielen syntaksi on määritelty BNF-notaatiolla. Tämä luku perustuu hyvin pitkälle ensiksi mainittuun lähteeseen.

TLA⁺-spesifikaation pienin täydellinen yksikkö on moduuli. Moduuli koostuu parametreista (**parameters**), operaattorien ja funktioiden määritelmistä (**de-**

fnitions), oletuksista (**assumptions**) sekä teoreemoista (**theorems**).

Parametrit ovat joko totuusarvoja tai muuttujia. Muuttujat voivat olla joko vakioparametreja, jotka vastaavat TLA:ssa sellaisia muuttujia, joiden arvot eivät muutu (rigid variables), tai muuttujaparametreja, jotka vastaavat TLA:n tavallisia muuttujia (flexible variables).

TLA⁺-spesifikaatiot koostuvat suurimmalta osin operaattorien ja funktioiden määritelmästä. Operaattorit eroavat funktioista siinä, että operaattorilla O (ilman argumenttia) ei ole arvoa kun taas funktiolla f on jokin arvo. On siis syntaktisesti oikein kirjoittaa $1 + f$, mutta ei $1 + O$.

Määritelmässä voi esiintyä aiemmin esiteltyjä operaattoreita ja funktioita, mutta formaalien parametrien täytyy olla ennestään esittelemättömiä tunnisteita.

Operaattorien määritelmät eivät voi olla rekursiivisia, mutta funktioiden määritelmät voivat. Esimerkiksi kertoma-funktio luonnollisille luvuille voidaan määrittellä [Lam95]:

$$Fact[n : Nat] \triangleq \text{if } n = 0 \text{ then } 1 \\ \text{else } n * Fact[n - 1]$$

TLA:n operaattorit ovat käytössä myös TLA⁺:ssa, jossa niitä kutsutaan ei-vakio-operaattoreiksi. Niiden lisäksi on käytössä myös vakio-operaattoreiksi kutsuttuja operaattoreita.

Oletukset ja teoreemat ovat totuusarvoisia määritelmiä. Ne eroavat muista määritelmistä siinä, että ne alkavat avainsanalla (keyword) **assumption(s)** tai **theorem(s)**. Ne voivat määrittellä vain totuusarvoisia operaattoreita, joilla ei ole formaaleja parametreja. Oletukset eivät voi sisältää muuttujaparametreja tai mitään ei-vakio-operaattoreita kuten \square , \exists , $'$ tai **ENABLED**.

Oletukset ja teoreemat voivat edeltää moduulin sisällä niiden operaattorien määritelmiä, joihin ne viittaavat, mutta oletuksien ja teoreemojen nimiä ei saa käyttää muualla moduulissa.

Moduulin sanotaan olevan voimassa (valid) jos ja vain jos oletukset implikoivat teoreemat. Tämä tarkoittaa sitä, että jos A_1, \dots, A_n ovat moduulin oletuksia, niin $A_1 \wedge \dots \wedge A_n \Rightarrow T$ on tosi kaikilla moduulin teoreemoilla T .

TLA⁺:ssa ei ole tyyppejä, mutta niitä korvaavat vakiojoukot, sillä voimme määrittellä muuttujia tyyliin $phase \in \{\text{“Walking”}, \text{“Sitting”}, \text{“Rising”}, \text{“Ready”}\}$.

TLA⁺:ssa kaikilla syntaktisesti oikeilla lauseilla on jokin merkitys. Esimerkiksi nollalla jakaminen tuottaa jonkin arvon, emme vain pysty sanomaan minkä, emmekä toisaalta ole edes kiinnostuneita, mistä arvosta on kysymys.

4.3 Operaattorit

Kaikki TLA⁺:n vakio-operaattorit on esitelty kuvassa 2. Useat niistä ovat tuttuja predikaattilogiikasta tai joukko-opista.

TLA⁺:n erikoisuus on operaattori CHOOSE, joka tunnetaan myös Hilbertin ε -symbolina. CHOOSE määrittellään siten, että jos on olemassa sellainen arvo muuttujalle x , että totuusarvoinen lauseke p on tosi, niin CHOOSE $x : p$ saa jonkin sellaisen arvon. Jos sopivia arvoja on useita, sitä mikä valitaan ei määritellä, mutta valitaan kuitenkin aina sama arvo. Tällöin CHOOSE $x : p$ ja CHOOSE $x : q$ saavat saman arvon² jos $p \equiv q$. Jos sopivaa arvoa ei ole, on lausekkeen arvo määrittelemätön.

Lauseke SUBSET S tarkoittaa S :n kaikkien osajoukkojen joukkoa ja UNION S kaikkien S :n alkioiden unionia. Tästä seuraa, että lauseke UNION (SUBSET S) saa arvon S , olipa S mikä tahansa joukko.

TLA⁺:ssa tietue on funktio, jonka määrittelyalue on tietueen kenttien nimien joukko. Lauseke $r.xyz$ on lyhenne lausekkeesta $r[“xyz”]$, jonka arvo on tietueen r xyz -komponentin arvo.

Tuplat ovat myös funktioita, esimerkiksi n -tuplan $\langle e_1, \dots, e_n \rangle$ määrittelyalue on $\{1, \dots, n\}$. Tupla kuvaa i :n e_i :ksi, kun $1 \leq i \leq n$.

Merkkijonot kirjoitetaan TLA⁺:ssa lainausmerkein ympäröityinä (“abc”) ja luvut totuttuun tyyliin (234). Lukuja käytettäessä on huomioitava, että sen hetkellä näkyvyysalueella on operaattorin *Numeral* oltava määriteltynä, koska 234 on vain lyhennysmerkintä lausekkeelle *Numeral*(“243”). Ongelma kiertetään ottamalla käyttöön jokin valmiiksi määritelty moduuli, joka määrittelee operaattorin *Numeral*. Sama pätee desimaaliluvuillekin, sillä 3.234 on lyhennysmerkintä lausekkeelle *Decimal*(“3”, “234”).

4.4 Strukturoinnista

TLA:ssa spesifikaatiot strukturoidaan kirjoittamalla määritelmiä, joista kokonaiset spesifikaatio kootaan. Monimutkaiset määritelmät voidaan hajottaa pienemmiksi kokonaisuuksiksi, jotka konjunktiona muodostavat koko määritelmän. TLA⁺:ssa spesifikaation laatija voi lisäksi koota laajan spesifikaation useasta moduulista ja käyttää ohjelmointikielistä tuttuja rakenteita spesifikaation strukturointiin. Käytettävissä ovat seuraavat rakenteet:

if p then e_1 else e_2 -lauseke saa lausekkeen e_1 arvon, jos p on tosi ja lausekkeen e_2 arvon, jos p on epätosi. Lausekkeet e_1 ja e_2 eivät saa olla totuusarvoisia.

²Symboli \equiv merkitsee, että sen molemmilla puolilla olevilla loogisilla lausekkeilla on sama totuusarvo. Määritelmän mukaan $A \equiv B$ tarkoittaa samaa kuin $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Samoin määrittellään myös looginen ekvivalenssi \iff , joka luetaan “jos ja vain jos”.

Logic

TRUE FALSE \wedge \vee \neg \Rightarrow \equiv
 $\forall x : p$ $\exists x : p$ $\forall x \in S : p$ $\exists x \in S : p$
CHOOSE $x : p$ [Equals some x satisfying p]

Sets

$=$ \neq \in \notin \cup \cap \subseteq \setminus [set difference]
 $\{e_1, \dots, e_n\}$ [Set consisting of elements e_i]
 $\{x \in S : p\}$ [Set of elements x in S satisfying p]
 $\{e : x \in S\}$ [Set of elements e such that x in S]
SUBSET S [Set of subsets of S]
UNION S [Union of all elements of S]

Functions

$f[e]$ [Function application]
DOMAIN f [Domain of function f]
 $\{x \in S \mapsto e\}$ [Function f such that $f[x] = e$ for $x \in S$]
 $\{S \rightarrow T\}$ [Set of functions f with $f[x] \in T$ for $x \in S$]
 $\{f \text{ EXCEPT } ![e_1] = e_2\}$ [Function \hat{f} equal to f except $\hat{f}[e_1] = e_2$]
 $\{\{f \text{ EXCEPT } ![e] \in S\}\}$ [Set of functions \hat{f} equal to f except $\hat{f}[e] \in S$]

Records

$e.h$ [The h -component of record e]
 $\{h_1 \mapsto e_1, \dots, h_n \mapsto e_n\}$ [The record whose h_i component is e_i]
 $\{\{h_1 : S_1, \dots, h_n : S_n\}\}$ [Set of all records with h_i component in S_i]
 $\{r \text{ EXCEPT }!.h = e\}$ [Record \hat{r} equal to r except $\hat{r}.h = e$]
 $\{\{r \text{ EXCEPT }!.h \in S\}\}$ [Set of records \hat{r} equal to r except $\hat{r}.h \in S$]

Tuples

$e[i]$ [The i^{th} component of tuple e]
 $\langle e_1, \dots, e_n \rangle$ [The n -tuple whose i^{th} component is e_i]
 $S_1 \times \dots \times S_n$ [The set of all n -tuples with i^{th} component in S_i]

Strings and Numbers

" $c_1 \dots c_n$ " [A literal string of n characters]
STRING [The set of all strings]
 $d_1 \dots d_n$ $d_1 \dots d_n.d_{n+1} \dots d_m$ [Numbers]

Miscellaneous

if p **then** e_1 **else** e_2 [Equals e_1 if p true, else e_2]
case $p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n$ [Equals e_i if p_i true]
let $x_1 \triangleq e_1 \dots x_n \triangleq e_n$ **in** e [Equals e in the context of the definitions]
 $\wedge p_1$ [the conjunction $p_1 \wedge \dots \wedge p_n$] $\vee p_1$ [the disjunction $p_1 \vee \dots \vee p_n$]
 \dots \dots
 $\wedge p_n$ $\vee p_n$

Kuva 2: TLA⁺:n vakio-operaattorit. [Lam95]

case $p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n$ -lauseke saa lausekkeen e_i arvon, jos p_i on tosi. Lausekkeet e_i eivät saa olla totuusarvoisia. Case-lauseke on määritelty CHOOSE-operaattorin avulla:

$$\text{case } p_1 \rightarrow e_1, \dots, p_n \rightarrow e_n \triangleq \\ \text{CHOOSE } \$x : (p_1 \Rightarrow (\$x = e_1)) \wedge \dots \wedge (p_n \Rightarrow (\$x = e_n)),$$

jossa $\$x$ on jokin ennestään esittelemätön tunniste.

let $x_1 \triangleq e_1 \dots x_n \triangleq e_n$ **in** e -lauseke saa lausekkeen e arvon, määrittelyjen x_i ollessa voimassa vain lausekkeessa e . Tunnisteiden x_i täytyy olla esittelemättömiä siinä näkyvyysalueessa, jossa **let in** -lauseke on määritelty. Esimerkiksi lausekkeen

$$\text{let } x \triangleq 5 \\ y \triangleq 6 \\ F(n) \triangleq (n)^2 \\ \text{in } x * y + 9 + F(x) + F(y)$$

arvo on $5 * 6 + 9 + 5^2 + 6^2$.

Avainsana **definitions** aloittaa moduulissa funktioiden ja operaattorien määritelmät. Useimmiten on kuitenkin järkevää erottaa eri määritelmät toisistaan ja käyttää kuvaavampia nimiä kuten **actions**, **temporal** tai **predicates**.

Vaikka tyyliopasta ei olekaan saatavilla, on syytä pyrkiä yhtenäisyyden vuoksi käyttämään sellaisia merkintätapoja, joita muutkin käyttävät ja jotka parantavat spesifikaation luettavuutta. Esimerkiksi sulkuja ei tarvita, jos kirjoitetaan konjunktiot ja disjunktiot muodossa:

$$\begin{aligned} & \wedge \vee A \vee B \\ & \vee C \\ & \wedge \vee D \\ & \vee E \wedge F \\ & \vee G \end{aligned}$$

Esimerkki on yhtä pitävä lausekkeen $((A \vee B) \vee C) \wedge (D \vee (E \wedge F) \vee G)$ kanssa.

Kuten ohjelmointikielissäkin, spesifikaatiossa esiintyvien muuttujien ja operaattorien nimet on järkevää valita siten, että spesifikaatiota on helppo lukea ja ymmärtää. Esimerkiksi joukkojen nimissä voidaan käyttää yksiköllisiä muotoja, jolloin merkintä $a \in \text{Auto}$ voidaan lukea: "a on Auto."

4.5 Moduulit

TLA⁺:ssa moduulin määrittely aloitetaan vaakaviivalla (horizontal bar), johon kirjoitetaan moduulin nimi, ja lopetetaan toisella vaakaviivalla. Viivoilla ei ole

mitään semanttista merkitystä. Moduulissa voidaan käyttää muiden moduulien määritelmiä **import**- ja **include**-lauseiden avulla. Käytettäessä toisten moduulien määritelmiä on syytä huomata, että suunnatun graafin, jonka solmuina ovat moduulit ja kaarina **import**- ja **include**-lauseet, on oltava asyklinen.

import -lauseen avulla voidaan käyttää muiden moduulien parametreja, määritelmä, oletuksia ja teoreemoja.

include -lauseen avulla voidaan käyttää muiden moduulien määritelmiä. Käytönnotettavien moduulien parametrit korvataan lausekkeilla.

export -lauseen avulla voidaan kertoa, mitkä moduulin määritelmät näkyvät niihin moduuleihin, jotka ottavat sen käyttöönsä. Jos lausetta ei käytetä, näkyvät kaikki moduulissa annetut määritelmät ulospäin (alimoduulien tai käyttönnotettujen moduulien määritelmät eivät näy).

