業務システムのための知識処理サーバの設計と実装

梅田政信[†] 片峯恵一[†] 長澤勲[†]
橋本正明[†] 高田修[†]

企業等の基幹業務を支える情報システム（以下，業務システム）で専門知識を生かした高度な機能を実現するには，専門知識を体系的に整理，記述した知識ベースを内蔵する知識ベースシステムが適している．本論文では，Prolog 上に実装された知識ベースシステムを業務システムで運用するための知識処理サーバについて述べる．知識処理サーバは，特定の業務システムや知識ベースシステムに依存しない汎用の枠組みであり，新たに開発したマルチスレッド対応の Prolog 処理系と Java 処理系とを組み合わせて実現されている．本サーバは，業務システムとのインタフェースを RMI，CORBA 等の分散オブジェクト技術に対応させ，各種業務システムとの相互運用性を高めている．また，J2EE においては，知識ベースシステムによるトランザクションの継承を可能にし，トランザクションシステムへの組み入れを容易にした．評価実験の結果，Prolog 処理系のマルチスレッド機能は SMP システムで高い並列性が得られ，知識処理サーバが SMP システムでスケーラビリティを有することを確認した．知識処理サーバは，病院情報システムの臨床判断支援に実際に適用され，薬剤禁忌の検証等を可能にしている．この運用実績から，知識処理サーバが実用上十分な頑健性を備えていることも確認した．

The Design and Implementation of Knowledge Processing Server for Enterprise Information Systems

Masanobu Umeda,[†] Keiichi Katamine,[†] Isao Nagasawa,[†]
Masaaki Hashimoto[†] and Osamu Takata[†]

A knowledge-based system is suitable for realizing advanced functions that require domain-specific expert knowledge in enterprise-mission-critical information systems (enterprise applications). This paper describes the knowledge processing server that operates a knowledge-based system written in Prolog with an enterprise application. It is an independent framework of any enterprise application and knowledge-based system, and is realized by combining newly implemented multi-threaded Prolog and Java. The server improves interoperability with various enterprise applications due to its adaptation to distributed object technology, such as RMI and CORBA, using Java. The server also makes it easier to incorporate a knowledge-based system into a transaction system by allowing a knowledge-based system to inherit transactions of an enterprise application in the J2EE environment. Experimental results indicated that the multi-threaded Prolog could obtain high parallelism on an SMP system, and the server could also achieve scalability on it. The server has been applied to clinical decision support in a hospital information system, and it enables the validation of contraindications, such as drug-drug interactions, on prescription orders. This result demonstrated that the server was practically robust enough for use in an enterprise application.

1. はじめに

病院情報システムや物流システム等，企業等の基幹業務を支える情報システム☆（以下，業務システム）では，各分野の専門知識を生かした高度な機能への期待が高まっている．医療過誤を防止するための臨床判断支援[1)]や最適在庫のための発注支援等は，その一例である．このような機能の実現には，対象領域の専門知識を体系的に整理，記述した知識ベースを内蔵する知識ベースシステム[2)]が適している．

知識ベースシステムを業務システムに適用する試みには，プロダクションシステム[2),3)]を Java 技術と組み合わせ[4)]，業務ルールに適用する提案がある[5)~7)]．変更頻度が比較的高い業務ルールを手続型言語による業

† 九州工業大学大学院情報工学研究科
Graduate School of Computer Science and Systems Engineering, Kyushu Institute of Technology

☆ ここでは，複数のユーザでリソースを共有し，一貫性維持のためのトランザクション処理が不可欠な情報システムを指すものとする．

務フロー記述から分離することで，その開発や保守を効率化できる可能性があるが，プロダクションシステムを大規模な業務ルールに適用する上での課題[3),☆1]が十分解決されているわけではない．一方，知識処理に適したProlog処理系をJava技術と組み合わせることで情報システムの高度化を図る提案もある．しかし，これらの提案には，業務システムに不可欠なスケーラビリティやトランザクション処理に課題が残されている．たとえば，SICStus Prolog[8)]やCIAO Prolog[9)]は，プロセス間通信による業務システムとの接続を提案している．しかし，独自プロトコルのため，業務システムで多用されるJ2EE（Java 2 Enterprise Edition）のトランザクション処理等に汎用的に対応することは難しい．SWI-Prolog[10)]は，JNI（Java Native Interface）によるJavaインタフェースを持つが，これがマルチスレッドに対応していないために☆2スケーラビリティに課題が残る．tuProlog[11)]やJinni[12)]は，Javaによる Prolog実装であり，Javaによる業務システムとの親和性は高い．しかし，現状のJavaによる実装技術ではC/C++による実装と比較して推論性能は劣る．このためJinniでは，Cで実装されたBinPrologに負荷の高い処理をプロセス間通信で委譲する手法も提案しているが，CIAO Prolog等と同様にトランザクション処理に課題が残る．

