Lecture Notes in
Computer Science
757

Utpal Banerjee David Gelernter
Alex Nicolau David Padua (Eds.)

Languages and Compilers for Parallel Computing

5th International Workshop
New Haven, Connecticut, USA, August 1992
Proceedings

Springer-Verlag

Utpal Banerjee David Gelernter
Alex Nicolau David Padua (Eds.)

Languages and Compilers for Parallel Computing

5th International Workshop
New Haven, Connecticut, USA
August 3-5, 1992
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Utpal Banerjee
Intel Corporation
2200 Mission College Blvd., P. O. Box 58119, RN6-18
Santa Clara, CA 95052, USA

David Gelernter
Dept. of Computer Science, Yale University
51 Prospect St., New Haven, CT 06520, USA

Alex Nicolau
Dept. of Information & Computer Science, University of California
444 Computer Science Bldg., Irvine, CA 92717, USA

David Padua
Center for Supercomputing Research and Development
465 Computer and Systems Research Laboratory
1308 West Main St., Urbana, IL 61801, USA

CR Subject Classification (1991): F.1.2, D.1.3, D.3.1, B.2.1, I.3.1

ISBN 3-540-57502-2 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57502-2 Springer-Verlag New York Berlin Heidelberg

Printed in Germany

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Lecture Notes in Computer Science 757

Edited by G. Goos and J. Hartmanis

Foreword

The articles in this volume are revised versions of some of the papers presented at the Fifth Workshop on Languages and Compilers for Parallel Computing that took place in August, 1992 at Yale University in New Haven. The previous workshops in this series were held in Santa Clara (1991), Irvine (1990), Urbana (1989) and Ithaca (1988). We strove as in previous years for a reasonable cross-section of some of the best work in the field, and the papers in this volume show that we succeeded fairly well.

Thanks are due to many people for making the workshop and this volume a success: above all to Chris Hatchell, who provided the organizational and administrative glue that held the whole enterprise together. It's striking how little computers can achieve when all is said and done, and how much everything comes down (as it always has) to the right people working hard.

Utpal Banerjee
David Gelernter
Alex Nicolau
David Padua

Lecture Notes in Computer Science

For information about Vols. 1–690
please contact your bookseller or Springer-Verlag

Vol. 691: M. Ajmone Marsan (Ed.), Application and Theory of Petri Nets 1993. Proceedings, 1993. IX, 591 pages. 1993.

Vol. 692: D. Abel, B.C. Ooi (Eds.), Advances in Spatial Databases. Proceedings, 1993. XIII, 529 pages. 1993.

Vol. 693: P. E. Lauer (Ed.), Functional Programming, Concurrency, Simulation and Automated Reasoning. Proceedings, 1991/1992. XI, 398 pages. 1993.

Vol. 694: A. Bode, M. Reeve, G. Wolf (Eds.), PARLE '93. Parallel Architectures and Languages Europe. Proceedings, 1993. XVII, 770 pages. 1993.

Vol. 695: E. P. Klement, W. Slany (Eds.), Fuzzy Logic in Artificial Intelligence. Proceedings, 1993. VIII, 192 pages. 1993. (Subseries LNAI).

Vol. 696: M. Worboys, A. F. Grundy (Eds.), Advances in Databases. Proceedings, 1993. X, 276 pages. 1993.

Vol. 697: C. Courcoubetis (Ed.), Computer Aided Verification. Proceedings, 1993. IX, 504 pages. 1993.

Vol. 698: A. Voronkov (Ed.), Logic Programming and Automated Reasoning. Proceedings, 1993. XIII, 386 pages. 1993. (Subseries LNAI).

Vol. 699: G. W. Mineau, B. Moulin, J. F. Sowa (Eds.), Conceptual Graphs for Knowledge Representation. Proceedings, 1993. IX, 451 pages. 1993. (Subseries LNAI).

Vol. 700: A. Lingas, R. Karlsson, S. Carlsson (Eds.), Automata, Languages and Programming. Proceedings, 1993. XII, 697 pages. 1993.