Merkittävin ero **import**- ja **include**-lauseissa on, että **import** sisällyttää oletukset käyttöön otetusta moduulista oletuksiksi, mutta **include** sisällyttää ne teoreemoiksi. Tällöin käyttöön otetun moduulin oletukset on oltava todistettavissa käyttävän moduulin oletuksista. Tätä ominaisuutta voidaan käyttää esimerkiksi silloin, kun halutaan määritellä moduuli, joka asettaa sitä käyttäville moduuleille vaatimuksia.

Eräitä usein käytettyjä moduuleja on määritelty valmiiksi. Tällaisia ovat esimerkiksi *Naturals*, *Reals*, *Sequences*, *RealTime* ja *FiniteSets*.

Moduuli *Naturals* määrittelee luonnollisten lukujen joukon *Nat* ja niille joukon operaattoreita yhteen-, vähennys- ja kertolaskua, potenssiinkorotusta sekä vertailuja varten. Moduulissa on määriteltynä myös operaattori *Numeral*.

Moduuli *Reals* määrittelee reaalityyppien joukon *Real*, sekä joukon operaattoreita mukaanlukien operaattorin *Decimal*. Moduuli *Sequences* määrittelee operaattorit sarjan pituutta ja katenointia varten ja operaattorin \dots , jonka avulla voidaan esimerkiksi kertoa, että $i..j$ on kokonaislujen joukko välillä i :stä j :hin.

Moduuli *RealTime* sisältää reaaliaikajärjestelmien spesifointiin tarvittavia määritelmiä, ja *FiniteSets* äärellisten joukkojen kuvaamiseen tarvittavia operaattoreita.

Spesifikaation ymmärrettävyyttä voi parantaa kommentoimalla se kattavasti. Kommentteja voi kirjoittaa mihin tahansa kohtaan moduulia, mutta ne on syytä kirjoittaa esimerkiksi käyttäen sellaista kirjasinlajia, joka erottaa ne muusta tekstistä. Moduuliin voi selvyuden vuoksi lisätä myös vaakaviivoja. Yleensä **parameter**-, **import**- ja **export**-lauseet edeltävät määritelmiä.

module <i>MusicalChairsParams</i>
parameters <i>Player, Chair</i> : CONSTANT <i>music, players, chairs</i> : VARIABLE
imports <i>Naturals, FiniteSets</i>
assumption <i>Finiteness</i> \triangleq $IsFiniteSet(Chair) \wedge IsFiniteSet(Player)$ <i>NumberOfChairs</i> \triangleq $Cardinality(Player) = 1 + Cardinality(Chair)$ <i>NonChairs</i> \triangleq $(\text{“Standing”} \notin Chair) \wedge (\text{“OutOfGame”} \notin Chair)$

Kuva 3: Moduuli *MusicalChairsParams*.

4.6 The Game of Musical Chairs

Musical Chairs on tuttu seuraleikki, johon pelaajien lisäksi tarvitaan musiikin lähde, esimerkiksi kasettinauhuri, ja tuoleja siten, että tuoleja on yksi vähemmän kuin pelaajia. Musiikin soidessa pelaajat kiertävät tuoleja ja musiikin tauottua jokainen pelaaja yrittää päästä istumaan. Pelaaja, jolle ei tuolia riittänyt, joutuu pelistä pois, muut pelaajat nousevat ylös, yksi tuoli poistetaan käytöstä ja musiikki alkaa jälleen soida. Peli loppuu kun vain yksi pelaaja on enää mukana. Seuraavassa esitetään peli spesifioituna TLA⁺:lla. Spesifikaatio on peräisin Lamportilta [Lam93], mutta sitä on osittain muutettu, jotta se vastaisi paremmin uutta syntaksia. Spesifikaation voi tietenkin kirjoittaa monella eri tavalla, mutta Lamportin tavoitteena on ollut tehdä tiukka spesifikaatio, joka ei niinkään kerro sitä kuinka peliä pitäisi pelata, vaan pikemminkin sen, mistä oikein pelattu peli koostuu.

Spesifikaatio rakentuu neljästä moduulista, joista ensimmäinen, *MusicalChairsParams*, on kuvassa 3. Moduulissa ilmoitetaan parametrit ja otetaan käyttöön kaksi valmiiksi määriteltyä moduulia, *Naturals* ja *FiniteSets*. Moduuli *FiniteSets* määrittelee totuusarvoisen funktion $IsFiniteSet(S)$, jonka arvo on tosi, jos S on äärellinen joukko ja funktion $Cardinality(S)$, joka palauttaa S :n alkioiden lukumäärän, jos S on äärellinen. Moduuli esittelee kaksi vakioparametria, *Player* ja *Chair*, jotka vastaavat kaikkien pelaajien ja kaikkien tuolien joukkoja. Muuttujaparametri *music* kuvaa musiikin tilaa, ja *chairs* on kullakin hetkellä käytössä olevien tuolien joukko. Muuttujaparametri *players* on funktio joka saa argumentikseen pelaajan ja sen arvo on tuoli, jolla pelaaja istuu.

Moduuli määrittelee lisäksi kolme oletusta. Oletus *Finiteness* väittää, että *Chair* ja *Player* ovat äärellisiä joukkoja, ja oletus *NumberOfChairs*, että pelaajia on yksi enemmän kuin tuoleja. Oletus *NonChairs* väittää, että merkkijonot “Standing” ja “OutOfGame” eivät kuulu tuolien joukkoon.

module <i>MusicalChairsPreds</i>
parameters <i>phase</i> : VARIABLE
imports <i>MusicalChairsParams</i>
predicates $Init \triangleq \wedge music = \text{“On”}$ $\wedge players = [p \in Player \mapsto \text{“Standing”}]$ $\wedge chairs = Chair$ $\wedge phase = \text{“Walking”}$ $TypeOK \triangleq \wedge music \in \{\text{“On”}, \text{“Off”}\}$ $\wedge players \in [Player \rightarrow chairs \cup \{\text{“Standing”}, \text{“OutOfGame”}\}]$ $\wedge chairs \subseteq Chair$ $\wedge phase \in \{\text{“Walking”}, \text{“Sitting”}, \text{“Rising”}, \text{“Ready”}\}$

Kuva 4: Moduuli *MusicalChairsPreds*.

Toisessa moduulissa *MusicalChairsPreds* (kuva 4) otetaan käyttöön moduuli *MusicalChairsParams*. Moduulin parametri on muuttuja *phase*, joka kertoo, missä vaiheessa peli on. Peli voi olla jossakin neljästä vaiheesta, jotka ovat “Walking”, “Sitting”, “Rising” ja “Ready”. Lisäksi määritellään kaksi predikaattia. *Init* on tosi, jos *players* on funktio ja kaikki pelaajat seisovat pelin alkaessa, musiikki soi, kaikki tuolit ovat käytössä ja vaihe on “Walking”. *TypeOK* on tosi, jos kaikki muuttujat ovat oikeaa tyyppiä. Se väittää, että musiikki voi olla joko päällä tai poissa päältä ja että *chairs* on *Chair*-joukon osajoukko ja että peli on aina jossakin neljästä vaiheesta. Lisäksi väitetään, että kaikilla *Player* joukon alkioilla *p*, *player(p)* on joko tuoli, jolla pelaaja istuu, tai merkijono “Standing” tai “OutOfGame”. Jos pelaaja seisoo, kuvaa *player* pelaajan merkijonoksi “Standing” ja merkijonoksi “OutOfGame”, jos pelaaja on pois pelistä.

Kolmannessa moduulissa *MusicalChairsActions* (kuva 5) otetaan käyttöön moduuli *MusicalChairsParams* ja esitellään aktiot. Moduulin parametrina on *phase*.

Aktio *MusicOff* vaientaa musiikin kun peli on tilassa “Walking” ja muuttaa tilaksi “Sitting” eli siirtää pelin istumisvaiheeseen.

SitDown(p,c) vastaa pelaajan *p* istumista tuoliin *c*. Se voi tapahtua, jos peli on istumisvaiheessa, *p* seisoo ja *c* on vapaana.

LoserOut(p) vastaa tapahtumaa, jossa pelaaja *p* joutuu pelistä pois, jäätyään seisomaan vapaiden tuolien loputtua. Aktio voi tapahtua, kun peli on istumisvaiheessa, *p* seisoo ja kaikilla pelissä mukana olevilla tuoleilla istuu joku. Aktio siirtää pelin nousemisvaiheeseen.

ChairsActions. Niissä on parametrina *phase*, joten siinä näkyvyysalueessa, jossa moduulit otetaan käyttöön, täytyy muuttujan *phase* olla esiteltynä. Tätä varten määritellään alimoduuli, jonka parametrina muuttuja *phase* on. Alimoduuli ottaa moduulit käyttöön **include**-lauseella nimillä *Pred* ja *Act* ja määrittelee spesifikaation *ISpec* kanonisessa muodossa, jossa *Next* on disjunktio aktioista ja *v* on muuttujien tupla. Kanonisessa muodossa määritellään järjestelmän alkutila, mahdolliset askeleet ja vaaditaan heikko reiluus *Next*:iä kohtaan. Alimoduulin teoreema *TypeCorrectness* väittää, että *ISpec* on tyyppien osalta voimassa.

Määritelmä *Spec* on spesifikaatio, josta apumuuttuja *phase* on piilotettu. Predikaatti *Done* on tosi, jos peli on päättynyt eli yksi pelaajista istuu ja muut pelaajat ovat poissa pelistä. Moduulin lopuksi esitetään teoreema, joka väittää, että aina kun *Spec* on tosi, niin jostakin tilasta lähtien *Done* on aina tosi.

4.7 TLA⁺ spesifointimenetelmänä

TLA⁺ on selvästi matemaatikon työkalu. Menetelmä tarjoaa määrittelijälle, joka tuntee TLA:n, mahdollisuuden kirjoittaa selkeän struktuurin omaavia, laajoja spesifikaatioita. Logiikkaan ja matemaattisiin merkintöihin harjaantumaton määrittelijä voi sitä vastoin kokea menetelmän luotaantyöntäväksi.

Jotta TLA⁺ olisi käyttökelpoinen spesifointimenetelmä, on sen syntaksin vaikiinnuttava. Menetelmä kaipaa tuekseen myös työkaluja spesifikaation simulointiin, oikeaksi todistamiseen ja toteuttamiseen. Tässä muodossa TLA⁺ ei ole sellainen menetelmä, joka saisi esimerkiksi teollisuuden laajemmassa mittakaavassa kiinnostumaan formaaleista menetelmistä. Menetelmän käyttöönotto vaatii teoreettisen suuntautuneisuuden lisäksi aimo annoksen akateemista mielenkiintoa.

TLA⁺ on tervetullut sellaisiin sovelluksiin, joissa on tähän asti käytetty TLA:ta. Sillä on etunaan TLA:n semantiikka, joka tekee siitä formaalin sanan kaikessa merkityksessä. Sen avulla voidaan tehdä yksiselitteisiä ja tarkkoja spesifikaatioita, mikä saattaa kyllä vaatia hikeä ja kyyneliä.

Musical Chairs -pelin spesifikaatiot on kirjoitettu käyttäen apuna Lamportin L^AT_EX-tyylitiedostoa, joka auttaa kirjoittamaan moduulit syntaktisesti oikein. Tyylitiedosto on saatavilla TLA:n kotisivulta World Wide Webistä osoitteesta <http://www.research.digital.com/SRC/tla/>. TLA⁺-spesifikaation kirjoittaminen jollakin muulla tavalla ei ole realistinen vaihtoehto.

5 DisCo

5.1 Menetelmä

DisCo (Distributed Cooperation) on Tampereen teknillisessä korkeakoulussa kehitetty reaktiivisten järjestelmien spesifointimenetelmä, joka perustuu yhteisaktioteoriaan. Määrittelijän näkökulmasta katsottuna menetelmän tärkeimmät osat ovat DisCo-kieli [Jär92] ja DisCo-työkalu [Sys91].

DisCo-kieli on olioperustainen ja suoritettava, ja se muistuttaa syntaksiltaan ohjelmointikieltä. Suoritettavuudella tarkoitetaan tässä yhteydessä sitä, että spesifikaation kuvaaman järjestelmän toimintaa voidaan simuloida. Kielen semantiikka on määritelty TLA:lla [JärKur90, Jär92].

