ELSEVIER

Advances in Engineering Software 30 (1999) 211–224

ADVANCES IN ENGINEERING SOFTWARE

Formal specification of concurrent systems

Harpreet S. Chadha[a], John W. Baugh Jr[b,*], Jeannette M. Wing[c]

[a]*Make Systems Laboratories, 107A Fountain Brook Circle, Cary, NC 27511-4470, USA*
[b]*Department of Civil Engineering, North Carolina State University, Raleigh, NC 27695-7908, USA*
[c]*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA*

Abstract

This paper presents a formal methodology for developing concurrent systems. We extend the Larch family of specification languages and tools with the CCS process algebra to support the specification and verification of concurrent systems. We present and follow a refinement strategy that relates an implementation in a programming language to a formal specification of such a system. We illustrate our methodology on an example that uses the preconditioned conjugate gradient method for solving a linear system of equations.

Keywords: Larch; CCS; Equational specifications; Process algebra; Conjugate gradient method; Distributed systems; Concurrent systems; Programming languages; Formal methods

1. Introduction

Rapid advancements in information technology have led to a shift in the development of engineering systems that target stand-alone machines to those that exploit the immense computing and communication power of networks of interlinked processors. Development of such distributed systems is, however, considerably more complex; many researchers have suggested various tools, standards, and frameworks to simplify the task.

The precise nature of formal approaches, and the verification frameworks they offer, make them suitable for designing these complex systems. Formal approaches such as CSP [1] and CCS [2] are ideally suited for describing the behavior of concurrent processes. Formal approaches such as Z [3], VDM [4], and Larch [5] are ideally suited for describing state spaces (e.g. abstract data types) of sequential programs. Some approaches such as LOTOS [6] and RAISE [7] were designed to combine the two: the specification of concurrent processes and that of state spaces.

Many of these approaches support a way to relate to different specifications. For example, in CSP and CCS, one can show two different specifications actually describe equivalent concurrent systems, for some technical meaning of 'equivalence'. These and other approaches, e.g. Z and VDM, also support a different kind of relation between two different specifications—a notion of refinement:

Impl refines Spec

where a low-level specification, Impl, refines a higher-level specification, Spec. In a pure refinement process, one applies a sequence of correctness-preserving transformations to the high-level specification, Spec, to produce the lower-level specification, Impl. Alternatively, one can start with both the Spec and a purported Impl and then discharge a set of proof obligations to prove formally that the lower-level specification, Impl, is correct with respect to the higher-level one, Spec.[1]

Of the approaches mentioned, with one exception, none takes the notion of refinement all the way down to code written in a 'real' programming language like C or C++. The exception is Larch. In Larch, a specification describes individual program modules, e.g. a C function, a C++ class, an Ada package, or a Modula-3 module. Hence, the relationship between the Larch specification (Spec, earlier) and the program module (Impl, earlier) is very close. Larch adopts the proof obligation view of refinement rather than the transformational view.

However, no approach to date supports relating a high-level specification of a concurrent system to 'real' code. Thus, we fill this void by wedding aspects of all these approaches: we specify the CCS process algebra in the Larch equational framework and use the extended

[1] Often people incorrectly use the phrase 'Impl is correct.' Impl cannot be 'correct' by itself; it can be correct only with respect to something else, like Spec.

* Corresponding author.

0965-9978/99/$ - see front matter
PII: S0965-9978(98)00058-1

framework to specify and develop concurrent systems. This hybrid methodology facilitates specification of both the data abstractions and the control flow in these systems; moreover, it provides a strategy to refine these specifications into implementations.

Thus, we make two major contributions:

- We specify the data components of a system using Larch, a formal method specifically designed for describing data abstractions. We specify the control flow between the system's components using CCS, a formal method specifically designed for describing concurrent systems based on message-passing semantics.
- We propose and follow a strategy of program development that allows us to relate a high-level specification, i.e. written in Larch and CCS, to running code, i.e. written in C. Our strategy relies on extending the existing two-tiered approach of Larch with one that accommodates the specification of CCS processes.

We illustrate our hybrid approach on an engineering example: the preconditioned conjugate gradient method.

In Section 2, we first describe our extended verification framework for developing specifications in Larch. Section 3 then discusses our methodology for refining these specifications into implementations in a programming language. Subsequently, we present a case study using this methodology for sequential and distributed versions of the conjugate gradient method in Section 4. Section 5 discusses the maturity of Larch tools and Section 6 closes with some concluding remarks.

2. Verification frameworks

This work focuses on frameworks for equational specifications, which are convenient for describing abstract data types, and for process algebras, which are useful for describing concurrent systems. For brevity, we present only a short and informal overview of these areas, but enough to support the subsequent examples and discussion.

2.1. Equational specifications and larch

Equational specifications are convenient for describing abstract data types, and are formalized by theories that are defined in a restricted form of predicate logic with equality [8]. By 'formalized,' we mean that such specifications have precise and unambiguous semantics [9]. An equational specification defines a mathematical object by using equations to relate the operators defined for that object (operators simply map from a cross product of values to a single value). These equations are connected implicitly by logical conjunction, and each of the equations must be true over the domain of their variables. For example, a logical negation operator is implicitly defined by $not(not(x)) = x$, where x is a logical variable. Because equations may also be interpreted as left-to-right rewrite rules (under mild constraints), we have an operational mechanism for executing, and hence testing, our specifications.

The Larch two-tiered framework for specification of software and hardware modules separates the equational specification from the specification of a module's interface [5]. The equational theory (first tier) describes abstractions that are independent of any implementation language, and is written in the Larch shared language (LSL). The interface specification (second tier) describes actual software components in terms of their pre- and post-conditions, and includes implementation language-specific details such as error control and exception handling. Specific interface specification languages exist for various programming languages, e.g. C, C++, ML, Modula-3, Ada, and Smalltalk. The separation of concerns provided by the two-tiered approach allows equational theories to be reused far more easily than other kinds of computational objects. As discussed in the following sections, we introduce additional specifications to this two-tiered model to describe the control flow in distributed engineering applications.

2.2. Process algebra and CCS

Process algebras are a class of languages for specifying concurrency in communicating systems that can be translated into labeled transition systems. They define languages to specify communication among interacting processes, each built of actions and operators defined by the language. In the process algebra CCS, processes are built from a set of atomic actions A, which may be thought of as communication. Denoting the set of labels for these actions by Λ, a CCS action is either

- an input on $a \in \Lambda$ denoted by a; or
- an output on $a \in \Lambda$ denoted by $\bar{a}$; or
- an internal on $a \in \Lambda$ denoted by τ.