一方，著者らは，知識ベースシステムのための開発環境Inside Prolog[13)]を開発し，この上に設計計算支援システム[14)~16)]や健康管理支援システム[17),18)]等の各種知識ベースシステムを構築し☆3，実用に供してきた[19)]．Inside Prologは，ISO/IEC 13211-1[20)]に準拠した標準的なProlog機能に加えて，実用アプリケーション開発に不可欠な各種API（Application Programming Interface）を備え，知識ベースシステムのプロトタイプ開発から実用システム開発までを一貫して支援できる．しかし，スタンドアローンを想定した設計のために，これを業務システムにそのまま適用することは困難であった．

そこで著者らは，まずスケーラビリティの問題に対応できるようにInside Prologを発展させ，新たにマルチスレッド対応のProlog処理系を開発した．次に，これとJavaとを組み合わせることにより，各種知識ベースシステムを業務システムで運用するための知識処理サーバを構築した．このことにより，知識ベースシステムの開発から業務システムでの運用までを一貫して支援可能になる．知識処理サーバは，病院情報システムCAFE[21),22)]での臨床判断支援に適用され[23)~25)]，薬剤禁忌の検証☆4や，薬量や施用法等の提案，検査要約等の診療情報の提供を可能にしている．

以下，本論文では，まず2章で業務システムのための要件を明らかにし，3章でInside Prologの概要を，4章で業務システムのためのProlog処理系のマルチスレッド拡張について述べる．次に，5章で知識処理サーバの設計と実装について述べ，6章でシステムを評価する．

2. 業務システムのための要件

知識ベースシステムを業務システムに適用するための要件を以下に述べる．

システムのスケーラビリティ 業務システムには，利用者数や負荷の増大で応答性に問題が生じたとき，ソフトウェアを大きく変更することなく，ハードウェアの拡張や運用パラメータの調整で改善できるスケーラビリティが求められる．ここでは，SMP（Symmetric Multiple Processor）システムにおけるスケーラビリティを対象とする．

システムの頑健性 業務システムの障害は，基幹業務に大きな影響を及ぼす可能性がある．このため，メモリリーク等でクラッシュすることなく，安定したサービスを継続的に提供できるように，高い頑健性が求められる．

システムの相互運用性 業務システムの実現には，一般に様々なプログラミング言語や実装技術が使われる．このような業務システムの多様性に柔軟に対応するには，SOAP，CORBA等のプログラミング言語独立な分散オブジェク技術や，業務システムに多用されるJavaおよびRMIに対応できる必要がある．

トランザクション処理への対応 データベース等の一貫性を維持するトランザクション処理は，高い信頼性が要求される業務システムに不可欠であり，その管理をアプリケーションフレームワークに任せることが一般的になっている．ここでは，業務システムに多用されるJ2EEのトランザクション処理に対応可能とする．

運用管理の支援 知識ベースシステムの負荷や稼働の状況は，計算機リソースの最適配分等に不可欠である．また，知識ベースの障害時に，これを業務システ

☆1 副作用の生起，組合せ的爆発，および制御の飽和問題である．
☆2 SWI-Prologの処理系自身はマルチスレッドに対応している．
☆3 当初は市販のLispやPrologで開発され，可搬性や拡張性等からInside Prolog上に移植されたものもある．
☆4 薬剤の相互作用による禁忌（併用禁忌）や薬剤と病名との禁忌（病名禁忌）等を検証し，注意を促す機能である．

ムから切り離し，最低限のサービスを業務システムが継続できる必要もある．したがって，このような運用管理を支援する機能が求められる．

3. Inside Prolog の概要

Inside Prolog は，知識ベースシステムのプロトタイプから実用システムまでの開発を，単一のプログラミング言語で一貫して支援することを目的とした統合開発環境である[13),19)]．このために，Prolog を核言語として，この上に実用的なシステムの開発に必要となる様々な機能を提供している☆．これらの機能は，Unix や Windows 等の計算機環境と独立した API として提供し，必要に応じて拡張データ型[13)] と C/C++ 言語インタフェースを用いて拡張できる．**図 1** は Inside Prolog のシステム構成を表す．Inside Prolog は，WAM[28)] に基づく Prolog 抽象機械 TOAM を中核として，この上に ISO/IEC 13211-1[20)] に準拠した Prolog 処理系を有する．同処理系は，照合木を用いたユニフィケーションの最適化[29)] や決定的な述語の C 関数への変換[30)] 等の最適化機能を提供する．この上に，実用上不可欠な API として，ファイルシステムやプロセス，ネットワーク，Motif や Windows に対応した GUI，3 次元モデラ，数値計算ライブラリ，Java 処理系等とのインタフェースを，また外部システムとの連係のための API として，JDBC，ODBC，OLE 等のインタフェースを，それぞれ提供する☆☆．この特徴により，C/C++等で記述された一般のアプリケーションと同等の GUI や操作性を備えた知識ベースシステムを，単一のプログラミング言語 Prolog を用いて構築できる．

以下では，Inside Prolog のマルチスレッド化に関わりのある内部構成について，その概略を示す．

3.1 TOAM のメモリモデル

TOAM は，**図 2** に示すように，WAM 同様なコントロールスタック，トレイルスタック，グローバルスタック（ヒープとも呼ばれる）の 3 つのスタックと，述語定義等を保持するデータ領域（アトム領域とも呼ばれる）とを持つ．データ領域は，データ変更操作ができない永続データ領域（persistent area）と変更可能な一時データ領域（transient area）とに分けられ，