Jokainen DisCo-spesifikaatio on muunnettavissa TLA:ksi, mikä mahdollistaa todistamisen käyttäen TLA:ta. TLA onkin DisCon kannalta se voima, joka tekee todistukset mahdollisiksi. DisCo-spesifikaatiot voidaan esittää TLA:n kanonisessa muodossa, joten DisCo tarjoaa ohjelmointiin perehtyneelle helposti omaksuttavan ympäristön TLA-spesifikaatioiden laatimiseen. Vaikka vastaava spesifikaatio periaatteessa olisikin laadittavissa suoraan TLA:lla, logiikan kieli on kuitenkin vieras useille spesifikaation laatijoille.

DisCo tukee sekä ylhäältä alas että alhaalta ylös etenevää spesifointia. Useimmiten edetään ylhäältä alas, jolloin voidaan hyödyntää superpositiota, joka on DisCon tapa toteuttaa vaiheittainen tarkentaminen. Spesifioitavan järjestelmän toimintaa voidaan kaikissa tarkennusvaiheissa simuloida DisCo-työkalun avulla, joka tukee graafista visualisointia ja animointia.

5.2 Yhteisaktioteoria

Perinteisesti rinnakkaisten järjestelmien formaaliin spesifointiin tarkoitetut menetelmät ovat pohjautuneet keskenään kommunikoiviin prosesseihin. DisCo perustuu Backin ja Kurki-Suonion kehittämään yhteisaktioteoriaan [BacKur83, BacKur84, BacKur88a, BacKur88b], jossa kommunikaatio prosessien välillä tapahtuu *aktioissa*. Samantapainen suoritussmalli on käytössä Unity-menetelmässä [ChaMis88].

Yhteisaktio-paradigma on osoittautunut toimivaksi reaktiivisten ja rinnakkaisien järjestelmien suunnittelussa ja niistä pääteltäessä. Sen sijaan, että järjestelmän toimintaa tarkasteltaisiin yksittäisten prosessien kannalta, tarkastellaankin aktioita, joissa voi olla yksi tai useampi *osallistuja*. Prosessikäsitettä ei tarvita ja kommunikointi tapahtuu aktioissa implisiittisesti. Järvinen [Jär92] mainitsee kolme yhteisaktio-paradigman etua verrattuna prosessi-paradigmaan: päättely on helpompaa ja oikeellisuus on siten helpompi todentaa, tarkentamisen vaihtokukset on helpompi kohdistaa, ja systeemin tila on kokonaan muuttujissa. Aktiot voidaan myös ilmaista täysin samoin ja symmetrisesti, olipa osallistujia yksi tai kymmenen. [Jär92]

Järjestelmää toteutettaessa aktioissa tapahtuva prosessointi joudutaan yleensä jakamaan keskenään kommunikoiville prosesseille. Määrittelyvaiheessa halutaan kuvata järjestelmän toiminta korkeammalla abstraktiotasolla ja jako prosesseiksi katsotaankin osaksi aktiosysteemin toteutusta. [Jär92]

5.3 Objektit

DisCossa kaikki spesifikaation muuttujat on kapseloitu olioihin, joita kutsutaan objekteiksi. DisCo-kielessä on muiden oliokielten tapaan luokat, oliot ja perintä, mutta metodit on korvattu aktioilla. Aktiot eroavat metodeista siinä, että aktioissa voi olla useampikin kuin vain yksi osallistuja [Jär92]. Varsinaisia tyyppejä DisCossa ovat vain **boolean**- ja **integer**-tyypit. Niiden lisäksi käytössä on myös joukko- ja sekvenssi-tyypit.

Aktiossa jokaiselle osallistujille annetaan *rooli*, jonka perusteella osallistujiin viitataan. Objektit ovat systeemin kannalta tietorakenteita, joilla on kyky osallistua tiettyihin aktioihin tietyissä rooleissa. Objektit voidaan toteutuksessa kuvata joko prosesseiksi tai passiivisiksi tietorakenteiksi ja aktiot prosessien väliseksi kommunikoinniksi [Mik92].

Luokkamäärittelyssä kerrotaan ne muuttujat, joita luokan objekteilla on. Tilakone on muuttujan erikoistapaus, jota käytetään objektin tilan kuvaamiseen. Oletustila ilmaistaan lisäämällä tilan nimen eteen asteriski (*). Tilaan voi liittyä myös parametreja ja siihen voi lisätä alitiloja **extend**-lauseen avulla. Koko systeemin tila koostuu yksittäisten objektien tiloista.

Esimerkiksi jokaisella Player-luokan objektilla on tilakone, jonka tilat ovat Playing ja OutOfGame, joista Playing on oletustila:

```
system PlayersAndChairs is
  class Player is
    state *Playing, OutOfGame;
    extend Playing by
      state *Standing, Sitting(Chr: Chair);
    end Playing;
  end Player;
```

Tilassa Playing on kaksi alitilaa, Standing ja Sitting. Sitting-tilan parametrina on muuttuja Chr, joka on viite Chair-luokan objektiin. Jokaisella Chair-luokan objektilla on tilakone, jonka tilat ovat InUse ja Removed. Oletustilassa InUse on kaksi alitilaa, Free ja Occupied:

```
class Chair is
  state *InUse, Removed;
  extend InUse by
    state *Free, Occupied;
  end InUse;
end Chair;
```

Aktiossa kerrotaan roolit ja ne luokat, joiden objektit voivat osallistua aktioon näissä rooleissa. Lisäksi annetaan vahti, jonka ollessa tosi aktio voi tapahtua ja runko, jossa kerrotaan mitä aktiossa tapahtuu. Jokainen osallistuja voi aktion tapahtuessa olla vain yhdessä roolissa. Aktiossa voi olla myös parametreja, joiden arvot määräytyvät epädeterministisesti aktion tapahtuessa.

Aktion runko koostuu sijoituslauseista, ehtolauseista, tilasiirtymistä ja paikallisista assertioista. Rungossa voidaan viitata vain aktion parametreihin ja osallistujiin. Assertio on totuusarvoinen lauseke, joka väittää että jokin turvallisuusvaatimus on voimassa. Aktion paikallisen assertion tulee olla tosi, kun suoritus tulee sen kohdalle.

Esimerkiksi aktioon SitDown voi osallistua yksi Player-luokan objekti roolissa P ja yksi Chair-luokan objekti roolissa C:

```
action SitDown by P:Player; C:Chair is  
when P.Standing and C.Free do  
    -> P.Sitting(C);  
    -> C.Occupied;  
end SitDown;  
end PlayersAndChairs;
```

Aktio voi tapahtua vain jos P on tilassa Standing ja C tilassa Free. Aktion tapahtuessa P siirtyy tilaan Sitting, C asetetaan viittaamaan C:hen ja C siirtyy tilaan Occupied.

5.4 Spesifikaation rakenne

DisCo-spesifikaatio koostuu systeemi- ja luontiosista. Systeemiosia on kokoelma luokkia, funktioita ja aktioita sekä globaaleja assertioita ja alkuehtoja (initial conditions). Systeemiosassa voidaan tuoda aiemmin määriteltyjä systeemejä uuden systeemin osaksi *importoimalla*. Importoidun systeemin aktioita ja luokkia voidaan *tarkentaa*.

DisCon funktioissa ei saa olla silmukoita tai rekursiota, eikä niillä ole sivuvaikutuksia. Niiden parametrit ja paluuarvo voivat olla mitä tyyppiä tahansa.

Aktion paikallisten assertioiden lisäksi on mahdollista sijoittaa assertioita samalle tasolle luokkien ja aktioiden kanssa, jolloin niitä kutsutaan globaaleiksi assertioiksi, tai luokkamäärittelyn sisälle, jolloin puhutaan luokan paikallisista assertioista. Sekä globaalien että luokan paikallisten assertioiden tulee olla tosia aina, paitsi kesken aktion suorituksen [Aal96].

Globaalit alkuehdot ovat assertioita, joiden pitää olla tosia heti luomisen jälkeen. Luokan sisällä olevat paikalliset alkuehdot liittyvät johonkin tilaan ja niiden pitää olla tosia kun tila, johon ne liittyvät, tulee aktiiviseksi.

Koska DisCo-spesifikaation pitää olla simuloitava, täytyy määrittelijän antaa

systemin alku tila. Tämä tapahtuu spesifikaation luontiosassa, jossa luodaan objektit ja alustetaan niiden muuttujat. Objekteja on mahdollista luoda äärettömän määrä, mutta äärettömyyden mallintaminen on ongelmallista simuloinnin kannalta.

DisCossa systeemit ovat suljettuja, eli ne kuvaavat myös ympäristön toiminnan. Ympäristö ja sen tapahtumat mallinnetaan objekteilla ja aktioilla kuten itse järjestelmäkkin. [Mik95]

Vaikka DisCo-kieli onkin varsin helppolukuista, spesifikaatio on syytä kommentoida kattavasti. Ohjelmointikielten tapaan kommentteja voi DisCossa kirjoittaa koodin joukkoon. Kommentti aloitetaan merkkiyhdistelmällä "--" ja se päättyy rivin loppuun.

5.5 Superpositio

DisCo-spesifikaatiota tarkennetaan vaiheittain käyttäen superpositiota. Mekanismit säilyttää turvallisuusominaisuudet, mikä rajoittaa importoidulle systemille tehtäviä muutoksia. Superpositiossa laajennetaan systemin tilaa ja muutetaan aktioita siten, että ne voivat tapahtua laajennetuissa tiloissa. Tämä tapahtuu importoimalla valmis systemi ja tarkentamalla sen luokkia ja aktioita. Luokkia tarkennetaan käyttäen **extend**-lausetta ja aktioita käyttäen **refined**-lausetta. Turvallisuusominaisuuksien säilyminen varmistetaan kieltämällä sellaisten lauseiden lisääminen, jotka muuttavat importoidussa systemissä esiteltyjä muuttujia tai tiloja.

Esimerkiksi systemi `PlayersChairsAndState` importoi systemin `PlayersAndChairs`:

```
system PlayersChairsAndState import PlayersAndChairs; is
  class Phase is
    state *Walking, Sitting, Rising, Ready;
  end Phase;

  refined SitDown by ... Ph:Phase is
  when ... Ph.Sitting do
    ...
  end SitDown;
end PlayersChairsAndState;
```

Systemissä esitellään luokka `Phase`, jolla on nelitilainen tilakone ja tarkennetaan aktiota `SitDown` siten, että sillä on `Phase`-luokan objekti osallistujana roolissa `Ph`. Aktio vaihtaa tiukennetaan siten, että aktio voi tapahtua vain jos `Ph` on tilassa `Sitting`.

Systemi `MusicalChairs` importoi systemin `PlayersChairsAndState` ja asettaa assertion `Done`, joka on epätosin silloin kun vain yksi pelaaja on enää mukana pelissä:

```

system MusicalChairs import PlayersChairsAndState; is
    assert Done is not (or/P:Player:: P.Playing and      -- Only one player
        and/Q:Player:: (Q.OutOfGame or Q = P));      -- left
end MusicalChairs;

```

Luontiosa Game määrittelee alkutilan systeemille MusicalChairs:

```

creation Game of MusicalChairs is
    P : Player;
    for i in 1..6 loop
        P := new Player;
    end loop;

    C : Chair;
    for i in 1..5 loop
        C := new Chair;
    end loop;

    Ph : Phase;
    Ph := new Phase;
end Game;

```

Alkutilassa on kuusi Player-luokan objektia ja viisi Chair-luokan objektia sekä yksi Phase-luokan objekti Ph. Vaikka tässä esimerkissä luotaville objekteille annetaan samat nimet kuin aktion rooleille, se ei ole mitenkään välttämätöntä. DisCo ei myöskään erota isoja ja pieniä kirjaimia toisistaan. Esimerkeissä niitä on käytetty vain ymmärtämisen helpottamiseksi.

Superpositiossa elävyysominaisuudet eivät välttämättä säily. Sitä tärkeämpää on kuitenkin se, että turvallisuusominaisuudet säilyvät, sillä yleensä halutaan mieluummin järjestelmä, joka ei tee mitään kuin järjestelmä, joka toimii virheellisesti. [Setä92]

Vaiheittainen tarkentaminen on tärkein mekanismi rakennettaessa spesifikaatiota ylhäältä alas. Superpositiota käytettäessä lasketaan spesifikaation abstraktiotasoa luottaen siihen, että turvallisuusominaisuudet säilyvät. Valmista spesifikaatiota on turha lähteä pilkkomaan tarkennusaskeliin.

5.6 Strukturoinnista

DisCo-kielen syntaksi on haluttu tehdä mahdollisimman helppolukuiseksi. Kielen suunnittelussa on täytynyt huomioida myös suoritettavuus ja animoitavuus, koska DisCo-työkalu on alusta lähtien ollut oleellinen osa menetelmää. Lausekielistä DisCo muistuttaa lähinnä Ada-kieltä. Se tarjoaa määrittelijälle useita strukturointimahdollisuuksia sekä ylhäältä alas että alhaalta ylös rakennettavaan spesifikaatioon.