A process P is defined with the syntax

P	$:=$	*nil*	*termination*
	|	$a.P(a \in A)$	*pefixing*
	|	$P+P$	*external choice*
	|	$P\|P$	*parallel composition*
	|	$P\backslash L$	*restriction*
	|	$P[f](f{:}\Lambda - \Lambda)$	*relabeling*
	|	X	*process invocation*

The semantics are given operationally via inference rules that define the transitions available to CCS processes. Rules for each operator are given using the transition relation $\rightarrow \subseteq Proc \times A \times Proc$, where *Proc* is the set of processes. For example, one rule for *external choice*

$$\frac{p \rightarrow^{a} p'}{p + q \rightarrow^{a} p'}$$

states that if the process p can perform action a and become p', then the process $p+q$ can also perform an action a and become p'. Rules for other operators are similarly given.

```
ProcessAlgebraChoice(P, +, nil) : trait
    introduces
        __ + __ : P, P → P
        nil :→ P
    asserts
        ∀p, q, r : P
            p + q = q + p
            (p + q) + r = p + (q + r)
            p + p = p
            p + nil = p
    implies
        AC(+, P)
        ∀p : P
            nil + p = p
```

Fig. 1. External choice.

The operational semantics thus defined is used in building algorithms for equivalence testing of CCS specifications.

2.3. Extended Larch framework

In this study, we axiomatize in LSL laws describing the behavior of CCS processes. These laws provide a formal basis for concurrency in our algebraic framework. We take advantage of the sound and complete theories that exist for bisimilation equivalence and observational congruence (without recursion), and axiomatize laws needed to develop these theories; other lemmas and theorems can be added as needed for documentation or for guiding proofs.

We start with the laws for external choice (see Fig. 1). A trait is the basic unit of specification in LSL. The *ProcessAlgebraChoice* trait asserts that processes are commutative and associative for +, the external choice operator. It also asserts that the external choice between two identical processes ($p+p$) is equivalent to process p itself. Finally, it is asserted that the choice between a process p and the *nil* process is equivalent to the process p itself. The implies clause allows us to make claims of properties we expect to hold. In this example, we expect + to be associative and commutative (AC), and $nil+p$ to be equivalent to p. Such claims act as semantic checks for our specifications, providing a means to ensure that the specifications written have the desired properties. To prove these properties mechanically, we use the Larch prover (LP) [5], which gives an operational semantics to LSL by treating equations as rewrite rules. LP is a proof assistant for a subset of multi-sorted first-order logic, which is also the basis of the Larch languages. The availability of tools to convert LSL specifications into input for LP, and tools to ensure correct syntax, further help in the development of clear and precise specifications.

Fig. 2 includes properties for +, and introduces operators for *prefixing*,[2] the internal action τ, and the notion of when two actions can communicate with each other—*complement*(a) corresponds to $\bar{a}$ in the CCS syntax of Section 2.

[2] We use * instead of . for prefixing in LSL since the use of . is reserved. Similarly, \ for restriction and | for parrallel composition are reserved, and / and || are used instead, respectively.

The axiomatization of the *restriction* operator in Fig. 3 now uses theories developed in other traits:

- *ProcessAlgebraBasics* provides the operators * and complement;
- the *Set* trait allows modeling sets of (hidden) actions; and
- the *Conditional* overloads the if-then-else operator with process arguments.

The three axioms following the asserts clause completely model the semantics for restriction. The hypothesis in the implies clause provides a semantic check on the theory—the proof is carried out in LP:

```
### details deleted ###

LP3.15: prove
(~ (complement(b:A) = a∨b:A = a) ⇒ ((a*p)+(b:A*q))/
insert(a,{ }) = (b:A*(q / insert(a, { }))))
..
Suspending proof of conjecture ProcessAlgebraRestric-
tionTheorem.1
LP3.16: resume by ⇒
Level 2 subgoal for proof of ⇒
[] Proved by normalization
Conjecture ProcessAlgebraRestrictionTheorem.1
[] Proved ⇒ .
LP3.17: qed
All conjectures have been proved.
```

Note that we are using the rewriting mechanism in LP to check our equational specifications in LSL. Proofs of equality between two processes can also be carried out using the mechanism for checking bisimulation equivalence [10]—these mechanisms exploit the operational semantics of CCS discussed in Section 2. In this study, we exploit both of these proof strategies—rewriting for LSL specifications, and bisimulation for CCS specifications.

The axioms for other operators are written and checked in the same fashion [11]. Together, the laws define a sound and complete theory for strong bisimulation (see Fig. 4) and observational equivalence (see Fig. 5) without recursion. Similar approaches using rewriting as the operational mechanism for non-recursive equational theories are taken by various tools for algebraic manipulation of process

```
ProcessAlgebraBasics(P, A) : trait
    includes
        ProcessAlgebraChoice(P, +, nil)
    introduces
        __ * __ : A, P → P
        tau :→ A
        not_tau : A → Bool
        complement : A → A
        is_complement : A, A → Bool
    asserts
        ∀a : A
            not_tau(a) ⇒ is_complement(a, complement(a))
```

Fig. 2. Basic CCS operations.

algebras [12–14]. Proofs involving recursive processes in these tools are guided using induction [2]—similar extensions for recursive processes can be added to our verification framework, if required.

3. Design methodology

We present a strategy for developing software from a high-level specification to a low-level implementation. We utilize specifications written in the extended Larch framework; thus, our implementations would be written the programming language, such as C and C++, that our interfaces target. We use a four-layered framework for software specification (see Fig. 6) with the upper two layers consisting of abstract descriptions of data components, and the lower two layers providing the descriptions of control flow. We first describe our development strategy, followed by a discussion of its advantages and limitations using a simple example.

3.1. Specification framework

Implementations in a programming language such as C generally exploit procedural abstraction to structure sets of statements into procedures. In contrast to dealing with low-level statements, this abstraction allows reasoning about programs at a more abstract procedural level. Thus, we raise our level of reasoning from the statement level to the program module.

The user interface to the program further consists of a small fraction of these procedural interfaces, i.e. C prototypes, where it provides only those hooks into the program that a user needs to see; the user needs not understand the low-level implementation details at all. The Larch two-tiered model suggests the use of formal specification of this interface. These two tiers form the upper two layers in our specification framework:

Layer I is programming-language independent, and is described in LSL; while

Layer II describes language-specific details in a Larch

```
ProcessAlgebraRestriction : trait
    includes
        ProcessAlgebraBasics,
        Set(A, ActionSet),
        Conditional(P)
    introduces
        __/__ : P, ActionSet → P
    asserts
        ∀a : A, p, q : P, l : ActionSet
            nil/l == nil
            (a * p)/l == if (a ∈ l ∨ complement(a) ∈ l) then nil
                else a * (p/l)
            (p + q)/l == (p/l) + (q/l)
    implies
        ∀a, b : A, p, q : P
            ¬(complement(b) = a ∨ b = a) ⇒
                ((a * p) + (b * q))/insert(a, {}) = b * (q/insert(a, {}))
```

Fig. 3. Restriction.

```
ProcessAlgebraStrongBisim : trait
    includes
        ProcessAlgebraBasics,
        ProcessAlgebraRestriction,
        ProcessAlgebraRelabel,
        ProcessAlgebraParallel
```

Fig. 4. Strong bisimulation.

```
ProcessAlgebraObCong : trait
    includes
        ProcessAlgebraStrongBisim,
        ProcessAlgebraTau
```

Fig. 5. Observational congruence.

interface language, such as the Larch/C interface language (LCL) for C [15].