Vol. 701: P. Atzeni (Ed.), LOGIDATA+: Deductive Databases with Complex Objects. VIII, 273 pages. 1993.

Vol. 702: E. Börger, G. Jäger, H. Kleine Büning, S. Martini, M. M. Richter (Eds.), Computer Science Logic. Proceedings, 1992. VIII, 439 pages. 1993.

Vol. 703: M. de Berg, Ray Shooting, Depth Orders and Hidden Surface Removal. X, 201 pages. 1993.

Vol. 704: F. N. Paulisch, The Design of an Extendible Graph Editor. XV, 184 pages. 1993.

Vol. 705: H. Grünbacher, R. W. Hartenstein (Eds.), Field-Programmable Gate Arrays. Proceedings, 1992. VIII, 218 pages. 1993.

Vol. 706: H. D. Rombach, V. R. Basili, R. W. Selby (Eds.), Experimental Software Engineering Issues. Proceedings, 1992. XVIII, 261 pages. 1993.

Vol. 707: O. M. Nierstrasz (Ed.), ECOOP '93 – Object-Oriented Programming. Proceedings, 1993. XI, 531 pages. 1993.

Vol. 708: C. Laugier (Ed.), Geometric Reasoning for Perception and Action. Proceedings, 1991. VIII, 281 pages. 1993.

Vol. 709: F. Dehne, J.-R. Sack, N. Santoro, S. Whitesides (Eds.), Algorithms and Data Structures. Proceedings, 1993. XII, 634 pages. 1993.

Vol. 710: Z. Ésik (Ed.), Fundamentals of Computation Theory. Proceedings, 1993. IX, 471 pages. 1993.

Vol. 711: A. M. Borzyszkowski, S. Sokołowski (Eds.), Mathematical Foundations of Computer Science 1993. Proceedings, 1993. XIII, 782 pages. 1993.

Vol. 712: P. V. Rangan (Ed.), Network and Operating System Support for Digital Audio and Video. Proceedings, 1992. X, 416 pages. 1993.

Vol. 713: G. Gottlob, A. Leitsch, D. Mundici (Eds.), Computational Logic and Proof Theory. Proceedings, 1993. XI, 348 pages. 1993.

Vol. 714: M. Bruynooghe, J. Penjam (Eds.), Programming Language Implementation and Logic Programming. Proceedings, 1993. XI, 421 pages. 1993.

Vol. 715: E. Best (Ed.), CONCUR'93. Proceedings, 1993. IX, 541 pages. 1993.

Vol. 716: A. U. Frank, I. Campari (Eds.), Spatial Information Theory. Proceedings, 1993. XI, 478 pages. 1993.

Vol. 717: I. Sommerville, M. Paul (Eds.), Software Engineering – ESEC '93. Proceedings, 1993. XII, 516 pages. 1993.

Vol. 718: J. Seberry, Y. Zheng (Eds.), Advances in Cryptology – AUSCRYPT '92. Proceedings, 1992. XIII, 543 pages. 1993.

Vol. 719: D. Chetverikov, W.G. Kropatsch (Eds.), Computer Analysis of Images and Patterns. Proceedings, 1993. XVI, 857 pages. 1993.

Vol. 720: V.Mařík, J. Lažanský, R.R. Wagner (Eds.), Database and Expert Systems Applications. Proceedings, 1993. XV, 768 pages. 1993.

Vol. 721: J. Fitch (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1992. VIII, 215 pages. 1993.

Vol. 722: A. Miola (Ed.), Design and Implementation of Symbolic Computation Systems. Proceedings, 1993. XII, 384 pages. 1993.

Vol. 723: N. Aussenac, G. Boy, B. Gaines, M. Linster, J.-G. Ganascia, Y. Kodratoff (Eds.), Knowledge Acquisition for Knowledge-Based Systems. Proceedings, 1993. XIII, 446 pages. 1993. (Subseries LNAI).