Kielen suunnittelussa yksi tärkeimmistä päämääristä oli modulaarisuuden tuominen aktio-systeemeihin. Modulaarisuus tukee spesifikaatioiden uudelleenkäyttöä ja uudelleenkäyttöä varten lisätyt ominaisuudet tukevat modulaarisuutta. Tällaisia ominaisuuksia DisCossa ovat systeemien yhdistäminen (composition), perintä ja **stuff**-tyypit, jotka ovat kaikki käyttökelpoisia kun spesifikaatiota rakennetaan alhaalta ylös. Systeemien yhdistäminen ja **stuff**-tyypit ovat tärkeitä myös edettäessä ylhäältä alas. [Jär92]

Systeemit yhdistetään importoimalla ne samaan systeemiin. Koska importoitujen systeemien turvallisuusominaisuuksien tulee olla edelleen voimassa, itsenäisiä aktioita joudutaan yhdistämään. Jos aktioilla on yhteinen alkuperä eli jos ne on alunperin tarkennettu samasta aktiosta, ne yhdistetään implisiittisesti ja niiden tapahtuminen itsenäisesti estetään. Uudella aktiolla on kaikki vanhojen aktioiden parametrit ja osallistajat, sen vahti on vanhojen aktioiden vahtien konjunktio ja sen runko sisältää kaikki vanhojen aktioiden runkojen lauseet. Aktioita voidaan yhdistää myös eksplisiittisesti. [Jär92]

Perintä tukee modulaarisuutta ja mahdollistaa uudelleenkäytön, joka taas mahdollistaa yleiskäyttöisten komponenttien kirjastoimisen. DisCon perintämekanismi poikkeaa perinteisesti oliokielissä käytetyistä mekanismeista. DisCon perintä on oikeastaan aggregointia, jossa perivä luokka sisältää perityn luokan. Siten on esimerkiksi mahdollista sisällyttää perittävä luokka kahteen kertaan perivään luokkaan tai vaikkapa sen alitilaan.

DisCon **stuff**-tyyppi mahdollistaa abstraktiotason nostamisen siten, että muutujan todellinen tyyppi määritellään vasta tarkennusvaiheessa. Ominaisuudella on käyttöä esimerkiksi tietoliikenneprotokollan spesifoinnissa, kun samalle datapaketilte voidaan eri tarkennuksissa antaa eri sisältö.

DisCo-kielessä moduulilla tarkoitetaan systeemiä, jonka aktioiden ja luokkien näkyvyyttä ulkopuolelle on rajoitettu. Tämä tapahtuu korvaamalla avainsana **system** avainsanalla **module** ja määrittelemällä **export**-listan avulla ne aktiot, luokat ja **stuff**-tyypit, joiden halutaan näkyvän ulkopuolelle. Koska samassa moduulissa voi määritellä useita erinimisiä **export**-listoja, voi importoivassa systeemissä listan nimen perusteella valita minkä osan importoidusta moduulista haluaa käyttöön.

DisCo-kielessä on myös sellaisia lausekkeita, joita ei ole ohjelmointikielissä. Niitä ovat esimerkiksi kvanttorit. DisCossa ainoastaan luokkien yli voi kvantifioida. Logiikasta tuttujen kaikki- (DisCossa **and**/) ja olemassaolokvanttorin (**or**/) lisäksi DisCossa on käytössä myös summa- (+/), tulo- (* /), minimi- (**min**/) ja maksimikvanttorit (**max**/).

5.7 Musical Chairs DisColla

Kohdassa 4.6 annettiin Musical Chairs -pelin spesifikaatio TLA⁺:lla. DisCo-kielinen spesifikaatio samasta pelistä on esiteltyä kuvissa 7, 8 ja 9.

```

system PlayersAndChairs is
  class Player is
    state *Playing, OutOfGame;
    extend Playing by
      state *Standing, Sitting(Chr: Chair);
    end Playing;
  end Player;

  class Chair is
    state *InUse, Removed;
    extend InUse by
      state *Free, Occupied;
    end InUse;
  end Chair;

  class Music is
    state *On, Off;
  end Music;

  action MusicOff by M:Music is
  when TRUE do
    -> M.Off;
  end MusicOff;

  action SitDown by P:Player; C:Chair is
  when P.Standing and C.Free do
    -> P.Sitting(C);
    -> C.Occupied;
  end SitDown;

  action LoserOut by P:Player is
  when P.Standing and (and/C:Chair:: not C.Free) do
    -> P.OutOfGame;
  end LoserOut;

  action GetUp by P:Player; C:Chair is
  when P.Sitting.Chr = C and (+/ B:Chair | B.InUse ::1) > 1 do
    -> P.Standing;
    -> C.Free;
  end GetUp;

  action RemoveChair by C:Chair is
  when (and/B:Chair:: not B.Occupied) and C.Free do
    -> C.Removed;
  end RemoveChair;

  action MusicOn by M:Music is
  when TRUE do
    -> M.On;
  end MusicOn;
end PlayersAndChairs;

```

Kuva 7: Systeemi PlayersAndChairs.

```

system PlayersChairsAndState import PlayersAndChairs; is
  class Phase is
    state *Walking, Sitting, Rising, Ready;
  end Phase;

  refined MusicOff by ... Ph:Phase is
  when ... Ph.Walking do
    ...
    -> Ph.Sitting;
  end MusicOff;

  refined SitDown by ... Ph:Phase is
  when ... Ph.Sitting do
    ...
  end SitDown;

  refined LoserOut by ... Ph:Phase is
  when ... Ph.Sitting do
    ...
    -> Ph.Rising;
  end LoserOut;

  refined GetUp by ... Ph:Phase is
  when ... Ph.Rising do
    ...
  end GetUp;

  refined RemoveChair by ... Ph:Phase is
  when ... Ph.Rising do
    ...
    -> Ph.Ready;
  end RemoveChair;

  refined MusicOn by ... Ph:Phase is
  when ... Ph.Ready do
    ...
    -> Ph.Walking;
  end MusicOn;
end PlayersChairsAndState;

```

Kuva 8: Systeemi PlayersChairsAndState.

Kyseinen TLA⁺-spesifikaatio on lähes suoraan muunnettavissa DisCo-kielelle. Suurin ero on, että pelaajan ja sen tuolin, jolla hän istuu, välinen relaatio ilmaistaan pelaajan Sitting-tilassa olevalla parametrilla Chr, joka on viite tuoliin. Yleisessä tapauksessa muunnos ei ole näin yksinkertainen.

Spesifikaatio on rakennettu käyttäen superpositiota kahdessa tarkennusaskellessa. Ensimmäisessä systeemissä PlayersAndChairs (kuva 7) määritellään luokat Player, Chair ja Music ja aktiot MusicOff, SitDown, LoserOut, GetUp, RemoveChair ja MusicOn.

Systeemissä PlayersChairsAndState (kuva 8) määritellään luokka Phase ja tarkennetaan ensimmäisen systeemin aktioita. Kaikkiin aktioihin lisätään luokan

```

system MusicalChairs import PlayersChairsAndState; is
  initially Init is (or/ M:Music:: M.On) and
    (and/ P:Player:: P.Standing) and
    (and/ C:Chair:: C.InUse) and
    (or/ Ph:Phase:: Ph.Walking);

  initially NumberOfChairs is (+/ P:Player ::1) = 1 + (+/ C:Chair ::1);

  assert Done is not (or/P:Player:: P.Playing and      -- Only one player
    and/Q:Player:: (Q.OutOfGame or Q = P));      -- left
end MusicalChairs;

creation Game of MusicalChairs is
  M : Music;
  M := new Music;

  P : Player;
  for i in 1..6 loop
    P := new Player;
  end loop;

  C : Chair;
  for i in 1..5 loop
    C := new Chair;
  end loop;

  Ph : Phase;
  Ph := new Phase;
end Game;

```

Kuva 9: Systemi MusicalChairs ja sen luontiosa Game.

Phase objekti osallistujaksi roolissa Ph ja tiukennetaan vahteja siten, että aktiot voivat tapahtua vain pelin ollessa tietyssä vaiheessa. Aktiot, jotka muuttavat pelin vaihetta, siirtävät Ph:n tilakoneen uutta vaihetta vastaavaan tilaan.

Kolmannessa systeemissä MusicalChairs (kuva 9) tarkennetaan toista systeemiä asettamalla globaalit alkuehdot Init ja NumberOfChairs sekä globaali assertio Done. Ne vastaavat samannimisiä määritelmiä TLA⁺-spesifikaatiossa. NumberOfChairs on toteutettu käyttäen DisCon summakvanttoria ja Init sekä Done käyttäen olomassaolo- ja kaikkikvanttoreita.

Oletusta Finiteness ei ole sisällytetty spesifikaatioon, koska DisCossa ei ole mahdollista kuvata äärettömyyttä kaikissa tapauksissa, vaikka objekteja on luontiosassa mahdollista luoda ääretön määrä. TLA⁺-spesifikaation määritelmät *NonChairs* ja *TypeOK* jätettiin pois tarpeettomina.

5.8 Peräkkäinen suoritusmalli

DisCon suoritusmalli on aktio-orientoitunut. Aktiot ovat atomisia ja niitä suoritetaan peräkkäisesti. Jokaisella aktiolla on osallistujat, vahti ja runko. Aktion

sanotaan olevan *vireessä*, jos on löydettävissä sopivat osallistujat siten, että vahti on tosi. Aktion suoritus tarkoittaa sen rungossa annettujen lauseiden suorittamista.

DisCo-spesifikaation luontiosa määrittelee systeemin alkutilan. Alkutilan jälkeen suoritus etenee kun valitaan epädeterministisesti jokin vireessä oleva aktio, suoritetaan se, valitaan uudestaan jne.

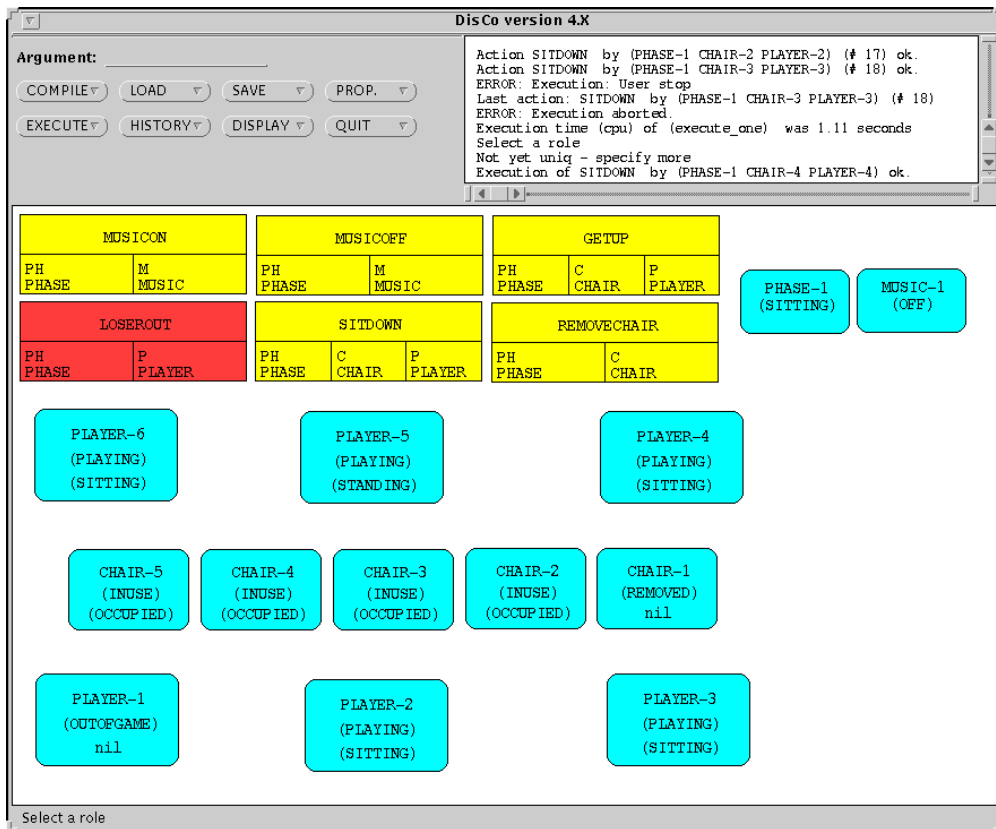
Rinnakkaisuuden mallintaminen peräkkäisyydellä vaatii hieman perusteluja. On lopputuloksen kannalta sama, suoritetaanko riippumattomia aktioita rinnakkaisesti vai peräkkäisesti [Kur96b] ja peräkkäistä järjestystä voidaan aina pitää vain järjestyksenä, jossa aktioiden on havaittu tapahtuvan. Lisäksi voidaan aina keskustella siitä, alkaako kaksi aktiota todella tapahtua samanaikaisesti vain onko niiden välillä millisekunnin murto-osan ero [Lam93]. Se, että yhden suorituksen aikana aktiot tapahtuvat tietyssä järjestyksessä ei tietenkään kerro mitään siitä, missä järjestyksessä ne tapahtuvat jonkin muun suorituksen aikana.