While the Larch two-tiered specification model (I and II) nicely captures interface information [16] it does not capture how the modules specified by these interfaces get used together. For example, a set of three interfaces might include one that specifies an initialization procedure; a second, a clean-up procedure; and the third, a procedure that does some interesting work. There is no way in Larch to say explicitly that the intended use of these procedures is to first call the initialization procedure, then the work procedure, and finally the clean-up procedure.

We provide the functionality of specifying control flow through two additional layers in the Larch model:

Layer III utilizes our CCS extension of Larch to describe control flows in the program. Names of actions and processes correspond to names of procedures specified in Layers II and IV. Thus, a Layer III specification is essentially the specification of a CCS process, but written in our Larch notation introduced in Section 2 for CCS processes.

Layer IV specifies some auxiliary procedures that are not part of the interface visible to the user, but are useful to describe the control flow of the program, as needed by Layer III. It uses the same interface language as Layer II.

Our software development strategy supports this process of refinement: at the top, Layers I and II contain an abstract description of the user's view of the program; then, Layers III and IV add a level of detail by specifying details of control flow; finally, the actual C implementation adds the lowest level kind of detail. As will be evident, not all four layers need to be used in practice: in some cases, the high-level specification is sufficient; in some others, Layer III might suffice if it names only procedures from standard libraries like *stdio.h*.

For a concurrent system consisting of one client distributing tasks to several servers, Layers I and II can simply specify that the system has several processes (clients and servers) executing in parallel along with high-level information such as user-input and expected-output from these processes. For users interested in low-level details, Layers III and IV would further help understand the control-flow and implementation-specific features in the concurrent system.

Using a variety of available tools, we can check the artifacts produced in the course of our development strategy. While some of these tools have been discussed above, we include them here to show the corresponding layers in our specification framework on which they can be used:

LSL. The Larch shared language checker is a syntax and type checker for LSL (Layers I and III).
LP. The Larch prover allows us to make semantic checks on the theories developed with LSL specifications (Layers I and III).
LCLint. This tool uses LCL specifications, which can include LSL traits, to specify and check C programs (all layers, including the implementation).
Concurrency workbench. This tool allows us to test the equivalence of control flow descriptions in Layer III.
Standard C tools such as dbx and lint are useful for debugging the implementations.

3.2. A simple example

Consider the problem of implementing in C a function *sumsquare* that takes two complex numbers as input, and returns the square of the square of their sum as the result. In our four-layered framework, we build the following:

Layer I. A LSL description of complex numbers which includes properties of floating point numbers.
Layer II. An LCL description of the interface to the function *sumsquare*.
Layer III. An LSL description of the control flow in the implementation. In this example, the square of products may be implemented as calls to two procedures *sum* and

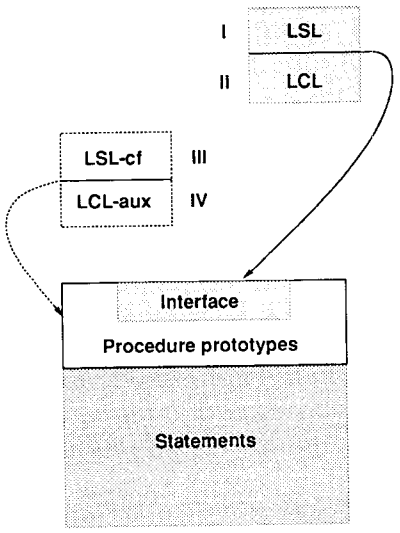

Fig. 6. Layers.

square which take complex numbers as formal parameters; the control flow for the process is given as *sumSquare* = = *sum*square*nil*.

Layer IV. LCL descriptions of procedures implemented in C, but not specified in the interface specifications, e.g. *sum* and *square*. These auxiliary specifications are put in a separate file, which is referred to in the interface specification.

Note, that while clients of the implementation module will generally overlook the additional specifications (III, IV), re-implementors will find them useful in understanding the implementation. Such detailed formal specifications, however, are neither required nor useful for this small problem, and are given only to describe our framework.

3.3. Why the extra work?

The primary motivation for this study is to provide a development strategy to transform formal specifications into code in a programming language. The use of a formal approach benefits both maintenance and reuse, and instills greater confidence in the systems developed [17, 18]. Further, our approach provides a framework for coordinating various specification and programming languages, whether based on temporal logic [19] or process algebra to specify concurrency, or C or C++ to build an implementation. In addition, Larch interface languages exist for other languages, such as C++ and ML, and can be used to specify implementation-specific details of modules developed using them. While various other approaches have been suggested to promote the reuse of the software systems [20] and to integrate implementations in different languages [21], they generally rely on informal specifications, and lack the precision and abstract quality of formal methods.

Our layered approach also makes it easier to use and reuse both formal specifications and the modules they specify. Modular use and reuse is especially important in a software development environment consisting of users with varying needs and sophistication. For example, the additional specifications are useful for re-implementors who find it difficult to understand the control flow linking procedures in the implementation. For a client who does not need the low-level details, however, the interface specifications are sufficient to reuse implementations. The methodology suggested can also be refined with graphical user interfaces to further simplify cooperative development of code in a large group. Since all users in such an environment are not as sophisticated, team leaders may be given the task to develop and maintain the specifications of modules. Changes, such as additional functionality of a module, may then first be referred to the respective team leader to ensure that specifications remain consistent after the modifications. Further, in a scenario where the original developer is unavailable, it is relatively easier to understand the specifications and make appropriate modifications, than it would be to modify unspecified code.

The use of CCS to model the control flow in the implementations allows an additional functionality, that of formally specifying performance. This capability is an important consideration for engineering applications, and generally, performance parameters such as computational complexity and memory usage of implemented algorithms are given informally. Since CCS is used to specify control flow with actions corresponding to procedures in the implementation, specification of such parameters for processes may be determined from the parameters for individual actions. For example, the time taken for a process composed of actions with the *prefixing* operator * is simply the sum of time taken for each action; for the example in Section 3

$$sumSquare_t = sum_t + square_t$$

the subscript t indicates the time taken for the respective action, and *nil* takes no time. Complexity analysis of this form is one of the areas for future work.