Vol. 724: P. Cousot, M. Falaschi, G. Filè, A. Rauzy (Eds.), Static Analysis. Proceedings, 1993. IX, 283 pages. 1993.

Vol. 725: A. Schiper (Ed.), Distributed Algorithms. Proceedings, 1993. VIII, 325 pages. 1993.

Vol. 726: T. Lengauer (Ed.), Algorithms – ESA '93. Proceedings, 1993. IX, 419 pages. 1993

Vol. 727: M. Filgueiras, L. Damas (Eds.), Progress in Artificial Intelligence. Proceedings, 1993. X, 362 pages. 1993. (Subseries LNAI).

Vol. 728: P. Torasso (Ed.), Advances in Artificial Intelligence. Proceedings, 1993. XI, 336 pages. 1993. (Subseries LNAI).

Vol. 729: L. Donatiello, R. Nelson (Eds.), Performance Evaluation of Computer and Communication Systems. Proceedings, 1993. VIII, 675 pages. 1993.

Vol. 730: D. B. Lomet (Ed.), Foundations of Data Organization and Algorithms. Proceedings, 1993. XII, 412 pages. 1993.

Vol. 731: A. Schill (Ed.), DCE – The OSF Distributed Computing Environment. Proceedings, 1993. VIII, 285 pages. 1993.

Vol. 732: A. Bode, M. Dal Cin (Eds.), Parallel Computer Architectures. IX, 311 pages. 1993.

Vol. 733: Th. Grechenig, M. Tscheligi (Eds.), Human Computer Interaction. Proceedings, 1993. XIV, 450 pages. 1993.

Vol. 734: J. Volkert (Ed.), Parallel Computation. Proceedings, 1993. VIII, 248 pages. 1993.

Vol. 735: D. Bjørner, M. Broy, I. V. Pottosin (Eds.), Formal Methods in Programming and Their Applications. Proceedings, 1993. IX, 434 pages. 1993.

Vol. 736: R. L. Grossman, A. Nerode, A. P. Ravn, H. Rischel (Eds.), Hybrid Systems. VIII, 474 pages. 1993.

Vol. 737: J. Calmet, J. A. Campbell (Eds.), Artificial Intelligence and Symbolic Mathematical Computing. Proceedings, 1992. VIII, 305 pages. 1993.

Vol. 738: M. Weber, M. Simons, Ch. Lafontaine, The Generic Development Language Deva. XI, 246 pages. 1993.

Vol. 739: H. Imai, R. L. Rivest, T. Matsumoto (Eds.), Advances in Cryptology – ASIACRYPT '91. X, 499 pages. 1993.

Vol. 740: E. F. Brickell (Ed.), Advances in Cryptology – CRYPTO '92. Proceedings, 1992. X, 593 pages. 1993.

Vol. 741: B. Preneel, R. Govaerts, J. Vandewalle (Eds.), Computer Security and Industrial Cryptography. Proceedings, 1991. VIII, 275 pages. 1993.

Vol. 742: S. Nishio, A. Yonezawa (Eds.), Object Technologies for Advanced Software. Proceedings, 1993. X, 543 pages. 1993.

Vol. 743: S. Doshita, K. Furukawa, K. P. Jantke, T. Nishida (Eds.), Algorithmic Learning Theory. Proceedings, 1992. X, 260 pages. 1993. (Subseries LNAI)

Vol. 744: K. P. Jantke, T. Yokomori, S. Kobayashi, E. Tomita (Eds.), Algorithmic Learning Theory. Proceedings, 1993. XI, 423 pages. 1993. (Subseries LNAI)

Vol. 745: V. Roberto (Ed.), Intelligent Perceptual Systems. VIII, 378 pages. 1993. (Subseries LNAI)

Vol. 746: A. S. Tanguiane, Artificial Perception and Music Recognition. XV, 210 pages. 1993. (Subseries LNAI).