Äärettömän pitkään jatkuvassa suorituksessa on otettava huomioon reiluusvaatimukset. DisCossa aktioilla ei ole mitään sisäänrakennettuja reiluusvaatimuksia, vaan jokainen vireessä oleva aktio voi tapahtua, mutta minkään action ei ole pakko tapahtua. Reiluus pitää eksplisiittisesti lisätä rooleihin aktioissa. Tällaisen roolin saanutta osallistujaa sanotaan reiluksi osallistujaksi. DisCon reiluus määrittellään seuraavasti: jos aktio on äärettömän usein vireessä siten, että sama objekti voisi osallistua siihen samana reiluna osallistujana, action täytyy tapahtua äärettömän usein tämän objektin ollessa sinä reiluna osallistujana. Jos reiluus vaaditaan useammalle kuin yhdelle osallistujalle, yllä annettu reilusehto on pädeittävä myös jokaiselle reilujen osallistujien yhdistelmälle. Aktion reilut osallistujat erotetaan muista merkitsemällä asteriskit niiden nimien eteen. [Jär92].

5.9 Työkalu

DisCo-työkalu mahdollistaa DisCo-spesifikaatioiden simuloinnin. Se visualisoi spesifikaation graafisesti ja animoi sen suorituksen. Kuvassa 10 on animoituna Musical Chairs -spesifikaatio. Työkalu luo animointinäytöt automaattisesti ja antaa käyttäjälle mahdollisuuden animoinnin graafisten ominaisuuksien muuttamiseen. Käyttäjä voi valita tarkasteltavaksi minkä tahansa niistä tarkennusvaiheista, joita vastaaville systeemeille on annettu luontiosa. Myös suoritushistorian havainnollistaminen graafisena skenaariona on mahdollista.

DisCo-työkalu antaa mahdollisuuden testata spesifikaation toimivuutta. Koska animaatiosta on helpompi saada palautetta kuin kirjallisessa muodossa olevasta spesifikaatiosta, työkalu auttaa myös kommunikoinnissa määrittelijän ja sovel-lusalueen asiantuntijoiden välillä. Spesifikaation rakenne ja sisäiset kytkennät ovat lisäksi helpommin ymmärrettävissä. DisCo-työkalu helpottaa myös tutustumista DisCo-menetelmään ja -kieleen. [Sys95]



Kuva 10: DisCo-työkalu, johon on ladattu Musical Chairs -spesifikaatio.

Työkalu antaa mahdollisuuden valita suoritettavat aktiot ja niiden osallistujat manuaalisesti, mutta se voi suorittaa spesifikaatiota myös automaattisesti. Automaattisessa suorituksessa käytetään satunnaislukugeneraattoria epädeterministisen valinnan mallintamiseen.

Simulointi helpottaa vaikeasti havaittavien virheiden löytämistä. Työkalu mahdollistaa mallilla leikkimisen, mikä yleensä johtaa sellaiseen suoritukseen, jota ei yleensä ensimmäisenä tule ajatelleeksi. Työkalun manuaalisella suorituksella voidaan käydä läpi kiinnostavia suorituspolkuja, mutta kattavaa testausta tällä ei luonnollisesti kuitenkaan saada aikaan [Sys94]. Työkalu ilmoittaa käyttäjälle jos jotakin alkuehtoa tai assertiota rikotaan. DisCo-työkalu ei simuloidessaan ota huomioon spesifikaation reiluusvaatimuksia, koska suoritukset ovat äärellisen pituisia.

5.10 Todistamisesta

Yksi tärkeimmistä syistä käyttää formaalia spesifointimenetelmää on, että spesifikaation virheettömydestä voidaan vakuuttautua tekemällä todistuksia. Jo-

kainen DisCo-spesifikaatio on muunnettavissa TLA:ksi, mikä mahdollistaa todistusten tekemisen TLA:lla.

Edettäessä ylhäältä alas todistetaan aina uuden askeleen ominaisuudet. Turvallisuusominaisuuksien todistamista helpottaa se, että aikaisempien systeemien turvallisuusominaisuudet ovat voimassa. Jos taas edetään alhaalta ylös, todistetaan jokaisen osan turvallisuusominaisuudet erikseen. Lopullisen järjestelmän turvallisuusominaisuudet ovat sen osien turvallisuusominaisuuksien unioni. [Setä92]

Turvallisuus- ja elävyysominaisuudet todistetaan yleensä erikseen. Elävyysominaisuuksien todistaminen voi olla huomattavasti vaikeampaa kuin turvallisuusominaisuuksien. [Lad96]

Usein todistuksista tulee pitkiä ja työläitä, jolloin virheiden tekemisen todennäköisyys kasvaa. Todistusten tekeminen käsin on usein turhaa, sillä vain harvat määrittelijät pystyvät viemään todistuksia läpi riittävällä tarkkuudella. Tästä syystä on mielekästä pyrkiä automatisoimaan todistuksia mahdollisimman paljon. Mekaanisen teoreemantodistimen kehittäminen DisCoon onkin aloitettu [Kel95]. Vaikka todistaminen tulisikin automaattiseksi, se ei poista tarvetta tehdä spesifikaatiosta informaaleja päättelyitä, vaan nämä kaksi täydentävät toisiaan [Kur96a].

6 DisCo-spesifikaation muuntaminen TLA⁺:ksi

6.1 Muunnos DisCosta TLA⁺:ksi

Järvinen [Jär92] on esittänyt muunnoksen, jonka avulla jokainen DisCo-spesifikaatio voidaan muuntaa TLA:ksi. Muunnos antaa DisCo-kielelle formaalin semantiikan ja mahdollistaa todistamisen TLA:lla.

Koska TLA⁺ ei rajoita TLA-lausekkeiden käyttöä spesifikaatioissa, voidaan DisCo-spesifikaatiot muuntaa TLA⁺-spesifikaatioiksi. Muunnoksen tekemistä voi perustella esimerkiksi sillä, että muuntaminen edellyttää, todistamisen tapaan, spesifikaation yksityiskohtaista ymmärtämistä, ja antaa näin mahdollisuuden nähdä spesifikaatio uudessa valossa.

Järvisen esittämän muunnoksen avulla voidaan jokainen DisCo-spesifikaatio muuntaa mekaanisesti TLA:ksi. Mikäli yksittäinen spesifikaatio muunnetaan käsin, ei muunnosta kannata seurata orjallisesti, vaan kannattaa käyttää hyväksi TLA⁺:n tarjoamia mahdollisuuksia mm. määrittelemällä tietorakenteita ja moduuleja.

6.2 Hajautettu lajittelu DisColla

Hajautettu lajittelu oli yksi ensimmäisistä ongelmista, johon yhteisaktioteoriaa sovellettiin [BacKur83]. Sitten hajautetun lajittelun yksinkertaistetut versiot ovat löytäneet tiensä esimerkkeinä moniin DisCo-aiheisiin dokumentteihin, kuten tähänkin.

Hajautetussa lajittelussa on prosesseja, jotka ovat jonossa vasemmalta oikealle. Jokaisella prosessilla on hallussaan jokin kokonaisluku ja prosessien välillä on kommunikointikanava, jonka välityksellä vierekkäiset prosessit voivat vaihtaa lukunsa keskenään. Tavoitteena on järjestää luvut nousevaan suuruusjärjestykseen (tai pikemminkin ei-laskevaan, koska yhtäsuuruus sallitaan) vasemmalta oikealle.

Hajautetun lajittelun DisCo-spesifikaation systeemiosa on kuvassa 11. Systemissä Processes määritellään luokka Process, jonka objekteilla on kaksi muuttujaa. Muuttujan X arvo on prosessin hallussa oleva luku ja muuttuja Next on viite seuraavaan prosessiin. Aktio Exchange vaihtaa kahden vierekkäisen prosessin luvut, jos ne ovat väärässä järjestyksessä. Lauseke $A.X := B.X \parallel B.X := A.X$; tarkoittaa, että sijoitukset tapahtuvat rinnakkaisesti, eli apumuuttujia ei tarvita.

Systemissä Communication tarkennetaan superpositiolla systeemiä Processes siten, että luokkaan Process lisätään muuttuja Prev, joka on viite edelliseen prosessiin, ja totuusarvoinen muuttuja Sorted, joka on tosi, jos jonon luvut ovat vasemmalta tähän prosessiin asti järjestyksessä. Globaali alkuehto NotSortedOthers on tosi, jos kaikkien prosessien, ensimmäistä lukuun ottamatta,

```

system Processes is
  class Process is
    X: integer;
    Next: Process;
  end Process;

  action Exchange by A:Process; B:Process is
  when A.Next = B and A.X > B.X do
    A.X := B.X || B.X := A.X;
  end Exchange;
end Processes;

system Communication import Processes; is
  extend Process by
    Prev: Process;
    Sorted : boolean;
  end Process;

  initially NotSortedOthers is
    and/P:Process | P.Prev /= Null :: not P.Sorted;

  assert SortedFirst is
    and/P:Process | P.Prev = Null :: P.Sorted;

  assert StructureOK is and/P1:Process; P2:Process::
    (P1.Next /= Null => P1.Next.Prev = P1) and
    (P1 /= P2 => P1.Next /= P2.Next) and
    (+/ P:Process | P.Prev = Null ::1) = 1;

  refined Exchange is
  when ... do
    ...
    if A.Prev /= Null then
      A.Sorted := FALSE;
    end if;
  end Exchange;

  action Communicate by A:Process; B:Process is
  when A.Next = B and A.X <= B.X and A.Sorted and not B.Sorted do
    B.Sorted := TRUE;
  end Communicate;
end Communication;

```

Kuva 11: Systemeit Processes ja Communication.

```

creation Life of Communication is
  P1, P2: Process;
  P1 := new Process;
  P1.Prev := Null;
  P1.X := 7;
  P1.Sorted := TRUE;

  for i in 1..5 loop
    P2 := new Process;
    P1.Next := P2;
    P2.Prev := P1;
    P2.X := i;
    P1 := P2;
  end loop;
end Life;

```

Kuva 12: Luontiosa Life.

Sorted-kentän arvo on epätosi. Spesifikaatioon lisätään myös kaksi globaalia asertiota. SortedFirst varmistaa, että ensimmäisen prosessin Sorted-kentän arvo on aina tosi, ja StructureOK että prosessit muodostavat kahteen suuntaan linkitetyn jonon.

Aktiota Exchange tarkennetaan siten, että sen vasemmanpuoleisen osallistujan Sorted-kentän arvoksi asetetaan epätosi, paitsi jos se on jonon ensimmäinen prosessi vasemmalta lukien. Tämä vaaditaan, koska vaihdon jälkeen vasemmanpuoleinen osallistuja ei ole enää välttämättä järjestyksessä sen vasemmalla puolella olevan prosessin kanssa. Ensimmäinen prosessi on kuitenkin aina järjestyksessä itsensä kanssa. Lisäksi määritellään aktio Communicate, joka vaihtaa oikeanpuoleisen osallistujansa Sorted-kentän arvoksi tosi, jos sen vasemmanpuoleisen osallistujan Sorted-kentän arvo on tosi ja osallistujien luvut ovat keskenään oikeassa järjestyksessä. Näin tieto jonon järjestyksestä saadaan kulkemaan vasemmalta oikealle.

Kuvassa 12 annetaan systeemille Communicate luontiosa Life. Siinä luodaan kuusi prosessia, annetaan niille järjestettävät luvut ja alustetaan viitteet siten, että prosessit muodostavat jonon.

Esimerkissä invarianttina on, että jono on järjestyksessä vasemmalta siihen prosessiin asti, joka on oikealta lukien ensimmäinen jonka Sorted-kentän arvo on tosi. Kun viimeisen prosessin Sorted-kentän arvo on tosi, koko jono on järjestyksessä.

