Abstract. This paper describes a method to implement the functionality of shared passive packages on top of a logical distributed memory — Linda. From a shared passive package a compiler can construct a new normal package that replaces the shared passive package. The new package contains the same subprograms and is extended with abstract data structures mapping Ada objects onto the storage units of Linda. A short program example is included to illustrate the construction process.

1. Introduction

The Ada 95 standard, RM95 [1], defines a distributed Ada program as a number of partitions working cooperatively. The partitions can be mapped on one or more processing nodes. The standard also distinguishes between active and passive partitions; an active partition being loosely equivalent to an Ada 83 program, and a passive partition being a collection of subprograms and global data that are shared among active partitions. Active partitions cooperate by calling subprograms in other active partitions through remote call interfaces, or by reading and updating variables in shared passive packages residing in passive partitions.

The restrictions imposed on shared passive packages by the standard suggest that they are intended for use only when partitions share a logical address space [2]. This can be achieved either by the use of shared physical memory, where the logical and physical address space are identical, or through distributed shared virtual memory (DSVM). An example of distributed shared virtual memory for Ada can be found in [3]. On a system not having DSVM or on a heterogeneous system it becomes practically impossible to use shared passive packages, due to different data representation and different instruction sets.

Even in the absence of true shared memory it is desirable to have access to shared passive packages. It offers an uncoupled way for processes to communicate. There are distributed algorithms that uses shared variables, e.g. the Bakery algorithm for mutual exclusion [4]. Shared passive packages also offers a simple mutual exclusion policy through shared protected objects. Also things like mode changes can easily be signaled through shared variables. One solution in the absence of DSVM is to use a higher level shared memory that provides a shared logical address space to implement shared passive packages.

Linda [5] is a language independent model for concurrent and distributed programming. The model introduces a few simple operators on a logical memory called Tuple Space (TS) that can be hosted by an existing language, e.g. Ada 95, without changing the semantics of the host language. There are many implementations of Linda and Lin-
da-like systems [6] so the model is well tested and spread. An example of a Linda implementation for Ada 83 is Ada-Linda [7].

In [5] it is suggested that Linda could be regarded as a machine language for a Linda machine, and that high level language constructs could be compiled into Linda. For example, operations on objects declared in Ada 95 shared passive packages could be compiled into Linda operations.

In this paper we show how the use of shared passive packages can be extended to architectures without (physical/virtual) shared memory among partitions. This is done by implementing an approximation of shared passive packages on top of a Linda-based memory model. The paper is organized as follows: Section 2 describes the Linda model briefly. Section 3 and 4 contains the core of the paper and presents the implementation technique and discusses elaboration issues. Section 5 contains a hand compiled example that illustrates the implementation technique. Ideally this transformation should be done by a compiler or by a pre-processor. Finally section 6 concludes the paper.

2 The Linda Model

Linda is a model for concurrent programming that offers a logical distributed memory — the Tuple Space (TS). The storage unit of the TS is the logical Tuple, an ordered set of values, \((v_0, \ldots, v_N)\), called fields. Theoretically a tuple could have any number of fields. Instead of accessing tuples by address they are accessed by matching a template tuple to their structure in a way similar to how unification is done in e.g. Prolog.

2.1 Matching of Templates to Tuples

In a tuple all fields are values. For a template to match a tuple, the number of fields in the template and tuple must be the same, and all fields must match. Templates can have either actual or formal fields, where an actual field is a value and a formal field denotes any value of a given type. An actual field of a template matches any corresponding field of a tuple with the same value, and a formal field matches any corresponding field having the same type.

Example

The template \((S : \text{String}, 42, I : \text{Integer})\) matches any of the two tuples, \(("\text{foo"}, 42, 17)\) or \(("\text{bar"}, 42, 4711)\). However it would not match the tuple \(("\text{foo"}, 17, 4711)\) since the actual field 42 of the template does not match the second field 17 of the tuple.

2.2 Operations on Tuple Space

Instead of the normal read and write used to access a conventional memory Linda uses three primitive operations to access the tuple space: read, add and remove. As a consequence a tuple can not be modified in place which is a useful property for the implementation.

On top of these primitive operations Linda defines several operations on tuple space that are available to the programmer. For the scope of this paper only the operations Out, Rd, and In are of interest, see fig. 1, they correspond to the primitive operations with some additional semantics.