Vol. 747: M. Clarke, R. Kruse, S. Moral (Eds.), Symbolic and Quantitative Approaches to Reasoning and Uncertainty. Proceedings, 1993. X, 390 pages. 1993.

Vol. 748: R. H. Halstead Jr., T. Ito (Eds.), Parallel Symbolic Computing: Languages, Systems, and Applications. Proceedings, 1992. X, 419 pages. 1993.

Vol. 749: P. A. Fritzson (Ed.), Automated and Algorithmic Debugging. Proceedings, 1993. VIII, 369 pages. 1993.

Vol. 750: J. L. Díaz-Herrera (Ed.), Software Engineering Education. Proceedings, 1994. XII, 601 pages. 1994.

Vol. 751: B. Jähne, Spatio-Temporal Image Processing. XII, 208 pages. 1993.

Vol. 752: T. W. Finin, C. K. Nicholas, Y. Yesha (Eds.), Information and Knowledge Management. Proceedings, 1992. VII, 142 pages. 1993.

Vol. 753: L. J. Bass, J. Gornostaev, C. Unger (Eds.), Human-Computer Interaction. Proceedings, 1993. X, 388 pages. 1993.

Vol. 754: H. D. Pfeiffer, T. E. Nagle (Eds.), Conceptual Structures: Theory and Implementation. Proceedings, 1992. IX, 327 pages. 1993. (Subseries LNAI).

Vol. 755: B. Möller, H. Partsch, S. Schuman (Eds.), Formal Program Development. Proceedings. VII, 371 pages. 1993.

Vol. 756: J. Pieprzyk, B. Sadeghiyan, Design of Hashing Algorithms. XV, 194 pages. 1993.

Vol. 757: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1992. X, 576 pages. 1993.

Vol. 758: M. Teillaud, Towards Dynamic Randomized Algorithms in Computational Geometry. IX, 157 pages. 1993.

Vol. 759: N. R. Adam, B. K. Bhargava (Eds.), Advanced Database Systems. XV, 451 pages. 1993.

Vol. 760: S. Ceri, K. Tanaka, S. Tsur (Eds.), Deductive and Object-Oriented Databases. Proceedings, 1993. XII, 488 pages. 1993.

Vol. 761: R. K. Shyamasundar (Ed.), Foundations of Software Technology and Theoretical Computer Science. Proceedings, 1993. XIV, 456 pages. 1993.

Vol. 762: K. W. Ng, P. Raghavan, N. V. Balasubramanian, F. Y. L. Chin (Eds.), Algorithms and Computation. Proceedings, 1993. XIII, 542 pages. 1993.

Vol. 763: F. Pichler, R. Moreno Díaz (Eds.), Computer Aided Systems Theory – EUROCAST '93. Proceedings, 1993. IX, 451 pages. 1994.

Vol. 764: G. Wagner, Vivid Logic. XII, 148 pages. 1994. (Subseries LNAI).

Vol. 765: T. Helleseth (Ed.), Advances in Cryptology – EUROCRYPT '93. Proceedings, 1993. X, 467 pages. 1994.

Vol. 766: P. R. Van Loocke, The Dynamics of Concepts. XI, 340 pages. 1994. (Subseries LNAI).

Vol. 767: M. Gogolla, An Extended Entity-Relationship Model. X, 136 pages. 1994.

Vol. 768: U. Banerjee, D. Gelernter, A. Nicolau, D. Padua (Eds.), Languages and Compilers for Parallel Computing. Proceedings, 1993. XI, 655 pages. 1994.

80.