6.3 Hajautettu lajittelu TLA⁺:lla

Kuvassa 13 on hajautetun lajittelun spesifikaatio esitettyinä TLA⁺:lla. Moduulin *Sort* parametrina on tupla (tuple) *Process*, jonka komponentit ovat tietueita. Tietueet vastaavat luokan *Process* objekteja ja niillä on kentät *X*, *Prev*, *Next*

module <i>Sort</i>
parameters <i>Process</i> : VARIABLE
imports <i>Naturals</i>
definition $n \triangleq \text{CHOOSE } j : (j \in \text{DOMAIN } Process) \wedge (\forall l \in \text{DOMAIN } Process : l \leq j)$
assumption $Assump \triangleq \wedge \forall i \in \{1, \dots, n\} : \vee Process[i].X \in Nat$ $\vee Process[i].X = 0$ $\vee (0 - Process[i].X) \in Nat$ $\wedge \forall i \in \{1, \dots, n\} : Process[i].Prev \in \{1, \dots, n\} \cup \{\text{"null"}\}$ $\wedge \forall i \in \{1, \dots, n\} : Process[i].Next \in \{1, \dots, n\} \cup \{\text{"null"}\}$ $\wedge \forall i \in \{1, \dots, n\} : Process[i].Sorted \in \{\text{TRUE}, \text{FALSE}\}$
definitions $U \triangleq \{Process[i].X, Process[i].Prev, Process[i].Next,$ $Process[i].Sorted : i \in \{1, \dots, n\}\}$ $Init \triangleq \wedge Process[1].Sorted$ $\wedge Process[1].Prev = \text{"null"}$ $\wedge \forall i \in \{2, \dots, n\} : \neg Process[i].Sorted \wedge Process[i].Prev = i - 1$ $\wedge \forall i \in \{1, \dots, n - 1\} : Process[i].Next = i + 1$ $\wedge Process[n].Next = \text{"null"}$ $exchange \triangleq \exists k_1, k_2 \in \{1, \dots, n\} :$ $\wedge k_1 \neq k_2$ $\wedge Process[k_1].Next = k_2 \wedge Process[k_1].X > Process[k_2].X$ $\wedge Process[k_1].X' = Process[k_2].X$ $\wedge Process[k_2].X' = Process[k_1].X$ $\wedge \text{if } Process[k_1].Prev \neq \text{"null"}$ $\quad \text{then } Process[k_1].Sorted' = \text{FALSE}$ $\quad \text{else } Process[k_1].Sorted' = \text{TRUE}$ $\wedge \text{UNCHANGED } \langle U \setminus \{Process[k_1].X, Process[k_2].X, Process[k_1].Sorted\} \rangle$ $communicate \triangleq \exists k_1, k_2 \in \{1, \dots, n\} :$ $\wedge k_1 \neq k_2$ $\wedge Process[k_1].Next = k_2 \wedge Process[k_1].X \leq Process[k_2].X$ $\wedge Process[k_1].Sorted \wedge \neg Process[k_2].Sorted$ $\wedge Process[k_2].Sorted' = \text{TRUE}$ $\wedge \text{UNCHANGED } \langle U \setminus \{Process[k_2].Sorted\} \rangle$ $Spec \triangleq \wedge Init$ $\wedge \square [exchange \vee communicate]_U$

Kuva 13: Moduuli *Sort*.

ja *Sorted*, joilla on samat merkitykset kuin DisCo-spesifikaatiossa. Moduuli ottaa käyttöön *Naturals*-moduulin, joka määrittelee luonnollisten lukujen joukon *Nat*. Koska tietueiden määrä tuplassa on tuntematon, se otetaan selville operaattorin CHOOSE avulla. Tuplat on TLA⁺:ssa funktioita ja merkintä *Domain Process* tarkoittaa tuplan *Process* määrittelyjoukkoa eli joukkoa $\{1, \dots, n\}$, jossa n on tietueiden lukumäärä tuplassa. Muuttujat *Prev* ja *Next* voivat saada arvoja tuplan *Process* määrittelyjoukosta, siten että arvo i tulkitaan viitteeksi i :nneteen prosessiin.

Spesifikaatiossa esitetään oletus *Assump*, joka väittää, että tietueiden kentät ovat oikeaa tyyppiä. Prosessin hallussa oleva luku on DisCossa **integer**-tyyppiä ja oletuksessa tarkistetaan, että joko kentän X arvo on luonnollinen luku tai sen vastaluku on luonnollinen luku. Siltä varalta, että *Nat* ei sisällä lukua 0, tarkistetaan myös se mahdollisuus että X :n arvo on 0.

Avainsanan **definitions** jälkeen annetaan joukko määritelmiä: U on kaikkien muuttujien joukko, *Init* vastaa systeemin Communication alkuehtoja ja assertioita, *exchange* ja *communicate* vastaavat samannimisiä aktioita DisCossa ja *Spec* on koko järjestelmän spesifikaatio kanonisessa muodossa. Aktiot vastaavat DisCon aktioita sillä erotuksella, että niiden vahteihin on täytynyt lisätä ehto, joka varmistaa, että osallistujat ovat eri prosesseja. DisCossa tämä määritellään kielen tasolla, mutta koska TLA:ssa ei tällaista rajoitusta ole, se täytyy lisätä eksplisiittisesti. Aktioissa täytyy myös ilmoittaa lauseen UNCHANGED avulla niiden muuttujien joukko, joiden arvot eivät muutu.

Spec kertoo, että alkutilan jälkeen tulevat askeleet ovat joko *exchange*- tai *communicate*-askelia tai änkytysaskelia, joissa U :n alkioiden arvot eivät muutu.

6.4 Muunnoksen yleistäminen

Alkuperäinen spesifikaatio koostui kahdesta DisCo-systeemistä, mutta lopputuloksena oli vain yksi TLA⁺:n moduuli. Muunnoksessa unohdettiin tarkoituksella ylimmän tason systeemi Processes ja muunnettiin vain tarkennuksen lopputuloksena saatu systeemi Communication. TLA⁺-spesifikaatio olisi voitu hajottaa kahdeksi moduuliksi, mutta sitä olisi ollut vaikea perustella näin lyhyen spesifikaation tapauksessa.

DisCossa jokainen systeemi on spesifikaatio, jolla on operationaalinen tulkinta. Vaiheittainen tarkentaminen on sisäänrakennettuna menetelmään ja jokainen systeemi kuvaa järjestelmän toiminnan jollakin tarkkuustasolla. Vaiheittainen tarkentaminen toteutetaan superpositiolla ja sen halutaan takaavan turvallisuusominaisuuksien säilyminen tarkennuksen edetessä. Tästä syystä importoitujen systeemien muuttujia tai tiloja ei saa muuttaa.

Tästä seuraa, että kaikki ne aktiot, jotka muuttavat muuttujien arvoja tai aiheuttavat tilasiirtymiä on annettava samassa systeemissä kuin kyseiset muuttujat ja tilat. Koska aktioiden vahteja on mahdollista tiukentaa tarkennettaessa, annetaan ensimmäisessä systeemissä usein aktioita joiden vahdit ovat identti-

sesti tosia. [Jär92]

Spesifioitaessa TLA^+ :lla on vaiheittainen tarkentaminen kuitenkin vain yksi mahdollinen lähestymistapa. Spesifikaation voi strukturoida kuten DisCossakin, mutta määrittelijän täytyy itse varmistua siitä, että turvallisuusominaisuudet säilyvät tarkennettaessa voimassa.

Matematiikassa määritellään uusia abstraktioita käyttämällä hyväksi vanhojen määritelmiä, sen sijaan että muutettaisiin jotain jo määriteltyä [Val94]. Tämä tarkoittaa sitä, että tarkentaminen täytyy tehdä siten, että jokaisessa tarkennusaskeleessa annetaan uudet määritelmät, jotka ovat edellisessä vaiheessa annettujen määritelmien ja niihin lisättävien tarkennusten konjunktioita.

TLA^+ ei estä määrittelemästä aktioita ja luokkia vaiheittain, konjungoimalla niihin tarkennuksen edessä uusia lausekkeita, ja antamasta samassa moduulissa useita kanonisessa muodossa olevia spesifikaatioita, jotka kuvaavat järjestelmän tietyllä tarkkuustasolla. Tämä ei kuitenkaan ole välttämättä paras tapa strukturoida spesifikaatioita TLA^+ :ssa.

TLA^+ :ssa on luonnollista käyttää moduulaarisuuden tukena TLA :n tapaa piilottaa muuttujia. Kun spesifikaatio annetaan kanonisessa muodossa, osa muuttujista voidaan piilottaa kvantifoimalla ne pois temporaalisen olemassaolokvanttorin avulla. Muuttujia, jotka piilotetaan, pidetään spesifikaation apumuuttujina, joita ei tarvitse välttämättä kuvata toteutukseen, mikäli spesifikaation toteuttaja löytää jonkin muun tavan saada varsinaiset muuttujat käyttäytymään spesifikaation mukaisesti [Mik95].

Piilottamisen semantiikka perustuu siihen, että piilotettuja muuttujia sisältävän spesifikaation tyydyttävät kaikki sellaiset käyttäytymiset, joista saadaan alkuperäisen spesifikaation tyydyttävä käyttäytyminen lisäämällä äärellinen määrä änkytysaskelia mielivaltaisesti minkä tahansa askelten väliin, ja sijoittamalla piilotetuille muuttujille tiloissa jotkin arvot. Intuitiivisesti tämä merkitsee sitä, että piilotettujen muuttujien arvoista ei tarvitse välittää.

DisCo-spesifikaatio voidaan muuntaa Järvisen esittämän muunnoksen avulla TLA -spesifikaatioksi, josta saadaan TLA^+ -spesifikaatio kirjoittamalla se moduuliksi. Sellaiset muuttujat, joiden arvot eivät muutu, määritellään vakioparametreiksi avainsanalla `CONSTANT` ja piilottamattomat tavalliset muuttujat muuttujaparametreiksi avainsanalla `VARIABLE` [Lam93]. Laajan spesifikaation tapauksessa on järkevää hajottaa spesifikaatio useammaksi moduuliksi.

TLA^+ :ssa ei moduulien käyttöä ole rajoitettu, joten spesifikaation hajottaminen moduuleiksi on suunnittelukysymys. Kohdassa 4.6 esitellyssä Musical Chairs -spesifikaatiossa käytetty tapa sijoittaa intuitiivisesti yhteenkuuluvat määritelmät samaan moduuliin, esimerkiksi aktiot omaan moduuliinsa, on vain yksi tapa. Ohjeita DisCo-spesifikaation jakamisesta TLA^+ :n moduuleiksi on hankala antaa, koska ei ole olemassa vain yhtä oikeaa tapaa strukturoida spesifikaatioita.

DisCossa spesifikaatioiden kuvaamat systeemit ovat suljettuja, koska spesifi-

kaatioiden halutaan olevan suoritettavia. TLA^+ ei rajoita sitä, onko systeemi suljettu vai ei. TLA^+ mahdollistaa spesifikaation strukturoinnin esimerkiksi siten, että spesifioitava järjestelmä kuvataan avoimena systeeminä ja ympäristö annetaan erillisissä moduuleissa.

TLA^+ -spesifikaatiossa voi ennakoida spesifikaation jakoa prosesseiksi kirjoittamalla jokaisen prosessin omaksi moduulikseen. Lähteessä [Lam93] Lamport soveltaa tätä lähestymistapaa Musical Chairs -spesifikaatioon ja antaa jokaisessa prosessia kuvaavassa moduulissa spesifikaation, joiden konjunktio on koko järjestelmän spesifikaatio. Koko järjestelmän spesifikaatiosta piilotetaan lopuksi kunkin prosessin käyttämät apumuuttujat.

Muunnettaessa spesifikaatiota käsin TLA^+ -spesifikaation strukturi kannattaa suunnitella itse. Tällöin on kuitenkin aina syytä tarkastella spesifikaatiota kriittisessä valossa ja varmistuttava siitä, että sen tyydyttävät samat käyttäytymiset kuin Järvisen esittämällä muunnoksella saatavan spesifikaation.

7 TLA⁺ ja DisCo spesifointimenetelminä

7.1 Menetelmien erot

TLA⁺:lla ja DisColla on paljon yhteistä, sillä molempia menetelmiä voidaan käyttää reaktiivisten järjestelmien formaaliin spesifointiin. Menetelmät on kuitenkin suunnattu erilaisille käyttäjille. TLA⁺ soveltuu vahvan matemaattisen taustan omaavalle määrittelijälle, joka haluaa esittää algoritminsa TLA:lla ja todistaa sen ominaisuuksia, kun taas DisCo on pikemminkin ohjelmoijalle soveltuva työkalu formaaliin ohjelmistosuunnitteluun. Menetelmien kohdistaminen on ollut perusteltua, koska menetelmä, joka pyrkii sopimaan kaikille, ei yleensä sovi kenellekään.

Maaailma on pullollaan esimerkkejä spesifikaatioista, jotka on laadittu osoittamaan jonkin spesifointimenetelmän sopivuutta tietyn tyyppisten ongelmien ratkaisemiseen. Valitettavasti monet menetelmät eivät toimi, jos niitä yritetään soveltaa muun tyyppisiin ongelmiin. Leluesimerkit puolustavat paikkaansa, kun halutaan opettaa menetelmän käyttöä mutta niiden perusteella ei voi tehdä pitkälle meneviä johtopäätöksiä menetelmän käyttökelpoisuudesta. Toisaalta on selvää, että oikeiden järjestelmien spesifikaatiot ovat aivan liian pitkiä ja monimutkaisia sopiakseen esimerkeiksi. Esimerkit pitäisi kuitenkin valita siten, että ne ovat konkreettisia, eivät vain menetelmän ominaisuuksien esittelyä. Aiempien lukujen esimerkit lukeutuvat lelusarjaan, mutta niistä voi kuitenkin vetää joitakin johtopäätöksiä.

Esimerkit osoittavat ensinnäkin sen, että yksinkertaisten spesifikaatioiden muuntaminen näiden kahden kielen välillä voi olla todella helppoa. Tämä johtuu pitkälti siitä, että molemmat menetelmät perustuvat aktio-paradigmaan. Toiseksi, TLA⁺-spesifikaatioissa täytyy ilmaista paljon sellaista eksplisiittisesti, mitä DisCo-spesifikaatio kertoo implisiittisesti. Tämän voi katsoa olevan perua perustavaa laatua olevasta erosta matematiikan ja ohjelmointikielten välillä. Itse asiassa se, että ohjelmoija olettaa ympäristön takaavan jotain implisiittisesti, on yksi syy virheiden syntymiseen. Kolmanneksi, TLA⁺ on kielenä ilmaisuvuomaisempi kuin DisCo ja sallii hyvinkin monimutkaisten matemaattisten funktioiden määrittämisen.