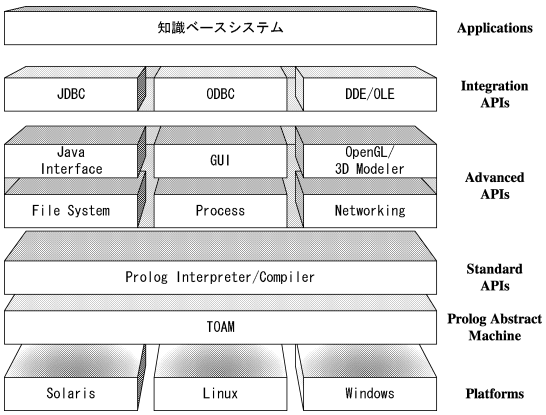


図 1 Inside Prolog の構成
Fig. 1 Architecture of Inside Prolog.

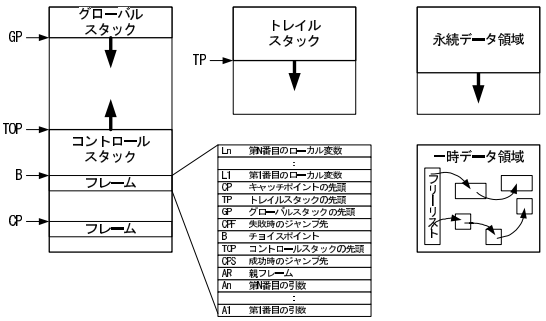


図 2 TOAM のメモリモデル
Fig. 2 Memory model of TOAM.

永続データ領域には知識ベースシステムの推論機構や GUI 等の動的に変更されないものを，一時データ領域には知識ベースや実行結果等の動的に変化するものを，それぞれ格納する．データ領域を二分することにより，(1) ゴミ集め処理（garbage collection）☆☆☆から永続データ領域を除外できるため，処理時間を短縮できる，(2) 推論機構等を誤って破壊することを防止でき，ソフトウェアの信頼性を向上できる，(3) プログラムのメモリ配置が不変なことを利用した最適化（3.3 節で詳述）が可能となる，等の利点がある．

3.2 プログラムコードの表現

知識ベースシステムの構成要素は，推論機構や GUI のように動的に変更されない静的なプログラム，製品カタログ等の技術情報[31)] や推論ルールのように編集操作によって動的に変更可能なデータ，推論ルールや技術情報から動的に生成されるプログラム等，いくつかの性質が異なるものに分類できる．Prolog はプログ

☆ Prolog を核言語とするこのような統合開発環境は，古典的な Lisp マシン[26)] や Prolog マシン[27)] と似たアプローチであり，他に見当たらない．

☆☆ たとえば，OLE のように Windows に固有な機能は，最小限の機能をライブラリとして供給することにより，アプリケーションの記述を一元化できる工夫をしている．

☆☆☆ アルゴリズムには実アドレスによるバイトコード管理が容易な mark-and-sweep を採用している．

ラムとデータとを同列に扱えるため，これらを 1 つのプログラムコード表現にできるが，それぞれに得失がある．たとえば，推論ルールから生成するプログラムに対して推論機構と同様の最適化を施すと，実行は高速化されるが翻訳処理に時間を要するため，推論ルールの対話的なデバッグには適さない．そこで Inside Prolog では，述語のプログラムコード表現を，その役割や場面に応じて，次の 4 つの中から選択可能にすることにより，実用的な知識ベースシステム構築を可能にしている．

Static プログラム 最適化コンパイラ[29] により生成される TOAM のバイトコードである．これは，推論機構のように，実行速度が重要で，かつ動的に変更されない static 述語☆1に適している．Static プログラムは，指示により永続データ領域か一時データ領域のいずれかに格納される．

Native プログラム 決定的な述語を C 関数に変換[30] して組み込み述語としたものである．これは，変更の可能性が低く，実行速度が特に優先される決定的な static 述語に適している．述語の変換は自動的に行えるが，オブジェクトコードが計算機環境に依存するため，使用は限定的になる．

Incremental プログラム `assertz/1` 等により節単位に生成される TOAM のバイトコード☆2であり，推論ルールから生成する実行コードのように，動的に定義，実行される dynamic 述語☆3に用いる☆4．動的に述語を定義，実行する場合，各処理に要する時間のバランスが重要である．定義時は最適化を省いて翻訳時間を短縮し，また実行では ISO/IEC 13211-1 に規定された logical database update[20],☆5を省略して高速化を図っている．

Interpretive プログラム Prolog インタプリタにより解釈実行される項表現であり，incremental プログラム同様に動的に定義，実行されるが，バイトコードへの翻訳効果が低い dynamic 述語に用いる．特に，基底項からなる単位節の場合，グローバルスタックに項を複写せずに共有するため，構造共有方式[32] と同様の効果が得られる．たとえば，製品カタログ[31] や薬剤の禁忌情報[23),24)] は，基底項からなる単位節の集合で表現できるため，interpretive プログラムが適している．