Contents

Compilation of a Highly Parallel Actor-Based Language 1
W. Kim and G. Agha, *University of Illinois at Urbana-Champaign*

A Concurrent Execution Semantics for Parallel Program Graphs 16
and Program Dependence Graphs
V. Sarkar, *IBM Palo Alto Scientific Center, Palo Alto, California*

Using Profile Information to Assist Advanced Compiler 31
Optimization and Scheduling
W. Chen, R. Bringmann, S. Mahlke, S. Anik, T. Kiyohara, N. Warter, D. Lavery, W.-M. Hwu, R. Hank and J. Gyllenhaal
University of Illinois at Urbana-Champaign

A Hierarchical Parallelizing Compiler for VLIW/MIMD Machines 49
C. Brownhill and A. Nicolau, *University of California at Irvine*

Dynamic Dependence Analysis: A Novel Method for Data 64
Dependence Evaluation
P. Peterson and D. Padua, *University of Illinois at Urbana-Champaign*

On the Feasibility of Dynamic Partitioning of Pointer Structures 82
J. Solworth, *University of Illinois at Chicago*

Compiler Analysis for Irregular Problems in Fortran D 97
R. von Hanxleden, K. Kennedy, and C. Koelbel
Rice University, Houston, Texas
R. Das and J. Saltz
University of Maryland, College Park, Maryland

Data Ensembles in Orca C 112
C. Lin and L. Snyder, *University of Washington at Seattle*

Compositional C++: Compositional Parallel Programming 124
K. Mani Chandy and C. Kesselman
California Institute of Technology at Pasadena

Data Parallelism and Linda 145
N. Carriero and D. Gelernter
Yale University, New Haven, Connecticut

Techniques for Efficient Execution of Fine-Grained Concurrent 160
Programs
A. Chien, W. Feng, V. Karamcheti, and J. Plevyak
University of Illinois at Urbana-Champaign

Computing Per-Process Summary Side-Effect Information 175
T. Jeremiassen and S. Eggers, *University of Washington at Seattle*

Supporting SPMD Execution for Dynamic Data Structures 192
A. Rogers, *Princeton University, Princeton, New Jersey*
J. Reppy, *AT&T Bell Labs*
L. Hendren, *McGill University, Montréal, Québec, Canada*

Determining Transformation Sequences for Loop Parallelization 208
W. Appelbe, *Georgia Institute of Technology, Atlanta, Georgia*
K. Smith, *Emory University, Atlanta, Georgia*

Compiler Optimizations for Massively Parallel Machines: 223
Transformations on Iterative Spatial Loops
M. Chen and Y. Hu, *Yale University, New Haven, Connecticut*

Handling Distributed Data in Vienna Fortran Procedures 248
B. Chapman and H. Zima, *University of Vienna, Vienna, Austria*
P. Mehrotra, *ICASE, Hampton, Virginia*

On the Synthesis of Parallel Programs from Tensor Product 264
Formulas for Block Recursive Algorithms
S. Gupta, C.-H. Huang, and P. Sadayappan
The Ohio State University, Columbus, Ohio
R. Johnson, *St. Cloud State University, St. Cloud, Minnesota*

Collective Loop Fusion for Array Contraction 281
G. Gao and R. Olsen, *McGill University, Montréal, Québec, Canada*
V. Sarkar, *IBM Palo Alto Scientific Center, Palo Alto, California*
R. Thekkath, *University of Washington at Seattle*

Parallel Hybrid Data Flow Algorithms: A Case Study 296
Y.-F. Lee and B. Ryder
Rutgers University, New Brunswick, New Jersey

A Control-Parallel Programming Model Implemented on SIMD Hardware 311
H. Dietz and W. Cohen, *Purdue University, West Lafayette, Indiana*

C**: A Large-Grain, Object-Oriented, Data-Parallel Programming Language ... 326
J. Larus, *University of Wisconsin at Madison*

A Calculus of Gamma Programs 342
C. Hankin and D. LeMétayer
Imperial College of Science, Technology and Medicine, London, UK
D. Sands, *University of Copenhagen, Copenhagen, Denmark*

A Linda-Based Runtime System for a Distributed Logic Language..... 356
P. Ciancarini, *University of Bologna, Bologna, Italy*

Parallelizing a C Dialect for Distributed Memory MIMD Machines 369
O. Lempel, S. Pinter,and E. Turiel
Technion — Israel Institute of Technology, Haifa, Israel