Ohjelmointikieliet ovat perinteisesti tarjonneet tarkan syntaksin ja matematiikka tarkan semantiikan. Tämä ero on nähtävissä jossakin määrin myös TLA⁺:ssa ja DisCossa, sillä siinä missä spesifikaation simulointi DisCo-työkalun avulla vaatii syntaksin ehdotonta oikeellisuutta, keskittää TLA⁺ määrittelijän huomion pikemminkin siihen, mitä spesifikaatio todella merkitsee. DisCon käyttäjän täytyy tuntea DisCo-kielen semanttinen perusta, TLA, vain halutessaan todistaa spesifikaationsa oikeaksi, kun taas TLA⁺:aa voidaan pitää TLA:n syntaktisena makropakettina, jota ei voi käyttää ennen TLA:n perusteiden tuntemista.

Hyvän spesifointimenetelmän perusedellytys on joustavuus; menetelmän käytön pitää olla yksinkertaista, olipa spesifikaation pituus sitten yksi tai sata si-

vua. Modulaarisuus on yksi keino saavuttaa joustavuutta kielen tasolla. TLA^+ :n ja DisCon soveltuvuudesta satojen sivujen mittaisten spesifikaatioiden laatimiseen ei vielä ole kokemuksia, mutta mikään ei viittaa siihen, että ylitsepääsemättömiä ongelmia esiintyisi.

Kummankaan menetelmän käytöstä laajan, todellisen järjestelmän spesifiointiin ei vielä ole kovinkaan paljon käytännön kokemuksia. Tämä johtuu pitkälti siitä, että kumpikaan menetelmä ei vielä ole laajemmin herättänyt teollisuuden kiinnostusta.

7.2 Kielten erot

TLA^+ on kielenä ilmaisuvoimaisempi kuin DisCo. DisCossa ei esimerkiksi salli ta kvantifiointia luonnollisten lukujen joukon yli, mutta TLA^+ ei tunne tällaisia rajoituksia. DisCo-kielen rajoitukset johtuvat siitä, että kieli on suoritettava, ja esimerkiksi kvantifioinnin salliminen äärettömän joukon yli aiheuttaisi ongelmia DisCo-kääntäjää tehtäessä [Aal96]. Nykyisin käytössä oleva DisCo-kieli ei rajoita esimerkiksi **integer**-tyypin lukualuetta, mutta toteutuksessa se saa jotkin rajat. Vaikeasti toteutettavia ominaisuuksia ei kannata lisätä kieleen ilman hyviä perusteluja. Toisaalta kielen uuteen versioon on vaikeasti toteutettavia ominaisuuksia jo lisätty.

DisCo-kääntäjä tarkastaa spesifikaation syntaksin ennen sen simulointia DisCo-työkalussa. Kääntäjä ilmoittaa havaitsemistaan virheistä, mikä auttaa määrittelijää kirjoittamaan syntaktisesti oikeita DisCo-spesifikaatioita. TLA^+ -kääntäjää ei tätä kirjoitettaessa ole saatavilla eikä määrittelijällä ole syntaksitarkastuksen tekemiseen käytettävissään mitään muutakaan apuvälinettä. Puute toivottavasti korjaantuu, kun kielen syntaksi vakiintuu.

DisCo-kieli muistuttaa ohjelmointikieltä, mikä mahdollistaa ohjelmoijalle kiivuttoman siirtymisen formaaliin ohjelmistosuunnitteluun. Selkeän syntaksin ansiosta spesifikaatioiden ymmärtäminen sujuu jopa ilman lisäkoulutusta [Setä92]. Koska jokainen DisCo-spesifikaatio on muunnettavissa TLA :ksi, voidaan DisCo-kieltä tarkastella myös ohjelmointikielimäisenä tapana kirjoittaa TLA -spesifikaatioita. DisCon käyttäjän ei kuitenkaan tarvitse välttämättä edes tietää TLA :n olemassaolosta.

Setälä [Setä92] moittii DisCo-kieltä siitä, että se ei ole erityisen nopeaa kirjoittaa, se sisältää suuren määrän syntaktista sokeria, ja spesifikaatioista tulee aina melko pitkiä. Suurimmiksi eduiksi vastaavasti mainitaan spesifikaatioiden modulaarisuus ja selkeys. Voidaan kuitenkin sanoa, että spesifikaatioiden selkeys on saavutettu juuri oikealla määrällä syntaktista sokeria ja että spesifikaation kirjoittaminen vie kuitenkin vain pienen osan koko määrittelyyn kuluva ajasta.

DisCossa korostuu ylimmän tason spesifikaation ja tarkennuksen vaiheiden merkitys [Sys94]. Jos ylimmän tason spesifikaatiota suunniteltaessa ei ota huomioon tulevia tarkennuksia, joutuu yleensä palaamaan takaisin ylimmälle tasol-

le. Superposition tehokas käyttö vaatii harjaantumista, mutta määrittelijällä on käytössään ohjeita spesifikaation laatimisesta DisColla [Sys94, Jär92, Setä92].

DisCo-kieli ei sovellu esim. tietorakenteiden kuvaamiseen, koska siinä ei ole tarvittavia tietotyyppejä ja rakenteita [Sys94]. Kielen perustuminen olioille helpottaa spesifikaatioiden uudelleenkäyttöä. Järvinen [Jär92] mainitsee kielen parhaimmiksi ominaisuuksiksi:

1. aktio- ja olio-paradigmojen yhdistäminen
2. spesifointi suljettuna systeeminä tukee spesifikaatioiden pohjalta tehtävää päättelyä, mutta ei aiheuta ongelmia uudelleenkäyttämisen tai modulaarisuuden suhteen
3. superpositio, systeemien yhdistäminen modulaarisuuden päämekanismina.

TLA⁺ perustuu aktio-paradigmaan kuten DisCokin ja antaa mahdollisuuden kirjoittaa spesifikaatiot suljettuina systeemeinä. Tietorakenteet kuvataan määrittelemällä funktioita ja joukkoja. Oliopiirteitä tai superposition kaltaista mekanismia ei TLA⁺:aan ole rakennettu sisään. Tarkennettaessa spesifikaatiota vaiheittain ei ole mitään varmuutta siitä, että turvallisuusominaisuudet säilyvät.

Modulaarisuus on pääasiallinen keino spesifikaatioiden strukturointiin molemmissa kielissä. TLA⁺:ssa moduulien käyttöä ei oikeastaan ole rajoitettu ollenkaan. Niitä käytetään pääasiassa hajottamaan laajat spesifikaatiot pienemmiksi, helpommin ymmärrettäviksi kokonaisuuksiksi. Tätä tapaa on käytetty esimerkiksi kohdassa 4.6 esitetyssä Musical Chairs -spesifikaatiossa, jossa mm. aktioiden määrittelyt on koottu yhteen moduuliin. DisCossa vaiheittainen tarkentaminen on rakennettu selkeästi sisään menetelmään ja DisCon moduulit voidaan käsittää tarkennuksen vaiheiksi.

Perustavaa laatua oleva ero DisCon ja TLA⁺:n välillä on spesifikaation käsite. DisCossa jokainen systeemi on spesifikaatio, jolla on operationaalinen tulkin-ta. TLA⁺:ssa moduuli ei yleensä ole spesifikaatio, vaan vain kokoelma määritelmää, jotka eivät yksinään merkitse mitään. Spesifikaatio annetaan useimmiten eksplisiittisesti TLA:n kanonisessa muodossa ja se voi sisältää määritelmiä useista **import**- tai **include**-lauseen avulla käyttöön otetuista moduuleista.

7.3 Uusi DisCo-kieli

DisCo-kielestä on suunnitteilla uusi versio. Syynä sen kehittämiseen ovat eräät vanhan version puutteet ja ongelmat sekä se, että kieltä halutaan kehittää OMT-määrittelymenetelmän periaatteiden mukaiseksi. Koska kieltä on voitava suorittaa, ei uusien ominaisuuksien toteuttaminen ole kuitenkaan ongelmaton-ta. Jos esimerkiksi sallittaisiin kvantifioiminen luonnollisten lukujen joukon yli, koituisi siitä kääntäjää tehtäessä ongelmia. [Aal96]

Tavoitteena on perintämekanismien muuttaminen aggregoinnista sellaiseksi, joka vastaa paremmin muiden oliokielen perintämekanismia. Nykyinen perintämekanismi toteuttaa pikemminkin "has a" -relaation kuin "is a" -relaation johdetun luokan ja kantaluokan välille. Uudessa kielessä ei ole sellaisia ominaisuuksia kuin viite objektiin, moduuli, **stuff**-tyyppi tai **if**-lause aktion rungossa. Ne on jätetty pois tarpeettomina tai korvattu uusilla rakenteilla. Myös tilakoneiden toteutus ja kielen syntaksi ovat kokeneet muutoksia. Uutena piirteenä perintämekanismien ohella on mm. tietuetyyppi ja se, että kvantifiointi voidaan suorittaa muidenkin kuin luokkien, esimerkiksi **integer**-tyypin yli. Uusi versio sisältää myös reaaliajan kuvaamiseen tarvittavat piirteet. [Aal96]

7.4 Menetelmien kehittämisestä

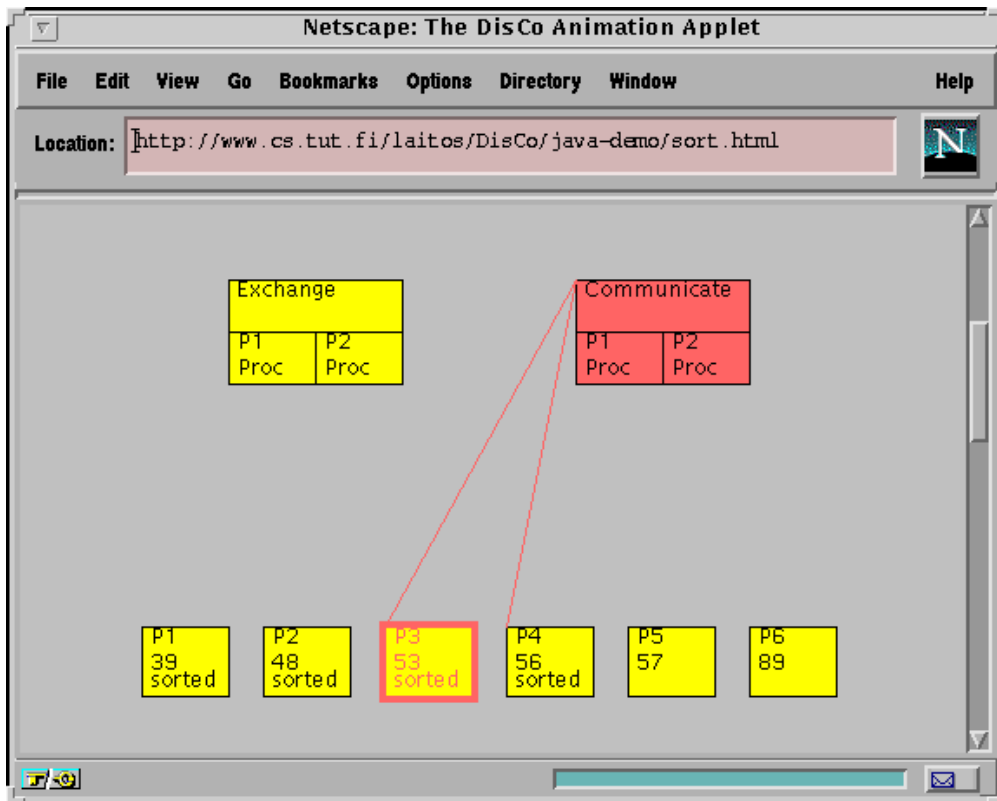
Koska formaalilla todistuksella pyritään osoittamaan spesifikaatio virheettömäksi, on todistuksenkin tärkein ominaisuus virheettömyys. Mikäli todistukset joudutaan tekemään käsin, menetetään suuri osa menetelmän luotettavuudesta, koska käsin tehtävät todistukset ovat hyvin alttiita virheille. Formaalin menetelmän käyttökelpoisuutta lisäisi todistusten osittainkin automatisointi. Menetelmän tulisi myös tukea dokumentointia ja spesifikaation toteutusta sekä toteutuksen testausta.

Todistusten virheettömyys pyritään saavuttamaan tarkkuudella, mutta todistus tarvitsee myös selkeän rakenteen, jotta se olisi helposti ymmärrettävissä ja hyväksyttävissä. Lamport on kehittämässä hypertekstipohjaista järjestelmää todistusten esittämiseksi. Sen avulla pitäisi olla mahdollista tarkastella todistusta sillä tarkkuudella, jonka katsoo kussakin vaiheessa tarpeelliseksi. Itse todistamista ei tämäkään järjestelmä kuitenkaan helpota.

DisCo-spesifikaatiota voi animoida DisCo-työkalun avulla. Tulevaisuudessa animointi on ehkä mahdollista myös WWW-selaimessa. Spesifikaatioiden animointia kokeillaan Java-kielellä, joka mahdollistaisi spesifikaation animoinnin monelle käyttäjälle yhtäaikaaisesti ja laitteistoriippumattomasti. Kuvassa 14 on hajautetun lajittelun animointi toteutettuna Java-applettina.