☆1 ISO/IEC 13211-1 に規定された節集合を変更できない述語．
☆2 Static プログラムと異なりコードの最適化は行われない．
☆3 ISO/IEC 13211-1 に規定された節集合を変更可能な述語．
☆4 `clause/2` 用に翻訳前の項表現もあわせ持つ．
☆5 述語を実行中に，当該述語への `assertz/1`，`retract/1` 等による編集操作が行われたときの影響範囲を規定している．

3.3 命令書き換えによる最適化

一般に述語を呼び出す際には，シンボル表から述語名に対応するプログラムコードを取得する必要がある．1 回のシンボル表参照に要する時間は微少であるが，繰り返し実行すると，その累積時間は無視できない．そこで TOAM では，述語呼び出し命令 `call`，`execute` 等が実行されると，プログラムコードの種類に応じて，これを書き換えて最適化する．たとえば，呼び出し命令 `call` は，永続データ領域に格納された static プログラムの呼び出しなら，そのメモリ配置が不変なことを利用して，実アドレスによる呼び出し `call_direct` に書き換える．また，native プログラムなら `call_native` に，それ以外ならシンボル表を参照する `call_indirect` に書き換える．この結果，シンボル表の参照は `call_indirect` 等のいくつかの命令に限定される．

4. 業務システムのためのマルチスレッド拡張

本章では，Inside Prolog の適用範囲を業務システムに広げるためのマルチスレッド機能について述べる．

4.1 Prolog 処理系のマルチスレッド化

Prolog 処理系のマルチスレッド化には，(1) スレッドのスケジューリングやコンテキスト切替えを処理系の機構として実現する手法[8),33)] と (2) 汎用のマルチスレッドライブラリを用いて実現する手法[9),10)] とが知られている．また，Java 上の Prolog 実装では，(3) シングルスレッドの Prolog 処理系を複数生成し，これらを複数の Java スレッド上で動作させる手法もある[11),34)]．(1) は，コンテキスト切替えや排他制御を単純化できるため高速に動作可能な利点があるが，SMP システムを活用することが難しい[35]．一方 (2) は，同期処理やコンテキスト切替えのコストが前者より大きいが，SMP システムで並列処理によるスループット向上が可能である．(3) は，処理の並列性は高いが，スレッド間でデータ領域を共有できないため，大規模な知識ベースには向かない．

Inside Prolog のマルチスレッド化では，大規模な知識ベースにも対応でき，SMP システムでのスループット向上に有利な (2) の方式を採用した．マルチスレッドライブラリには，POSIX threads 準拠のライブラリや Windows スレッドライブラリを利用する．

4.2 実行モデル

マルチスレッド化された Prolog 処理系（以下，特に混乱のない限り，改めて Inside Prolog と呼ぶ）の実行モデルを**図 3** に示す．

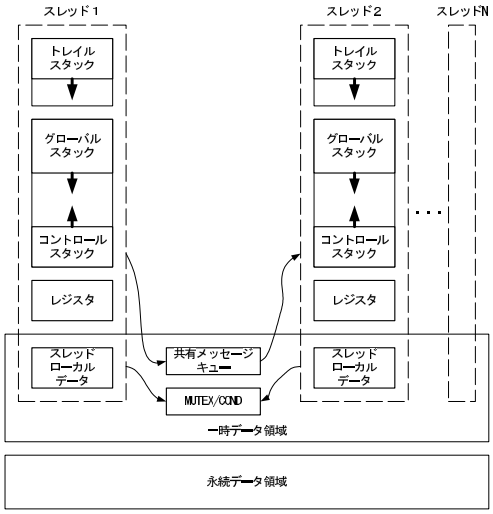


図 3 マルチスレッド化された Prolog の実行モデル
Fig. 3 Execution model of multi-threaded Prolog.

変数の共有 スレッド間での変数共有は，バックトラックの扱いや抽象機械の構成に大きな変更を必要とする[9]．また，各スレッドの推論処理は，共通して参照されるデータ（たとえば薬剤情報のマスタデータ）を実装上の工夫として共有するような場合を除いて，相互に干渉することもない．そこで，変数はスレッド間で共有できないものとし，各スレッドの実行過程は，バックトラックを含めて，従来のシングルスレッドの場合と同様とする．

通信方式 スレッド間の通信方式には，Linda のタプル空間（tuple space）[36] に代表される黒板モデルに基づく方式[9),11),12)] やメッセージ指向の方式[8),10),33)] が知られている．ここでは，多数のスレッド間での通信を多用しないため，後者を採用する（共有メッセージキューと呼ぶ）☆1．送受信できるデータは変数を含む任意の項とし，変数は受信時に改名される．また，例外処理の拡張として，他スレッドから例外をあげる機構も用意する．

同期方式 スレッド間の同期には，POSIX threads に規定された mutex（mutual exclusion object），cond（condition variable，条件変数），および read/write mutex を基本的な機構として用いる☆2．これらは，一時データ領域上に作成され，スレッド間で共有される．