4. Distributed PCG

A large number of numerical problems in engineering require solution of a linear system of equations:

$$[K]\{X\} = \{F\}$$

where $[K]$ is a known matrix, $\{F\}$ is a known vector, and $\{X\}$ is the desired solution vector. For problems in which $[K]$ is positive-definite, several direct and iterative techniques have been suggested [22]. The preconditioned conjugate gradient method (PCG) [23, 24] is one such iterative technique for which various preconditioners have been proposed to improve the rate of convergence. Our sequential and distributed versions of the PCG consist of the modules illustrated in Fig. 7:

vecmat. Defines abstractions for vectors (*vec*) and matrices (*mat*).

pcg. Defines the sequential version of PCG using the abstractions defined in *vecmat*—this dependency is indicated in the figure by the solid arrow linking the two modules.

tcp. Defines the prototypes to be used for interprocess communication using the TCP/IP protocol.

vectort. Defines special vector abstractions (*vect*) that use the communication prototypes from *tcp*. The dotted arrow indicates that similar internal representation of vectors

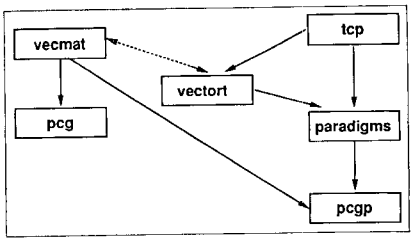

Fig. 7. Modules.

(*vec* and *vect*) are used in the two modules. Note that with this strategy, it is also possible to have different implementations of vectors that have separate interface specifications for the abstractions; users would select the implementation most suitable to their needs.

paradigms. Uses *vectort* and *tcp* to define high-level abstractions to development of our distributed algorithms.

pcgp. Defines the distributed version of PCG. The arrows in the figure illustrate that only the paradigms module is additionally required for developing the distributed implementation.

Below, we first discuss our implementation for sequential PCG, followed by a description of our client-server paradigm for modeling communication [25]. Development of distributed PCG is then discussed, and its equivalent shown with the sequential version.

4.1. Sequential PCG

The algorithm for PCG shown below uses the variable $[\tilde{K}]$ for the preconditioner approximating $[K]$. A guess for the displacement vector $\{X_0\}$ generates initial values for the residual load $\{R\}$ and the residual displacement $\{D\}$. Subsequently, at each iteration, a new conjugate direction $\{P\}$ is computed and the convergence checked using the criterion suggested by Ortega [26] which is based on the norm of the current residual vector that appears naturally in the PCG method. A line search is then used to update the solution $\{X\}$ and residual $\{R\}$. Programs implementing the algorithm often follow the same sequence, with procedure prototypes corresponding to steps in the algorithm as illustrated below:

```
Initialization:
  {X} = {X0}                          void initializeRX(vec r, vec x, vec f);
  {R} = {F} - [K]{X}
Iterative loop:
  {D} = [K~]^-1 {R}
  if first iteration
    γ = {R}^T {D}
  {P} = {D}
else
  γ' = {R}^T {D}
  if γ' < tolerance Stop
  β = γ'/γ
  γ = γ'
  {P} = {D} + β{P}
end if
{U} = [K]{P}
α = γ/(P)^T {U}
{X} = {X} + α{P}                      void modifyXR(vec x, vec r, .... );
{R} = {R} - α{U}
```

```
FVector : trait
  includes
    Vector(FVec for Vec, F for Value),
    FloatingPoint
  introduces
    __^__ : F, F → F
    __^__ : F, Int → F
    sum_exp : FVec, Int → F
    pnorm : FVec, Int → F
  asserts
    ∀v : FVec, i : Int
      size(v) = 1 ⇒ sum_exp(v, i) = v[1]^i
      size(v) > 1 ⇒ sum_exp(v, i) = (v[1]^i) + sum_exp(vec_tail(v), i)
      pnorm(v, i) == sum_exp(v, i)^float(1/pos(i))
  implies
    ∀v, v1, v2 : FVec, i : Int, f : F
      pnorm(v, i) ≥ 0
      pnorm(v1 + v2, i) ≤ (pnorm(v1, i) + (pnorm(v2, i)))
      pnorm(vec_scale(v, f), i) == f * pnorm(v, i)
```

Fig. 8. Vector *p-norm*.

Descriptions of numerical algorithms in mathematical texts [27] are generally given in a similar manner as a sequence of operations on certain predefined variables, and implementors often write code using the same variable names and following that sequence. While the mathematical formulation is precise, we feel that a more abstract description of the implementation is more intuitive, easier to implement and reuse, and more amenable to formal representation and concurrent implementations. For example, an abstract description of the above algorithm facilitates more intuitive procedure prototypes:

```
Initialization:                       void initialize(mat k, mat kbin, vec r, vec f);
  Precondition
  Initialize residual load
Iterative loop:
  Compute residual displacement
  Check convergence                   bool checkConvergence(double *gammad, .... );
  if convergent then Stop
  else
    Find conjugate direction
    Find increment factor
    Compute displacement              void compDisp(vec x, vec p, double *alpha);
    Compute residual load
  end if
```

This shift from a data-oriented to a procedure-oriented description of the algorithm is exploited in developing the specifications and implementations of sequential PCG in our layered framework.

Layer I. Defines properties of abstract data types such as vectors and matrices, hypergraphs, elements, and nodes [28]. These descriptions provide a 'vocabulary' of mathematical concepts that may be used to specify the procedures defined in Layer II. For example, Fig. 8 presents the properties of vectors whose elements are floating point numbers. The polynomial-norm, or *p-norm*, of a vector

$$pnorm(v, i) = \left[\sum_{j=1}^{j \le size(v)} v[j] \right]^{1/i}$$

```
PCG : trait
    includes
        ProcessAlgebraObCong
    introduces
        pcg, iterate :→ P
        precondition, initResidualLoad, compDisp, compResidualDisp :→ A
        compResidualLoad, findConjugateDirection, findIncrementFactor :→ A
        checkConvergence :→ Bool
    asserts
        ∀p : P
            pcg == precondition * (initResidualLoad * iterate)
            iterate == compResidualDisp*
                ( if checkConvergence then nil
                else findConjugateDirection*
                (findIncrementFactor*
                (compDisp * (compResidualLoad * iterate))))
```

Fig. 9. Control flow in sequential PCG.

is specified algebraically. The implies clause further presents properties satisfied by all vector-norms (including *p-norms*).

Layer II. Specifies the interfaces in LCL to the procedures such as pcg, which returns true if the given matrix is positive-definite, and false otherwise:

```
bool pcg(mat k, vec f, vec x, double
*tolerance) {
  requires rows(k^) = columns(k^) =
  size(f^)/\
size(f^) = size(x^) /\ (*tolerance)^ > 0 ;
modifies x;
ensures
  if positive_definite(k^)
pnorm(mat_vec_mul(k^,x') - f^,2) <=
  (*tolerance)^ /\
result = true
  else result = false;
```

The pcg procedure `requires` that its arguments are of the proper sizes and signs, `modifies` variable `x`, and `ensures` that the solution is within the specified tolerance, when it returns true. The superscripts ' and ^ represent the values of the variables on entry and exit, respectively.