A Singular Loop Transformation Framework Based on Non-Singular Matrices .. 391
W. Li and K. Pingali, *Cornell University, Ithaca, New York*

Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations 406
L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan
McGill University, Montréal, Québec, Canada

Doany: Not Just Another Parallel Loop 421
M. Wolfe
Oregon Graduate Institute of Science & Technology
Beaverton, Oregon

Data Dependence and Data-Flow Analysis of Arrays 434
D. Maydan, S. Amarsinghe, and M. Lam
Stanford University, Stanford, California

Experience with Techniques for Refining Data Race Detection 449
R. Netzer, *Brown University, Providence, Rhode Island*
B. Miller, *University of Wisconsin at Madison*

Extending the Banerjee-Wolfe Test to Handle Execution Conditions .. 464
D. Klappholz, *Stevens Institute of Technology, Hoboken, New Jersey*
X. Kong, *Sun Microsystems, Inc., Mountain View, California*

A FORTRAN Compiling Method for Dataflow Machines and Its 482
Prototype Compiler for the Parallel Processing System -Harray-
T. Yasue, H. Yamana, and Y. Muraoka
Waseda University, Tokyo, Japan

Distributed Slicing and Partial Re-execution for Distributed 497
Programs
E. Duesterwald, R. Gupta and M. Soffa
University of Pittsburgh, Pittsburgh, Pennsylvania

A Program's Eye View of Miprac 512
W. Harrison III and Z. Ammarguellat
University of Illinois at Urbana-Champaign

Symbolic Program Analysis and Optimization for Parallelizing 538
Compilers
M. Haghighat and C. Polychronopoulos
University of Illinois at Urbana-Champaign

Utilizing New Communication Features in Compilation for 563
Private-Memory Machines
S. Hinrichs and T. Gross
Carnegie Mellon University, Pittsburgh, Pennsylvania

1 Compilation of a Highly Parallel Actor-Based Language

W. Kim and G. Agha
University of Illinois at Urbana-Champaign

Abstract
HAL is a High-level Actor-based Language. HAL supports a number of communication mechanisms, local synchronization constraints, inheritance, and restricted forms of reflection. This paper discusses some issues in compiling HAL. Specifically, we describe three source-level transformations used by the compiler for HAL. Two of the transformations translate RPC-style message sending into asynchronous message sending. The third transformation performs code motion to optimize the implementation of replacement behavior. This optimization results in the reduction of object code size as well as execution time.

Keywords: Actor, concurrency, synchronization constraint, inheritance, optimization

1 Introduction

As multicomputer architectures have become more prevalent, an increasing number of research efforts have focused on designing languages or building compilers to efficiently use multicomputers. These efforts are inspired by different programming models. For example, one kind of effort, based on the SPMD model, has focused on extracting data parallelism out of programs written in sequential programming languages such as FORTRAN and C. These efforts focus on compiler techniques such as data dependence analysis, loop transformation and/or data distribution [8].

Although compilers built on the SPMD model have been quite successful in utilizing data parallelism inherent in many programs, they fail to benefit from the control parallelism inherent in algorithms and programs: some nodes may not be able to do useful work although they could execute tasks different from other nodes.

Another approach to implicit parallel programming is based on using new programming languages rather than developing compiling techniques for existing sequential languages. Functional programming is an example of this approach. Unfortunately, functional languages are not capable of modelling concurrency in a state-based, nondeterministic world.

We use the Actor model which unifies the functional approach with object-based concurrency [1, 2]. Actors support both control and data parallelism inherent in algorithms themselves and may be used to naturally model a state-based non-deterministic world. We have developed a high-level actor-based language called HAL [9]. In particular, HAL supports concurrent object-oriented programming.