4.3 抽象機械の拡張

マルチスレッド化のための TOAM の主な変更点は，スレッド制御データと排他制御の導入である☆3．スレッド制御データは，従来，大域変数で管理していたスタック等をスレッド単位に管理するもので，その実装にはマルチスレッドライブラリのスレッドローカル記憶☆4を利用すればよい．一方，排他制御は，スレッドに共有されるシンボル表やデータ領域を参照，操作する際に必要になるが，これを多用すると性能低下を招き，排他粒度が大きいと SMP システムでの並列性が低下する．

TOAM における排他制御は，(1) 一時データ領域に退避される例外情報を扱う `catch/3`，`throw/1` の処理，(2) 述語呼び出し命令 `call` 等の書き換え処理，および (3) シンボル表を参照する述語呼び出し命令 `call_indirect` 等の処理に必要である．(1) は例外発生時のみのため，推論性能と並列性に大きな影響を与えることはない．(2) はバイトコード中の `call` 命令等の各出現に対して 1 度しか発生しないため，その影響は十分小さい．一方，(3) は，一時データ領域に配置されたプログラムを呼び出す処理であり，いくつかの不可分な手順からなる．このため，述語の定義と実行とをマルチスレッド環境下でも安全に行えるようにするには，一般的に排他制御が必要である．しかし，この排他制御は，推論性能と並列性に大きく影響する．また，述語の定義と実行とを並行させる場合，アプリケーションプログラムによる明示的な排他制御で，述語定義の一貫性を保つのが一般的である．このため，TOAM による排他制御の省略が実用上問題となる可能性は低い．

そこで，static プログラムと incremental プログラムは TOAM による排他制御を行わない☆5．一方，interpretive プログラムは，indexing 用ハッシュ表の作成と検索，および logical database update 遵守のために，`clause/2`，`assertz/1` 等で排他制御を行う．この結果，たとえば知識ベースシステムの推論機構は，static プログラムを用いることで，高い推論性能と並列性を得られる．また，推論ルールから生成する実行

☆1 共有メッセージキューには `smq_create/1`，`smq_send/2`，`smq_receive/2` 等の述語が用意されている．たとえばこれは，推論処理を行うスレッドと実行時例外を遠隔からデバッグするためのスレッドとの間の通信に使用されている．

☆2 たとえば，mutex には `mutex_create/1`，`with_mutex_lock/2` 等の述語が用意されている．

☆3 TOAM の拡張に加えて，シンボル表やデータ領域を参照，操作する組み込み述語の多くに，排他制御のための変更が必要なことはいうまでもない．

☆4 たとえば，POSIX threads の `pthread_getspecific()` 関数群や Windows の `TlsGetValue()` 関数群である．

☆5 したがって，incremental プログラムを要求駆動で読み込むような場合は，推論処理の実行と競合しないように，アプリケーションプログラムによる明示的な排他制御が必要である．

コードは，知識ベースの開発中や知識処理サーバでの試験運用中は，incremental プログラムを用いて編集，翻訳，実行に要する時間のバランスをとり，知識処理サーバでの実運用では static プログラムを用いて推論性能を最優先にできる．一方，推論性能や並列性を犠牲にしても logical database update 遵守が重要な場合は，interpretive プログラムを用いる．このように，推論性能や並列性，述語定義の一貫性等の観点から，知識ベースシステムにおける述語の役割に応じた最適なプログラムコード表現を選択できる．

4.4 Java インタフェースの拡張

Inside Prolog は，JNI による双方向の Java インタフェースを持ち，`Resolver` class☆1が Java から Prolog を呼び出すための `boolean call(String name, Object[] args)` 等のメソッドを，また Prolog から Java を呼び出すための述語として `java_Call/3`☆2等を提供している．Prolog 処理系での Java のオブジェクトは，整数や浮動小数点数，文字列等の Prolog のデータ型に変換可能な一部を除いて，Java オブジェクトを指示する特殊な項☆3として表現され，一時データ領域に配置される．

この Java インタフェースは，後述の知識処理サーバから Prolog 処理系のマルチスレッド機能を利用するために，マルチスレッドに対応する必要がある．この際の主な変更点は，(1) Java オブジェクトを指示する項を一時データ領域に作成するときの排他制御，および (2) 一方から他方を呼び出すときのスレッドの初期化と解放である．(1) は，知識ベースシステムが参照するデータの多くが整数等の Prolog のデータ型に変換可能なものであるため，6.2.3 項で述べるように一般に影響は小さいと考えられる．そこで以下では，知識処理サーバの推論性能と並列性に関わりのある (2) について述べる．