Layer III. Formally specifies the control flow in the implementation of the PCG algorithm (see Fig. 9), described informally in a step-wise form above. As mentioned, the names of actions and processes in this control flow description correspond to the names of procedures in the implementation. The implementation of the pcg interface in Layer II, thus, consists of initial calls to the *precondition* and *initResidualLoad* routines, following which *iterate* is recursively called until *checkConvergence* returns true.

Layer IV. Specifies in LCL procedures which were modeled as actions or processes in the control flow description in Layer III, but were not specified in Layer II. For example, *checkConvergence* tests if convergence has been reached:

```
SendReceive : trait
    includes
        Set(Ch, ChSet),
        ProcessAlgebraObCong,
        FMatrix
    Ch tuple of p1 : P, p2 : P
    Data union of f : F, v : FVec, m : FMat
    introduces
        connect : P, P → Ch
        send : Ch, Data → A
        send : ChSet, Data → A
        receive : Ch, Data → A
        receive : ChSet, Data → A
    asserts
        ∀p, q : P, d : Data, c : Ch, chs : ChSet
            connect(p, q).p1 == p
            connect(p, q).p2 == q
            is_complement(send(c, d), receive(c, d))
            send(insert(c, chs), d) * q == send(c, d) * (send(chs, d) * q)
            receive(insert(c, chs), d) * q == receive(c, d) * (receive(chs, d) * q)
    implies
        ∀c : Ch, d : Data
            ((send(c, d) * nil) || (receive(c, d) * nil))/
                insert(send(c, d), insert(receive(c, d), {}))
                == tau * nil
```

Fig. 10. Sending and receiving.

```
ClientServer : trait
    includes
        ProcessAlgebraObCong,
        Set(P, PSet)
    introduces
        clientServer : P, PSet → P
        client : P, PSet → P
        servers : PSet, P → P
        server : P, P → P
    asserts
        ∀c, s : P, sl : PSet
            clientServer(c, sl) == client(c, sl) || servers(sl, c)
            servers({}, c) == nil
            servers(insert(s, sl), c) == server(s, c) || servers(sl, c)
```

Fig. 11. Clients and servers.

```
bool checkConvergence(double *gammad,
double *tolerance, vec r, ved d) {

  modifies *gammad;
  ensures (*gammad)' = dot_product
  (r^, d^) /\
  result == ((*gammad)' <= (*tolerance)^);

}
```

The four layers, thus, present a precise description of PCG in varying levels of detail. The abstract concepts and interface specifications in Layers I and II describe *what* is being done, are supplemented with information on how it is being done in Layers III and IV, and are gradually refined to an implementation.

4.2. Specifying client-server

Communication paradigms such as client-server are often used to simplify the development of distributed systems. Formal specification of the paradigms provides an unambiguous description of the properties of these systems, and helps eliminate problems such as deadlock and starvation. The specifications and implementations of our client-server model are developed in the modules **tcp**, **vectort** and **paradigms**, which are described below. As in sequential PCG above, the description is given with respect to our layered specification framework.

4.2.1. Module tcp

The **tcp** module is developed to provide an interface for sending and receiving messages between clients and servers using the TCP/IP protocol [29].

Layer I. Describes properties of clients and servers which communicate in our model via sending and receiving messages over channels (see Fig. 10). The **union** clause asserts that the information exchanged consists of a floating point number, a vector of floating point numbers, or a matrix of floating point numbers. The *is_complement* condition asserts that a message sent on a channel c synchronizes with a *receive* on channel c. Sending and receiving messages to/from multiple processes is defined recursively. The **implies** clause further highlights the fact that the *restriction* operator should be used to force synchronization between *sends* and *receives*. Our client-server system is modeled with processes running in parallel (see Fig. 11). The fact that a server communicates only with the client (and not with any other server) is specified by having each server see information only of the client (and not of any other server). Servers can, thus, interact with each other only through the client.

Layer II. Develops basic prototypes for initializing clients and servers, and for sending and receiving messages between them. The interface specification for initiating a server process at a given port is given as:

```
imports tcpservA;
void start_server(channel ch, void
(*task)(channel)) int errno;
{

  modifies ch;
  ensures true;

}
```

The *imports* clause here allows specification of auxiliary procedures that are not part of the interface in a separate file *tcpservA.lcl*, which is part of Layer IV. The `start_server` procedure initializes the connection with the client, and executes the given `task` – current lack of support for specifying procedure parameters in LCLint limits specification of what this task does.

```
TcpServF : trait
    includes
        ProcessAlgebraObCong
    introduces
        startServer, forkClientRequests, parent :→ P
        childProcess, task :→ P
        createSocket, accept, fork, closeParentSocket :→ A
        closeChildSocket, signal :→ A
    asserts
        ∀p : P
            startServer == createSocket * forkClientRequests
            forkClientRequests == accept * (fork * (childProcess || parent))
            childProcess == closeParentSocket * task
            parent == closeChildSocket * (signal * forkClientRequests)
```

Fig. 12. Control flow for TCP server.

Layer III. The *startServer* process, which refines the `start_server` procedure in Layer II, initially creates a socket, and then forks requests from the client process to execute specific tasks (see Fig. 12). The control flow for the client is similarly described, and for brevity, is not given here.

Layer IV. As before, we specify in this layer procedure prototypes for various actions and processes in the control flow description in Layer II. Further, auxiliary specifications are also given for standard header files to highlight relevant data structures, constants, etc., used in the implementation. For example, *socket.lcl* specifies only the following:

```
imports < types > ;
constant int SOCK_STREAM;
constant int AF_INET;
typedef struct sockaddr {

  u_short sa_family; /* address family */
  char sa_data[14]; /* up to 14 bytes of
  direct address */

} sockaddr;
```

4.2.2. Module vectort

This module introduces an abstract data type *vect* that has the same internal representation as *vec* in the *vecmat* module, but has new operators for sending and receiving vectors over sockets. While the communication operators could have been incorporated in the *vec* abstract data type, they would have overspecified the notion of a vector. Similar conflicts in deciding whether to have a separate module for each application, or one module attempting to satisfy all applications arise often. Since reuse usually requires some modification of existing abstractions, we prefer the former strategy. Since we use the programming language C, having two modules for vectors also results in two copies of the same internal representation of vectors; *vec* and *vect* differ only in the methods provided with the respective abstract data types. Though this duplication can be avoided with programming languages such as C++ that support code inheritance as a way to implement subtyping [30], one still has to decide whether to have *vect* as a subtype of a more generic class *vec* or to have just one class for vectors. For brevity, the various layers for the **vectort** module, and for the **paradigms** module below, are not given.