Sekä TLA⁺ että DisCo on kehitetty enemmänkin tutkimuskäyttöön kuin teollisuuden tarpeita ajatellen. Molempia menetelmiä kehitetään edelleen. TLA⁺:n keskeneräisyys häiritsee, sillä syntaksin elävyys hankaloittaa spesifikaatioiden kirjoittamista. Tässäkin työssä käytetyt esimerkit ovat lukuhetkellä jo syntaksinsa osalta vanhentuneita. Niin kauan kuin määrittelijät käyttävät kukin omaa syntaksiaan, menetetään osa menetelmän eduista.

Eräs tämän diplomityön tarkoituksista oli selvittää, kuinka DisCo voisi hyötyä TLA⁺:sta. TLA⁺:n hyödyntäminen on hankalaa. Se on kielenä DisCoa ilmaisuvoimaisempi ja sallii määrittelijän käyttää matemaattista luovuuttaan vapaammin. Jos DisCo-kielessä sallittaisiin samoja vapauksia, olisi etsittävä keinot toteuttaa uudet piirteet. Muussa tapauksessa vaikeasti laskettavien lausekkeiden määrä kasvaisi, mikä tarkoittaisi sitä, että spesifikaation suorituksessa tarvit-



Kuva 14: Hajautetun lajittelun animointi Java-applettina.

taisiin apua käyttäjältä. Se ei kuitenkaan ole hyvä ratkaisu. Kysymys kannattaa nostaa esille uudelleen, jos TLA⁺:lle saadaan toimiva kääntäjä.

8 Yhteenveto

8.1 Vertailun mielekkyys

Tämän työn tarkoituksena oli tarkastella spesifikaatioiden strukturointia TLA^+ :ssa ja DisCossa. Menetelmien vertailu oli mielekäästä, koska molemmat on tarkoitettu reaktiivisten järjestelmien formaaliin spesifiointiin ja niillä on yhteinen semanttinen perusta, TLA .

TLA^+ ja DisCo ovat kuitenkin kielinä varsin erilaiset. Tämä johtuu siitä, että menetelmillä on erilaiset lähtökohdat ja ne on suunnattu erilaisille käyttäjille. TLA^+ :n tarkoituksena on parantaa TLA -spesifikaatioiden ymmärrettävyyttä ohjelmointikielistä tutuilla rakenteilla, kun DisCo on alun perin tehty muistuttamaan mahdollisimman paljon ohjelmointikieltä. Voidaan sanoa, että siinä missä tyypillinen DisCon käyttäjä tuntee omakseen ohjelmointikielten maailman, siinä TLA^+ :n käyttäjä tuntee matematiikan maailman.

TLA^+ onnistuu tavoitteessaan parantaa TLA -spesifikaatioiden ymmärrettävyyttä. Sen mukanaan tuomat rakenteet ovat merkittävä parannus TLA :han, sillä ymmärrettävyys on spesifikaation tärkeimpiä ominaisuuksia. Tarkasta ja virheettömästä spesifikaatiosta ei ole mitään hyötyä, jos vain sen kirjoittaja pystyy sitä ymmärtämään.

DisCo on ohjelmistosuunnittelijalle helposti omaksuttava menetelmä formaalin spesifikaation laatimiseen. DisCo-kieli yhdistää onnistuneesti olio- ja aktio-paradigmat ja tarjoaa määrittelijälle tavan rakentaa spesifikaationsa tarkentamalla sitä vaiheittain, niin, että turvallisuusominaisuudet säilyvät.

Modulaarisuus on strukturoinnin päämekanismi kummassakin kielessä. Moduuleja kuitenkin käytetään TLA^+ :ssa ja DisCossa hieman eri tavoin. Matematiikkaa ei perinteisesti ole strukturoitu moduulien avulla, eikä niiden käytölle ole TLA^+ :ssa asetettu paljonkaan rajoituksia. DisCossa systeemi voidaan nähdä importoidun systeemin tarkennuksena ja toisaalta tarkennuksen lähtökohtana importoilvalle systeemille. DisCossa vaiheittainen tarkentaminen on paljon oleellisempi osa menetelmää kuin TLA^+ :ssa, ja DisCon systeemit ovatkin tavallaan tarkennusaskelia.

DisCossa jokainen systeemi on spesifikaatio, jolla on operationaalinen tulkinta. TLA^+ :ssa moduuli ei sitä vastoin välttämättä muodosta spesifikaatiota, vaan se voi olla vain joukko määritelmiä, jotka eivät yksinään merkitse mitään. TLA :n kanonisessa muodossa oleva spesifikaatio pitää TLA^+ :ssa määritellä eksplisiittisesti, ja sen sisältämät määritelmät voivat olla peräisin useasta eri moduulista.

Jokainen DisCo-spesifikaatio on muunnettavissa TLA :ksi, mikä mahdollistaa todistamisen TLA :lla. Muunnos on mekaaninen, mutta muunnettaessa käsin, kannattaa käyttää hyväksi TLA^+ :n tarjoamia strukturointimahdollisuuksia ja hajottaa laaja spesifikaatio moduuleiksi.

DisCo-työkalu luo tavallaan käyttöliittymän koko menetelmälle ja konkreti-

soi spesifikaation antamalla sille graafisen ulkoasun. Työkalun avulla DisCo-spesifikaatioita voi suorittaa ja testata.

Hintana DisCo-kielen suoritettavuudesta on, että ilmaisuvoimaa on täytynyt rajoittaa. TLA⁺ antaa kielen tasolla määrittelijälle vapaammat kädet, mutta ei toisaalta tarjoa mahdollisuutta testata spesifikaatiota.

Formaalin menetelmän suurimpia etuja on mahdollisuus lisätä todistuksien avulla uskoa spesifikaation oikeellisuuteen. Kumpikin menetelmä tarvitsisi tuekseen mekaanisen teoreemantodistimen, koska formaalien todistusten tekeminen käsin on hyvin vaikeaa. Myös dokumentointia ja spesifikaation toteutusta sekä toteutuksen testausta varten olisi syytä kehittää työkaluja.

8.2 Työn arviointi

Työn otsikko osoittautui vaativaksi, mutta onneksi myös joustavaksi. Vaikka menetelmillä onkin paljon yhteistä, ne ovat kuitenkin kielen tasolla varsin erilaisia. Tämä teki osaltaan vertailun mielekkääksi.

Tavoitteena oli löytää TLA⁺:sta sellaisia ominaisuuksia, joita voitaisiin hyödyntää DisCon jatkokehityksessä. Jo työn alkuvaiheessa kävi kuitenkin ilmi, että sellaisten löytäminen oli vaikea tehtävä. TLA⁺ mahdollistaa laskennallisesti vaikeiden lausekkeiden antamisen ja niiden tuominen DisCoon aiheuttaisi ongelmia ainakin DisCo-kääntäjän tekijälle.

DisCo-työkalu osoittautui tarpeelliseksi osaksi menetelmää. Sen avulla onnistuttiin löytämään virheitä monista esimerkeinä käytetyistä DisCo-spesifikaatioista. Koska TLA⁺-spesifikaatioita ei voi testata, vaaditaan niiden laatimisessa erityistä huolellisuutta. Huono puoli DisCo-työkalun käytössä on, että spesifikaatio tulee helposti laadittua kokeile ja korjaa -tyylillä, johon TLA⁺:n käyttäjä ei voi sortua.

Työtä tehdessä vahvistui käsitys siitä, että ohjelmistokoulutuksen saanut henkilö pystyy omaksumaan DisCon ajatusmaailman helpommin kuin TLA⁺:n. Matemaattisesti suuntautuneella henkilöllä tilanne olisi luultavasti täsmälleen päinvastoin.

Näkökulman jatkuva vaihtaminen DisCosta TLA⁺:aan ja toisin päin oli aluksi hankalaa. Työtä hankaloitti myös TLA⁺:n ja sen dokumentaation keskeneräisyys. Se on kuitenkin vain tekosyy sille, että TLA⁺:n täydellinen hallitseminen jäi vain haaveeksi. Toisaalta tavoitteena ei ollutkaan oppia käyttämään TLA⁺:aa täydellisesti, vaan pikemminkin tutustua DisCoon pintaa syvemmillä. Tämä onnistuikin kohtuullisesti.

Tämän diplomityön sivutuotteena ei syntynyt hyödyllistä ohjelmaa tai edes uusia ideoita. Tavoitteena oli kuitenkin ennen kaikkea oppiminen. Työn kautta avautui myös mahdollisuus tutustua tutkimuksen kiehtovaan maailmaan. Tämän kaltaisia mahdollisuuksia diplomi-insinöörin opinnot harvoin tarjoavat.

Lähdeluettelo

- [Aal96] Aaltonen, T., DisCo-kielen kehittämisestä. Diplomityö, Tampereen teknillinen korkeakoulu, 1996.
- [AbaLam94] Abadi, M., Lamport L., An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–1571, September 1994.
- [AbaLam95] Abadi, M., Lamport L., Conjoining specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–533, May 1995.
- [BacKur83] Back, R.J.R., Kurki-Suonio, R., Decentralization of process nets with a centralized control. *Distributed Computing 3* (1989), 73-87. An earlier version in *2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug. 1983, 131-142.
- [BacKur84] Back, R.J.R., Kurki-Suonio, R., A case study in constructing distributed algorithms: distributed exchange sort. In *Proceedings of the Winter School on Theoretical Computer Science*, Lammi, Finland, January 1984, Finnish Society of Information Processing Science, 1-33.
- [BacKur88a] Back, R.J.R., Kurki-Suonio, R., Serializability in distributed systems with handshaking. In *Automata, Languages and Programming* (Ed. T. Lepistö and A. Salomaa), LNCS 317, Springer-Verlag 1988, 52-66.
- [BacKur88b] Back, R.J.R., Kurki-Suonio, R., Distributed cooperation with action systems. *ACM Transactions on Programming Languages and Systems*, 10(4):513–554, October 1988.
- [ChaMis88] Chandy, K.M., Misra, J., *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
- [ESA96] European Space Agency, ARIANE 5: Flight 501 Failure, Report by the Inquiry Board. At URL <http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html> on the World Wide Web, Paris, France, July 1996.
- [HaiMär95] Haikala, I., Märijärvi J., *Ohjelmistutuotanto*. Suomen ATK-kustannus, 1995.
- [Jär92] Järvinen, H.-M., The Design of a Specification Language for Reactive Systems. Doctoral Dissertation, Tampere University of Technology, 1992.
- [JärKur90] Järvinen, H.-M., Kurki-Suonio, R., The DisCo Language and Temporal Logic of Actions. Tampere University of Technology, Software Systems Laboratory, Report 11, 1990.
- [Kel95] Kellomäki, P., Mechanizing Invariant Proofs of Joint Action Systems. In *Proceedings of the Fourth Symposium on Programming Languages and Software Tools*, Visegrad, Hungary, June 1995

- [Kur96a] Kurki-Suonio, R., Reactive Systems. Unpublished lecture notes, 1996.
- [Kur96b] Kurki-Suonio R., Fundamentals of Object-Oriented Specification and Modeling of Collective Behaviors. Manuscript, 1996.
- [Lad96] Ladkin, P.B., Formal But Lively Buffers in TLA+. At URL <http://www.techfak.Uni-Bielefeld.DE/techfak/persons/ladkin/newbuffer.ps.gz> on the World Wide Web, 1996.
- [Lam93] Lamport, L., Musical Chairs in TLA+. Manuscript, 1993.
- [Lam94] Lamport, L., The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
- [Lam95] Lamport L., TLA+. At URL <http://www.research.digital.com/SRC/tla/tla-plus.ps.Z> on the World Wide Web, 1995.
- [Lam96] Lamport L., The Syntax and Semantics of TLA+. Part 1: Definitions and Modules. At URL <http://www.research.digital.com/SRC/tla/modules.ps.Z> on the World Wide Web, 1996.
- [Mik92] Mikkonen, T., Formaalin määrittelyn toteutusperiaatteet. Diplomityö, Tampereen teknillinen korkeakoulu, 1992.
- [Mik95] Mikkonen, T., Implementation of Reactive Systems based on Closed-System Specifications. Licentiate of Technology thesis, Tampere University of Technology, 1995.
- [Mil56] Miller, G.A., The magical number seven, plus or minus two: some limits on our capacity for processing information. *The Psychological Review* 63:81–97, March 1956.
- [Par96] Parnas, D.L., Inspecting Critical Software. Unpublished lecture notes, Tampere University of Technology, 14th - 16th August 1996.
- [Seth96] Sethi, R., *Programming Languages: Concepts and Constructs*. Addison-Wesley, second edition 1996.
- [Setä92] Setälä, M., Lentokoneen avioniikan formaali mallintaminen. Diplomityö, Tampereen teknillinen korkeakoulu, 1992.
- [Sys91] Systä, K., A Graphical Tool for Specification of Reactive Systems. In *Proceedings of the Euromicro '91 Workshop on Real-Time Systems*, Paris, France, June 1991, IEEE Computer Society Press, 12-19.
- [Sys94] Systä, K., DisCo menetelmäohje, versio 1.0. Tampereen teknillinen korkeakoulu, 1994.
- [Sys95] Systä, K., A Specification Method for Interactive Systems. Doctoral Dissertation, Tampere University of Technology, 1995.
- [Val94] Valmari, A., Formaalit menetelmät. Julkaisematon luentomoniste, 1994.