Prolog 処理系は，推論処理の実行に際し，スタック等のリソースをスレッドごとに必要とする．このため，Java から Prolog を呼び出す際は，まず当該スレッド用のリソースを初期化し，呼び出し後にこれを解放する必要がある☆4．ところが，Java から Prolog を繰り返し呼び出す場合に，この初期化と解放を毎回行うのは非効率である．そこで，リソースの初期化と解放のための `attachThread()` および `detachThread()` メソッドを `Resolver` class に新たに導入し，Prolog の呼び出しとは独立に初期化と解放を可能にする．これにより，5.2 節で述べる Prolog エンジンの実装例のように，リソースの初期化と解放を最少化して Prolog の呼び出しを最適化できる．一方，Prolog から Java を呼び出す場合は，`java_Call/3` 述語等で必要に応じて JNI の `AttachCurrentThread()` 関数を用いて Java スレッドを初期化し，Prolog スレッドの終了時に `DetachCurrentThread()` 関数を用いて Java スレッドを解放する．これらの実装上の工夫により，Java と Prolog とが同一スレッドで動作し，相互の機能を効率的に利用できるようになる．

5. 知識処理サーバの設計と実装

5.1 知識処理サーバの概要

知識処理サーバと業務システムとのインタフェースには，クラスライブラリが整備された Java を用いる．これにより，J2EE，RMI 等の標準的な Java 技術や SOAP，CORBA 等のプログラミング言語独立な分散オブジェク技術への対応を容易にし，知識処理サーバの相互運用性を高める．

Prolog 処理系は，前述のようにスタック等のリソースを各スレッドごとに必要とする．限られたメモリ空間では，同時に作成できるスレッドも限られるため☆5，そのスレッド総数を管理する必要がある．しかし，既存のアプリケーションフレームワークは，必ずしもこのような機能を備えていない☆6．また，Prolog 処理系に加えて，知識ベースシステムも各スレッドごとのリソースを必要とする場合がある☆7．これらのリソースは，その初期化と解放にデータ領域の排他制御を必要とするため，1 つの処理要求ごとに初期化と解放を繰り返すと，推論性能と並列性の低下を招く．このため，semaphore 等を用いたスレッド数の管理だけでは必ずしも十分でない．

そこで，業務システムの処理要求を受け付けるスレッドと推論処理を行うスレッドとを分離し，後者を

☆1 tuProlog の `Prolog` class や SICStus Prolog の `SICStus` class に相当する．

☆2 たとえば `java_Call('Class', forName('org.postgresql.Driver'), Driver)` のように用いる．

☆3 Java オブジェクトを指示する項は，入出力用のストリーム等と同様に変更可能な内部状態を持ち，それへの操作は一般に副作用をともなう．

☆4 Java スレッドの終了を検知する標準的な手段が提供されていないために，Java スレッドの終了時に自動的に解放することは一般にできない．

☆5 32 bits 中 3 bits をデータ型の識別に用いるため，スタックとデータ領域等を合わせて 2 Gbytes 以下に制限される．

☆6 たとえば，J2EE 用アプリケーションサーバでは，各 Bean ごとのスレッド数を指定できても，Prolog 処理系を利用する Bean の総スレッド数を指定できない場合がある．

☆7 たとえば，臨床判断支援システムの推論機構では，高速化のためにスレッドごとのキャッシュやハッシュ表を多用している．

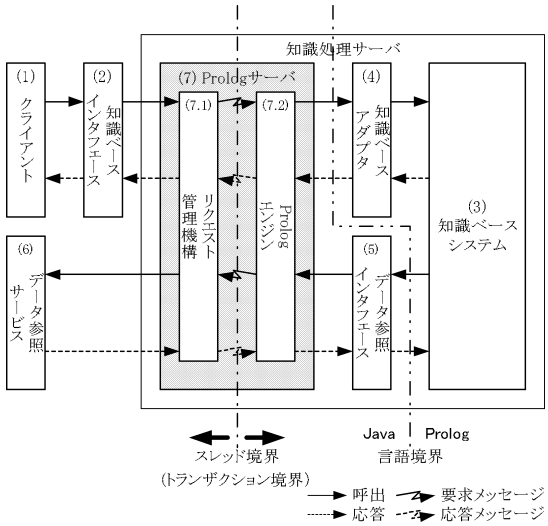


図 4 知識処理サーバのシステム構成
Fig. 4 System configuration of knowledge processing server.

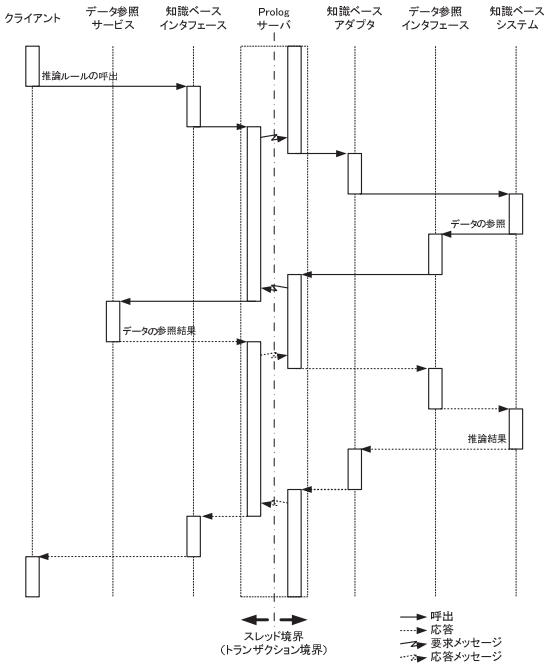


図 5 知識処理サーバのイベントトレース
Fig. 5 Event trace of knowledge processing server.