4.2.3. Module paradigms

Developing paradigms for concurrent computation reduces the complexity of the implementation, and reduces potential for errors. For our distributed PCG implementation, we add abstract data types for *client_vec* and *server_vec* to encapsulate the client and server vector operations. The interface specification for *server_vec* uses traits such as *ClientServer*, modeling connection-oriented client-server interactions, and *FVector*, modeling vector operations on sequential processors, to describe operations associated with the abstraction.

4.3. Distributed PCG

The specification of distributed PCG fits in the four-layered specification framework in a similar fashion. In this section, however, we focus the discussion on Layer III—the specification of control flow. We use the concurrency workbench to show that the control flow of our sequential and distributed versions are equivalent. We first present some lemmas that hold true of CCS processes, and then use them in the following proof of equivalence.

4.3.1. Useful lemmas

The lemmas are based on the existence of two systems $Dis = (Cl|Ser)\backslash Comm$ and Seq such that:

- $Dis \approx {}^{c}Sq$
- Cl, Ser, and Seq are composed only with the operator thus modeling only sequential processes.
- $Comm$ is the set of labels for all actions in Ser. Thus, the process Dis has no visible actions from Ser.
- No actions of $Comm$ (and Ser) are in Seq.

An example is $Cl = s\bar{a}.a.s\bar{b}.b.nil$, $Ser = sa.sb.nil$, $Comm = \{sa, sb\}$, and $Seq = a.b.nil$. Intuitively, Seq is a sequential process and Dis is its distributed implementation using a single client Cl and a single server Ser. The composition of actions a and b models the sequential computations in Seq. In the distributed system here, each server does a fraction of the total work, e.g. the work done by action sa on server Ser is a fraction of the total work done by action a on Seq. The client server system Dis is said to have completed the computation a once each of the servers have finished their respective fractions sa. Note that the there is only one server in this example, and thus sa on the server Ser does the same amount of work as a on the sequential process Seq.

Lemma 1. *For each action a in Ser, there is a synchronizing action $\bar{a}$ in Cl.*

This follows since all actions in server Ser are hidden, and for the system Dis to be equivalent to Seq, corresponding synchronizing actions are required from Cl.

Lemma 2. *Let $Ser(n) = S_1|S_2|...|S_n$ where $S_1 = S_2 = ... = S_n = Ser$ and $Cl(n)$ is the process obtained on replacing each action a in Cl that synchronizes with an $\bar{a}$ in Ser with $a_1.a_2...a_n$ where $a_1 = a_2 = ... = a_n = a$. Then $(Cl(n)|Ser(n))\backslash Comm \approx {}^{c}Seq$.*

A consequence of this lemma is that proving a distributed system Dis, composed of one client Cl and multiple (and identical) servers, equivalent to a sequential system Seq with the above restrictions only requires proving that the system $Dis = (Cl|Ser)\backslash Comm$ is equivalent to Seq.

Lemma 3. *Let $Ser(n) = S_1|S_2|...|s_n$ where $S_i = Ser[a/a|\bar{a} \in Sort(Ser)]$, $Cl(n)$ is the process obtained on replacing each*

```
bi Q c.d.t.(nil + e.f.g.h.Q)
bi Seq t.a.b.Q

bi SQ sc.sV.rV.sd.sD.rD.t.(nil + se.sf.sD.rD.sg.sh.SQ)
bi PcgS sa.sV.rV.sb.SQ

bi CQ 'sc.rV.t.sV.c.'sd.rD.t.sD.d.t.(nil + 'se.e.'sf.rD.t.sD.f.'sg.g.'sh.h.CQ)
bi PcgC 'sa.rV.t.sV.a.'sb.b.CQ

bi Ser PcgS[rV/'sV,sV/'rV,rD/'sD,sD/'rD]
bi Cl PcgC

bi Dis (Cl | Ser)\{rV,sV,rD,sD,sa,sb,sc,sd,se,sf,sg,sh}

* cong Seq Dis
```

Fig. 13. Single-client single-server model for distributed PCG.

action a in Cl with $a_1.a_2...a_n$ and $Comm(n)$ is the set of actions obtained on replacing each $a \in Comm$ with $a_1.a_2...a_n$. Then $(Cl(n)|Ser(n))\backslash Comm(n) \approx {}^{c}Seq$.

As in Lemma 2, this lemma also allows showing equivalence between sequential and distributed systems, except that now the servers may not be identical.

Lemma 4. *Let $Ser(n) = S_1|S_2|...|S_n$ where $S_i = Ser[a/\bar{a} \in Sort(Ser)]$, $Cl(n)$ is the process obtained by replacing some sequences of action of Cl, e.g. a.b, with n corresponding sequences $a_1.b_1.a_2.b_2...a_n.b_n$ and $Comm(n)$ is the set of actions obtained on replacing a,b in Comm with $a_1.b_1.a_2.b_2...a_n.b_n$. Then $(Cl(n)|Ser(n))\backslash Comm(n) \approx {}^{c}Seq$.*

The results of this lemma are more restrictive than Lemmas 2 and 3 since more constraints are placed on how the client process $Cl(n)$ orders repetition of actions. However, the lemma allows us to conclude that `Seq` $\approx$ `Dis` for any number of servers added to the distributed system.

4.3.2. Sequential PCG $\approx$ ᶜdistributed PCG!

We build the specifications and implementations for distributed PCG using the CCS lemmas presented earlier and the specifications and implementations of sequential PCG and client-server. Fig. 13 presents an abstract CCS specification of the single-client single-server case in our model. The process `Seq` represents the sequential implementation of PCG described in Larch in Fig. 9 with the following actions relabeled for clarity:

`Seq` = *pcg*
`Q` = *iterate*
`a` = *precondition*
`b` = *initResidualLoad*
`c` = *compResidualDisp*
`d` = *checkconvergence*
`e` = *findConjugateDirection*
`f` = *findIncrementFactor*
`g` = *compdisp*
`h` = *compResidualLoad*

The process `Dis` models the distributed case, where `Cl` and `Ser` are the client and server processes, respectively. To define `Ser`, we use a relabeling on the process `PcgS`. Note that for each action in the sequential PCG process `Seq`, there is a corresponding server action, e.g. `sa` in `PcgS` for `a` in `Seq`, to model the fraction computed on each server. We also introduce additional actions `sV` and `rV`

```
bi Q c.d.(nil + e.f.g.h.Q)
bi Seq t.a.b.Q

bi SQ sc.sV.rV.sd.sD.rD.t.(nil + se.sf.sD.rD.sg.sh.SQ)
bi PcgS sa.sV.rV.sb.SQ

bi SQ2 sc2.sV2.rV2.sd2.sD2.rD2.t.(nil + se2.sf2.sD2.rD2.sg2.sh2.SQ2)
bi PcgS2 sa2.sV2.rV2.sb2.SQ2

bi CQ 'sc.'sc2.rV.t.rV2.t.sV.sV2.c.'sd.'sd2.rD.t.rD2.t.sD.sD2.d.t.\
   (nil + 'se.'se2.e.'sf.'sf2.rD.t.rD2.t.sD.sD2.f.'sg.'sg2.g.'sh.'sh2.h.CQ)
bi PcgC 'sa.'sa2.rV.t.rV2.t.sV.sV2.a.'sb.'sb2.b.CQ

bi Ser1 PcgS[rV/'sV,sV/'rV,rD/'sD,sD/'rD]
bi Ser2 PcgS2[rV2/'sV2,sV2/'rV2,rD2/'sD2,sD2/'rD2]
bi Cln PcgC

bi Disn (Cln | Ser1 | Ser2)\{rV,sV,rD,sD,sa,sb,sc,sd,se,sf,sg,sh,rV2,\
                            sV2,rD2,sD2,sa2,sb2,sc2,sd2,se2,sf2,sg2,sh2}

* cong Seq Disn
```

Fig. 14. Single-client two-server model for distributed PCG.

