# Abstract

## English

In the early 2000s, hardware manufacturers shifted their attention from manufacturing faster—yet purely sequential—unicore processors to manufacturing slower—yet increasingly parallel—multicore processors. In the wake of this shift, parallel programming became essential for writing scalable programs on general hardware. Conceptually, every parallel program consists of workers, which implement primary units of sequential computation, and protocols, which implement the rules of interaction that workers must abide by. As programmers have been writing sequential code for decades, programming workers poses no new fundamental challenges. What is new—and notoriously difficult—is programming of protocols.

In this thesis, I study an approach to protocol programming where programmers implement their workers in an existing general-purpose language (GPL), while they implement their protocols in a complementary domain-specific language (DSL). DSLs for protocols enable programmers to express interaction among workers at a higher level of abstraction than the level of abstraction supported by today's GPLs, thereby addressing a number of protocol programming issues with today's GPLs. In particular, in this thesis, I develop a DSL for protocols based on a theory of formal automata and their languages. The specific automata that I consider, called constraint automata, have transition labels with a richer structure than alphabet symbols in classical automata theory. Exactly these richer transition labels make constraint automata suitable for modeling protocols.

Constraint automata constitute the (denotational) semantics of the DSL presented in this thesis. On top of this semantics, I use two complementary syntaxes: an existing graphical syntax (based on the coordination language Reo) and a novel textual syntax. The main contribution of this thesis, then, consists of a compiler and four of its optimizations, all formalized and proven correct at the semantic level of constraint automata, using bisimulation. In addition to these theoretical contributions, I also present an implementation of the compiler and its optimizations, which supports Java as the complementary GPL, as plugins for Eclipse. Nothing in the theory developed in this thesis depends on Java, though; any language that supports some form of threading

and mutual exclusion may serve as a target for compilation. To demonstrate the practical feasibility of the GPL+DSL approach to protocol programming, I study the performance of the implemented compiler and its optimizations through a number of experiments, including the Java version of the NAS Parallel Benchmarks. The experimental results in these benchmarks show that, with all four optimizations in place, compiler-generated protocol code can compete with hand-crafted protocol code.

A more extensive summary of this thesis appears in Chapter 9.

## Nederlands

Sinds het begin van dit millennium hebben hardwarefabrikanten hun aandacht verschoven van het produceren van snellere—doch puur sequentiële—unicore processors ("éénkernige processoren") naar het produceren van langzamere—doch in toenemende mate parallelle—multicore processors ("meerkernige processoren"). Als een gevolg van deze verschuiving is parallel programmeren een essentieel onderdeel geworden van het schrijven van schaalbare programma's voor algemene hardware. Conceptueel gezien bestaat elk parallel programma uit arbeiders, die primair sequentiële berekeningen uitvoeren, en protocollen, die de interactieregels vastleggen waar arbeiders zich aan moeten conformeren. Aangezien programmeurs al decennialang sequentiële code schrijven zorgt het programmeren van arbeiders voor weinig nieuwe fundamentele uitdagingen. Wat wel nieuw is—en berucht in haar complexiteit—is het programmeren van protocollen.

In dit proefschrift onderzoek ik een methode voor het programmeren van protocollen waarin programmeurs arbeiders implementeren in een bestaande general-purpose language ("algemeen-gebruik taal"), afgekort GPL, terwijl zij protocollen implementeren in een domain-specific language ("domein-specifieke taal"), afgekort DSL. DSLs voor protocollen stellen programmeurs in staat interactie tussen arbeiders op een hoger abstractieniveau uit te drukken dan het abstractieniveau dat vandaag de dag wordt ondersteund door GPLs. Hierdoor word een aantal protocolprogrammeerproblemen van de huidige GPLs aangepakt. Om precies te zijn ontwikkel ik in dit proefschrift een DSL voor protocollen gebaseerd op een theorie van formele automaten en hun talen. De specieke automaten waar ik naar kijk, genaamd constraint automata ("beperkingsautomaten"), hebben transitielabels met een rijkere structuur dan alfabetsymbolen in de klassieke automatentheorie. Precies deze rijkere transitielabels maken constraint automata geschikt voor het modelleren van protocollen.

Constraint automata vormen de (denotationele) semantiek van de DSL die ik presenteer in dit proefschrift. Bovenop deze semantiek gebruik ik twee complementaire syntaxes: een bestaande grafische syntax (gebaseerd op de coordinatietaal Reo) en een nieuwe tekstuele syntax. De hoofdbijdrage van dit proefschrift bestaat uit een compiler en vier van haar optimalisaties, die ik allemaal formaliseer en correct bewijs op het semantische niveau van constraint automata door gebruik te maken van bisimulatie. Naast deze theoretische bij-

dragen presenteer ik ook een implementatie van de compiler en haar optimalisaties, die bestaat uit plugins voor Eclipse. Hoewel deze implementatie enkel Java ondersteunt als de complementaire GPL is niets aan de theorie in dit proefschrift Java-specifiek. Elke taal die threading en mutual exclusion ("wederzijdse uitsluiting") ondersteunt kan dienen als compilatiedoel. Om de praktische haalbaarheid van de GPL+DSL-aanpak te demonstreren bestudeer ik ten slotte de prestaties van de geïmplementeerde compiler en haar optimalisaties in een aantal experimenten, waaronder de Java-versie van de NAS Parallel Benchmarks. De resultaten van deze experimenten laten zien dat, wanneer alle vier optimalisaties in stelling worden gebracht, door de compiler gegenereerde protocolcode zich kan meten met handgeschreven protocolcode.

Hoofdstuk 9 geeft een uitgebreidere samenvatting van dit proefschrift.