独自に管理して再利用する仕組みを用意する．これにより，Prolog 処理系と知識ベースシステムとは，排他制御をともなうリソースの初期化と解放を繰り返す必要がなくなる．しかしこの手法には，(1) 2 つのスレッド間の通信のために同期処理が必要になり，また (2) J2EE のようにトランザクション☆がスレッドと関連付けられている場合に，業務システムのトランザクションを別スレッドで動作する推論ルールが継承できない，という問題がある．(1) については，同期処理による性能低下より，スケーラビリティを高めるための推論処理の並列性を優先することにした．(2) に対しては，処理要求を受け付けたスレッドにデータの参照を委譲する仕組みを用意し，知識ベースシステムによるトランザクションの継承を可能にする☆☆．

知識処理サーバのシステム構成を**図 4** に示す．図中，一点鎖線は，スレッドおよびトランザクションの境界を，また二点鎖線は Java と Prolog との言語境界を表す．クライアント (1) は，病院情報システムのような知識ベースシステムを利用する業務システムを表す．知識ベースインタフェース (2) は，Session Bean 等を用いて知識ベースシステムの機能を業務システムに提供する Java プログラムである．知識ベースシステム (3) は，臨床判断支援のような Prolog で記述されたプログラムである．知識ベースアダプタ (4) は，知識ベースインタフェースと知識ベースシステムとの間でのデータ型やデータ構造の変換☆☆☆，障害発生時の代替ルールへの切替え等の運用管理等を行うための Prolog プログラムである．データ参照インタフェース (5) は，業務システムが提供する各種データを JNDI や JDBC 等のデータ参照サービス (6) を通して参照するための Java プログラムである．Prolog サーバ (7) は，異なるスレッド上で動作する知識ベースインタフェースと知識ベースシステムとの間の通信を仲介する汎用の機構である．**図 5** にクライアントと知識ベースシステムとの間でのイベントトレースの一例を示す．

知識処理サーバをこのような構成とすることで，2 章にあげた業務システムのための要件に対して，1) 知識ベースインタフェースに Java を用いることによって業務システムとの相互運用性を，2) マルチスレッド対応の Prolog 処理系と，これを Java から呼び出す仕組みとを最適化することによってスケーラビリティを，3) データ参照サービスの利用を Prolog サーバに委譲することによってトランザクション処理への対応を，さらに 4) 知識ベースアダプタによって運用管理

☆ 本論文では，簡単のためにトランザクションとトランザクションコンテキストとを特に区別しない．

☆☆ トランザクションの継承は，トランザクションをスレッドと関連付けて管理していればよく，J2EE に限られるものではない．

☆☆☆ たとえば，Java の String 型は，Inside Prolog の Java インタフェースにおいて Prolog の文字列型に変換されるため，これをさらにアトム型に変換する場合などである．

の支援を，それぞれ満たすことができる．

5.2 Prolog サーバの構成

Prolog サーバは，リクエスト管理機構（7.1）と Prolog エンジン（7.2）とからなる．これらは，互いに異なるスレッド上で動作し，双方向のメッセージ通信により他方の機能を呼び出す．リクエスト管理機構の役割は，複数のスレッド上で動作する Prolog エンジンの管理，知識ベースインタフェースから送られる処理要求の Prolog エンジンへの送信，およびデータ参照インタフェースから送られるデータ参照要求の処理である．Prolog エンジンのスレッド数は，推論ルールの実行に必要な Prolog 処理系のスタックの大きさや単位時間あたりの処理要求数，推論に要する時間等を基にユーザが指定する☆．

一方，Prolog エンジンの役割は，リクエスト管理機構から送られる処理要求を知識ベースアダプタを経由して知識ベースシステムに送ること，および知識ベースシステムが業務システムのデータを参照する際に，データ参照インタフェースからの参照要求をリクエスト管理機構に送信することである．リクエスト管理機構が送信した処理要求は，要求待ちの状態にある任意の Prolog エンジンによって受信され，これと同一スレッド上で動作する Prolog 処理系によって処理される．

図 6 は，Prolog エンジン実装の概要である．処理要求のメッセージは，`PrologRequest` class のインスタンスとして表している．Prolog エンジンは，スレッドの開始時に `attachThread()` で Prolog 処理系のリソースを初期化し，スレッド終了時に `detachThread()` で解放する．これにより，Java から Prolog への反復的な呼び出しを効率化している．

一方，データの参照要求は，`PrologCallback` interface の実装として表している☆☆．この interface 実装は，先に作られた `PrologRequest` class のインスタンスを介して Prolog エンジンからリクエスト管理機構に送られ，ここで実行される．**図 7** は，JDBC によりデータベースを参照するデータ参照インタフェースの実装例である．この例では，`PrologCallback` interface を実装した無名クラスの `run()` メソッドがリクエスト管理機構のスレッドで実行される．