Actors are self contained, independent computational agents that communicate by asynchronous message sending. Each actor consists of its mail queue and behavior and is associated with a unique mail address. The mail queue of an actor buffers incoming communications (i.e. messages). The behavior of an actor specifies an action performed by the actor in response to each communication and comprises a persistent state and a set of method definitions. An actor's state is defined by its *acquaintances* (actors whose mail addresses are known to the actor).

All computation in an actor system is carried out in response to communications sent to actors in the system. Specifically, an actor may perform three kinds of actions when it accepts a message:

- It may change its behavior.
- It may send communications to its acquaintance actors asynchronously.
- It may create new actors.

The replacement behavior of an actor is the behavior with which the actor responds to the next message it processes. It is specified by using the `become` primitive. Whenever there is no executable `become` primitive in the thread of an actor computation, an identically behaving actor is assumed to be its replacement behavior (by default). Note that communications may contain mail addresses of actors; thus the interconnection topology of an actor system is dynamic. Delivery of a message is guaranteed after an arbitrary, but finite, delay (a fairness condition [5]).

This paper describes ongoing work on HAL and its compiler. In the following section, we describe some linguistic constructs of HAL with their semantics, emphasizing their implementation issues. Section 3 discusses the transformations of RPC style message sending and the transformation for code motion to optimize the implementation of replacement behavior. The last section provides future research directions and concluding remarks.

2 HAL: A High-level Actor Language

We briefly describe some constructs provided in HAL. First, we discuss how local synchronization constraints are specified in HAL. We then discuss inheritance, replacement behavior and some other programming constructs.

2.1 Specification of Synchronization Constraints

Consider a system in which a *producer* actor and a *consumer* actor collaborate through a bounded buffer actor that has `get` and `put` methods. The producer invokes the `put` method to store in the buffer a value it has generated. The consumer retrieves the next available value by sending a `get` message to the buffer actor. Sending a `put` message to a full buffer may result in loss of a generated value. Processing an invocation of `get` method by an empty buffer actor may not be desirable.

Synchronization constraints are programming constructs that specify the subset of possible states of an object under which that object's services may be invoked. By making explicit the *enabling* or *disabling* conditions for each method, the constraints can guarantee data consistency on a per object basis in a concurrent system. To be smoothly unified with inheritance, synchronization constraints need to be specified on per class basis and as a part of class description which is separated from its method definitions. Synchronization constraints can be specified using the following syntax:

```
(restrict  <msg-expr> <bool-expr>)
(extend    <msg-expr> <bool-expr>)
(overwrite <msg-expr> <bool-expr>)
(only      <msg-expr> <bool-expr>)
```

where `msg-expr` evaluates to a method name and `<bool-expr>` defines the condition under which the method may be invoked. In an actor not defined using a superclass, only the operators `restrict` and `only` may be used. In the first case, the set of enabling states in which the specified method may be invoked is restricted to those satisfying the given condition. `only` specifies that only the method specified in `<msg-expr>` can be processed as long as `<bool-expr>` evaluates `true`. These operators interact with inheritance to support the incremental modification of synchronization constraints. The `restrict` operator in a subclass specializes the constraint in the superclass – the method may be invoked only in those enabling states of the superclass which also satisfy the condition given in the subclass. On the other hand, `extend` adds states (generalizes) the set of enabling states of a method; the method may be invoked if either the synchronization condition in superclass actor or the subclass are satisfied. Finally, the operator `overwrite` is used to entirely redefine the set of enabling conditions for an inherited method.

2.2 Inheritance

As in Smalltalk [12], any method of the superclass of an actor may be redefined in the definition of the actor. As discussed above, the synchronization constraints of methods may be incrementally modified separately from the method definition.

Figure 1 illustrates the possible usage of inheritance and synchronization constraints. The most basic actor class is `Stack` which has the methods to push and pop one element to and from the stack, respectively. This definition has a constraint `(> count 0)` on `pop` which must hold in order for a pop message to be processed.