PcgClient : **trait**
includes
ProcessAlgebraObCong
introduces
$mergeV, mergeD :\rightarrow A$
$sV, rV, sD, rD :\rightarrow A$
$pcgC, iterateC :\rightarrow P$
asserts
$\forall p : P$
$pcgC == rV * (mergeV * (sV * iterateC))$
$iterateC == rV * (mergeV * (sV * (rD * (mergeD * (sD *$
$(nil + (rD * (mergeD * (sD * iterateC)))))))))$

Fig. 15. Client for distributed PCG.

that model sending and receiving of vectors in our distributed PCG implementations. Likewise, actions for similar operations on doubles, namely `sD` and `rD` are also used. The τ actions after each vector or double is received on the client, i.e. `rV` or `rD`, models the operation to merge (local) values from the servers into a global sum. The concurrency workbench [10] is used to check for equivalence of the sequential and distributed implementations, i.e. `Seq` $\approx$ `'Dis`:

```
The Edinburgh Concurrency Workbench
(Version 6.01, October 23, 1993) TCCS
includes maximal progress assumption
Command: if pcgp1
done
Command: pe
  CQ = 'sc.rV.t.sV.c'sd.rD.t.sD.d.t.
  (nil + 'se.e.'sf.rD.t. sD.f.'sg.g.'sh.
  h.CQ)
  Cl = PcgC
  Dis = (Cl | Ser)\{rV,sV,rD,sD,sa,sb,sc,
  se,sf,sg,sh}
  PcgC = 'sa.rV.t.sV.a.''sb.b.CQ
  PcgS = sa.sV.rV.sb.SQ
  Q = c.d.t.(nil + e.f.g.h.Q)
  SQ = sc.sV.rV.sd.sD.rD.t.(nil + se.sf.
  sD.rD.sg.sh.SQ)
  Seq = t.a.b.Q
  Ser = PcgS['rV/sV,'sV/rV,'rD/sD,'sD/
  rD]
Command: cong
Agent: Seq
Agent: Dis
true
```

After the input file given in Fig. 13 is loaded, the `pe` command prints the environment. The command `cong` is then used to test whether `Seq` and `Dis` are observationally congruent.

The lemmas described earlier now provide the equivalence proof for the case when the distributed version consists of a single client and multiple servers. The processes *Seq*, *Dis*, *Cl* and *Ser* in Fig. 13 satisfy the conditions under which the above lemmas are true. From Lemma 4, the single-client multiple-server case is equivalent to the sequential process if replaced sequences conform to certain requirements. The replaced sequences for our implementation can be seen by comparing the control flow descriptions of the single-client single-server and the single-client two-server cases in Figs 13 and 14, respectively. To link these CCS descriptions to our specification framework, corresponding Larch specifications of control flow are built for the client and server in Figs. 15 and 16, respectively. As before, the action names in these control-flow specifications correspond to names of procedures in the actual code, and interface specifications are also written to describe them. Using this refinement approach, the distinction between actions (in the control flow specifications) that represent computations and those that represent communication is maintained in the implementations. This separation of

PcgServer : **trait**
includes
ProcessAlgebraObCong
introduces
$pcgS, iterateS :\rightarrow P$
$preconditionS, initResidualLoadS, compDispS, compResidualDispS :\rightarrow A$
$compResidualLoadS, findConjugateDirectionS, findIncrementFactorS :\rightarrow A$
$checkConvergenceS :\rightarrow A$
$sV, rV, sD, rD :\rightarrow A$
asserts
$\forall p : P$
$pcgS == preconditionS * (sV * (rV * (initResidualLoadS * iterateS)))$
$iterateS == compResidualDispS * (sV * (rV *$
$(checkConvergenceS * (sD * (rD *$
$(nil + (findConjugateDirectionS *$
$(findIncrementFactorS * (sD * (rD *$
$(compDispS * (compResidualLoadS * iterateS)))))))))))$

Fig. 16. Server for distibuted PCG.

concerns at the procedural level is helpful in locating and correcting bugs in distributed programs.

Note that the CCS traits in LSL are useful to support both the low- and high-level needs: they facilitate the specification of abstractions and paradigms in distributed computation in Layer I, e.g. the interprocessor communications in the connection-oriented client-server model of Section 4; also, they serve another important purpose of providing a way to describe the control flow in our implementations in Layer III.

5. Tool support

Several syntax and semantic checks are performed in the refinement of specifications to 'real' code. The LSL checker is used to perform syntax checks on the LSL specifications, and to convert these into an input suitable for LP, which is used for semantic checks. The transition from LSL theory to interface specification is performed using LCLint, which is also used to check for conformance of these specifications with code in the C programming language. The concurrency workbench is used for equivalence testing of CCS specifications. Further, as in the usual development process where no formal specifications are used, regular checks on the implementations with test-inputs are done to catch errors in C programs.

A limitation of working with Larch specifications is that the tools are presently not mature enough for an average program developer [31]. Using LP has a high start-up cost and takes patience and expertise. LCLint is very much a prototype; it is limited in functionality and is not itself fully debugged. It still lacks support for some useful and common data types, e.g. bit-structs. A minor syntactic point is the lack of operator precedence in LSL, a feature available in LP; providing it in LSL would reduce the number of parentheses in control flow specifications, enhancing their readability. Nevertheless, the growing use of Larch in more studies, publication of experiences of users, and refinement of available tools together with development of new ones, are positive indications of more widespread use of such specification frameworks.

Using Larch to specify the control flow in implementations also creates a demand for more tools supporting our extended methodology. Development of parsers to and from the control flow descriptions in LSL from/to the concurrency workbench would help ensure consistency of the specifications. It would also be useful to have tools that ensure that the ordering of actions in the specified control flow are met in the actual code. Note that this consistency can also be achieved by directly generating procedure prototypes from control flow specifications.

6. Conclusions

This study proposes a formal method for developing concurrent systems. We demonstrated our method on the distributed PCG method. Using a formal approach to system development and maintenance increases the reliability of the end-product and promotes its reuse. An important advantage of the methodology is its ability to refine specifications to code in a programming language. Most frameworks for formal development, e.g. Unity [32], decouple a program from its implementation, and usually involve checks in the specification only. While tools based on languages defining a process algebra, such as the LOTOS Toolbox, overcome some of these drawbacks since they model the system more closely to its actual implementation, conformance of the code with the specifications are not dealt with much rigor. The LOTOS Toolbox does provide the TOPO back-end C generator [33] to develop prototypes for early evaluation of a design system, but does no checks for the actual implementation.

Our extension to the two-tiered Larch model for specification supports both an abstract description of the system which can be employed by users, and a detailed control flow specification which is refined to the implementation. The latter is useful since reuse often requires small changes in the implementation, and a formal specification helps ensuring the required changes are made. Further work, however, is required in developing better tools to support the development and maintenance of specifications and implementations in our layered approach. We also plan to pursue the idea of using the control flow description for specification of the computational complexity of processes. Further, we are extending this case study for distributed PCG to build a library of specifications and implementations for finite-element analysis. Work is also in progress to make our refinement strategy more formal. Specifically, we hope to exploit studies in action refinement of process algebras [34] to provide a more formal link between the abstract properties in Layers I and II and the control flow description in Layer III.

Acknowledgements

This work is supported in part by the National Science Foundation under grant number MSS-9201697 entitled 'Reusable Engineering Software Components,' and by the Department of Civil Engineering at North Carolina State University. The third author is supported in part by the Wright Laboratory, Aeronautical Systems Center, Air Force Material Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government. The authors thank Rance Cleaveland for his useful comments on earlier drafts of this paper.

References

[1] Hoare CAR. Communicating sequential processes. Englewood Cliffs, NJ: Prentice-Hall, 1985.

[2] Milner R. Communication and concurrency. Englewood Cliffs, NJ: Prentice-Hall, 1993.

[3] Spivey JM. Understanding Z, a specification language and its formal semantics. Cambridge, MA: Cambridge University Press, 1988.

[4] Jones CB. Systematic software development using VDM. Englewood Cliffs, NJ: Prentice-Hall, 1990.

[5] Guttag JV, Horning JM, Garland SJ, Jones KD, Modet A, Wing JM. Larch: languages and tools for formal specification. Berlin: Springer, 1993.

[6] Bolognesi T, Lagemaat Jvd, Vissers CA. LOTOSphere: software development with LOTOS. Dordrecht: Kluwer Academic, 1994.

[7] Group TRL. The RAISE specification language. Englewood Cliffs, NJ: Prentice-Hall, 1992.

[8] Ehrig H, Mahr B. Fundamentals of algebraic specification I: equations and initial semantics. In: EATCS monographs on theoretical computer science. Berlin: Springer, 1985.

[9] Wing JM. A specifier's introduction to formal methods. Computer 1990;23(9):8–24.

[10] Cleaveland R, Parrow J, Steffen B. The concurring workbench. ACM Transactions on Programming Languages and Systems 1993;15(1):36–72.

[11] Chadha HS. Formal specification and verification of concurrent systems. Ph.D. thesis, Department of Civil Engineering, North Carolina State University, May 1996.

[12] Lin H. PAM: a process algebra manipulator. Technical Report Computer Science 93:04, University of Sussex, 1993.

[13] Mauw S, Veltink GJ. A proof assistant for PSF. In Computer Aided Verification. LNCS 575, 1991.

[14] Buth K. Using SOS definitions in term rewriting proofs. In: Martin U, Wing JM, editors. First International Workshop on Larch. Berlin: Springer, 1992:36–54.

[15] Tan YM. Formal specification techniques for promoting software modularity, enhancing documentation, and testing specifications. Ph.D. thesis, Department of Electrical Engineering and Computer Science, MIT, May 1994.

[16] Baugh JW Jr. Is engineering software amenable to software specification? In: Martin U, Wing JM, editors. First International Workshop on Larch. Berlin: Springer, 1992:1–17.

[17] Bowen JP, Hinchey MG. Ten commandments of formal methods. Computer, April 1995.

[18] Craigen D, Gerhart S, Ralston T. An international survey of industrial applications of formal methods. NIST GCR 93-626-V2 PB93-178564/AS, March 1993.

[19] Chetali B. Formal verification of concurrent programs: How to specify UNITY using the larch prover. Technical Report n2475, INRIA-lorraine, January 1995.

[20] Mili H, Mili F, Mili A. Reusing software: issues and research directions. IEEE Transactions on Software Engineering 1995;21(6):528–561.

[21] Ahmed S, Carriero N, Gelernter D. A program building tool for parallel applications. In: DIMA CS Workshop on Specifications of Parallel Algorithms. Princeton University, May 1994.

[22] Chadha HS, Baugh Jr JW. Network distributed finite element analysis. Advances in Engineering Software 1996;25:267–280.

[23] Hestenes MR, Stiefel E. Methods of conjugate gradients for solving linear systems. Journal of Research, National Bureau of Standards, Section B, 1952:409–436.

[24] Shewchuk JR. An introduction to the conjugate gradient method without the agonizing pain. Technical report, School of Computer Science, Carnegie Mellon Uiversity, Pittsburgh, PA, August 1994.

[25] Sharma SK, Baugh Jr. JW, Chadha HS. A client-server approach for distributed finite element analysis. Advances in Engineering Software 1993;17:69–78.

[26] Ortega JM. Introduction to parallel and vector solutions of linear systems. New York: Plenum Press, 1988.

[27] Golub GH, Van Loan CF. Matrix computations, 2nd edn. Baltimore, MD: The John Hopkins University Press.

[28] Baugh Jr. JW. Using formal methods to specify the functional properties of engineering software. Computers and Structures 1992;45(3):557–570.

[29] Feit S. TCP/IP: Architecture, Protocols, and Implementation. Series on Computer Communications. New York: McGraw-Hill.

[30] Liskov BH, Wing JM. A behavioural notion of subtyping. ACM Transactions on Programming Languages and Systems, November 1994.

[31] Chalin P. Shortcomings of LCL 2.4. Technical Report CU/DCS-TR-95-04. Concordia University, Department of Computer Science, 1995.

[32] Brown M. Towards an integrated environment for concurrent programs development. In: Nivat M, Rattray C, Rus T, Scollo G, editors. Algebraic methodology and software technology. Berlin: Springer, 1993:407–408.

[33] van der Vloedt T. The LOTOS toolbox. In: Nivat M, Rattray C, Rus T, Scollo G, editors. Algebraic methodology and software technology. Berlin: Springer, 1993:409–410.

[34] Aceto L. Action refinement in process algebras. Cambridge, MA: Cambridge University Press, 1992.