The `Bounded-stack` class is a subclass of `Stack`. Bounded stacks satisfy a constraint on the size of the stack; a bounded stack cannot contain more than `max` elements at any time. Because we have separated the specification of synchronization constraints from the code for a method, we only need to state the new constraint without redefining the `push` method. Note that more restrictions are imposed on the `push` method in the `Bounded-stack` class than in the `Stack` class.

Finally, we define the `Pop2-stack` class as a subclass of the `Bounded-stack` class. `Pop2-stack` atomically pops two elements out of the stack when the `pop` method is invoked. The constraint associated with `pop` method is redefined using `overwrite` in the `Pop2-stack` class. By using this constraint specification scheme, the definition of both `pop` and `push` methods can be inherited from the `Stack` class by its descendent classes without causing the so-called "inheritance anomaly" [14]. Our scheme is

```
(defActor Stack (stack count)
    (restrict (pop) (> count 0))
    (method (push x)
        (update stack (cons x stack))
        (update count (+ count 1)))
    (method (pop entrypoint continuation)
        (send entrypoint continuation (car stack))
        (update stack (cdr stack))
        (update count (- count 1))))

(defActor Bounded-stack (max)
    (superclass Stack)
    (restrict (push) (<= count max)))

(defActor Pop2-stack
    (superclass Bounded-stack)
    (overwrite (pop) (> count 1))
    (method (pop entrypoint continuation)
        (send entrypoint continuation (car stack) (car (cdr stack)))
        (update stack (cdr (cdr stack)))
        (update count (- count 2))))
```

Figure 1: A hierarchy of stack actor classes.

flexible; incremental specialization, generalization and redefinition of synchronization constraints can be naturally expressed. Others have argued that inheritance of synchronization constraints should only support specialization [7] or generalization [15].

2.3 State Change

Local state change is specified by the `become` primitive in the Actor model. An actor may *become* an entirely different actor. In this case a new behavior definition and acquaintances need to be specified. However, in many cases, replacement behavior only involves change of one or two acquaintance(s) of an actor. The `update` primitive is a primitive used to specify that the replacement behavior is identical to the original behavior except for the change in the acquaintance specified in the statement. The syntax of update is as follows:

```
(update <acquaintance-name> <expr>)
```

Note that `update` conforms to a single-assignment semantics; the same acquaintance cannot appear in more than one update statement in any control flow of a method.

However, a method can have more than one `update` operation provided that the operations are applied to different acquaintances. Since the effect of replacement behavior is invisible to the current computation of an actor, multiple updates are semantically equivalent to a `become` with multiple acquaintance changes.

2.4 Message-passing Mechanisms

Besides the asynchronous message sending provided as a primitive by the Actor model, HAL provides two more message send constructs: `ssend` and `bsend`. The former is the message order preserving send primitive, or sequenced send. The latter is the RPC style message send primitive akin to Acore's `ask` primitive [13]. Although the two constructs provide some of the functionalities of synchronous message passing, they do *not* require synchronous communication – i.e., a sender does not need to wait until a receiver is ready to accept its message.

A sequenced send gives a sender the ability to ensure that messages to a given sender are received in the same order in which they are sent. Sequenced sends are implemented by tagging and, when necessary, reordering the messages at the recipient's end. The operator `bsend` is similar to a remote procedure call; the remainder of the sending computation proceeds once a reply is received. The implementation issues related to `bsend` are given in detail in Section 3.2.

2.5 Suicide

When executing a large actor program, many actors are typically created. Some of these actors will never be used again once they have accepted certain communications – they become garbage actors which waste space (and possibly other) resources until they are reclaimed by garbage collector. The operator, `suicide`, allows an actor to deallocate all resources it is using. Note that `suicide` is *unsafe* and is meant for use only by the compiler or in meta-programming.

2.6 Reflection

Reflection is a system's ability to reason and manipulate a causally connected description of itself [16]. A description is said to be causally connected to the object it describes if changes to the description have an immediate effect on the object. Currently, HAL provides the minimal support for reflection that is necessary to allow a system to be customized with respect to fault tolerance [3]. An actor can reify its dis-