5.3 運用管理の支援

Prolog サーバは，待機中の Prolog エンジン数，待ち状態の処理要求数，データ領域の消費量等の稼働状況に関する情報を提供する．これを用いて，知識ベースシステムへの負荷やデータ領域の占有率等を把握し，計算機リソースの最適な配分等を検討できる．

```
public class PrologEngine implements Runnable {
  private Resolver resolver = new Resolver();
  ...
  public void run() {
    // Prolog 処理系のリソースを初期化
    resolver.attachThread();
    try {
      while(isRunning()){
        // 新たな処理要求を受け取る
        PrologRequest request = getRequest();
        if(request == null){
          continue;   // タイムアウト
        }
        // 知識ベースアダプタを通して推論ルールを呼出す
        resolveRequest(request);
      }
    } finally {
      // Prolog 処理系のリソースを解放
      resolver.detachThread();
    }
  }
  ...
}
```

図 6 Prolog エンジン実装の概要
Fig. 6 Overview of Prolog engine implementation.

知識処理サーバには，知識ベースに障害が発生した際に，業務システムへの影響を最小限に抑えられるように，代替ルールに切り替えたり，特別な例外処理を施す機能が求められる．しかし，このような機能を汎用の Prolog サーバに組み入れることは難しい．知識ベースアダプタは，データ型の変換のように業務システムと知識ベースシステムとの間を調整する役割に加えて，知識ベースに障害が発生した際の運用管理の役割も持つ．たとえば，臨床判断支援システムでは，稼働中の知識ベースに対して，障害が発生した推論ルールをあらかじめ用意した代替ルールに読み変える機能☆☆☆等を提供している．

5.4 知識処理サーバの実装

知識処理サーバは，J2EE アプリケーションフレー

☆ 6.2.4 項に示すように，病院情報システム CAFE では 5 つのスレッドを使用している．

☆☆ デザインパターンでは Command パターンと呼ばれるプログラミング手法である．

☆☆☆ 知識ベースの修正による対応も可能であるが，これには一般に時間を要するため，緊急時の対策としては不十分である．

```
public class SQLDatabaseAccessor {
  ...
  public Object[] execute(final String sql)
    throws Exception {
    PrologEngine.callCallback(new PrologCallback() {
        public Object run()  // データ参照の実行
          throws Exception {
          Connection conn = dataSrc.getConnection();
          Statement stmt = conn.createStatement();
          stmt.execute(sql); // SQL 文の実行
          ResultSet res = stmt.getResultSet();
          ...                // 検索結果の処理
          return null;
        }
      });
    ...
  }
  ...
}
```

図 7 データ参照インタフェースの実装例
Fig. 7 Sample implementation of a data access interface.

ムワーク上の実装と，単独アプリケーションによる実装とが可能である．前者では，Inside Prolog の処理系を DLL（Dynamic Link Library）として提供し，これを JNI を通して Java VM に動的に結合する．J2EE 用アプリケーションサーバでは，Solaris，Linux，および Windows 用の WebLogic，JBoss，JEUS での動作が確認されている．一方，後者では，Inside Prolog に Java VM の DLL を動的に結合し，Prolog で記述された知識ベースシステムから Java で記述された RMI，SOAP，CORBA 等の機能を利用する．病院情報システム CAFE では，業務フローを扱うソフトウェアが J2EE アプリケーションサーバ上に構築されているため，前者を採用している．

6. システムの評価

本章では，まず Prolog 処理系のマルチスレッド機能について，そのオーバヘッドと並列性をベンチマークプログラムを用いた実験により評価する．次に，実験と運用実績に基づいて，知識処理サーバが 2 章にあげたスケーラビリティと頑健性の要件を満たしているか評価する．

6.1 実験の概要

6.1.1 実験 1（マルチスレッド化のオーバヘッド）

Prolog 処理系のマルチスレッド化によるオーバヘッドを評価するために，マルチスレッド化前後の

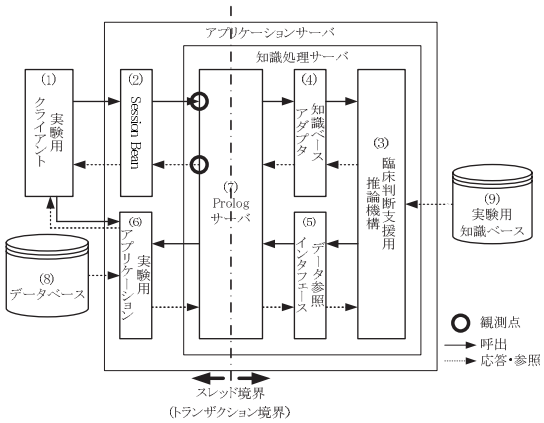


図 8 J2EE に適用した実験用知識処理サーバのシステム構成
Fig. 8 Experimental knowledge processing server applied to J2EE.

Prolog 処理系でベンチマークプログラムの経過時間（elapsed time）を計測した．ベンチマークプログラムには boyer，8 queens，qsort，takeuchi を使用し，プログラムコード表現は interpretive，incremental，static とした．Static プログラムは，永続データ領域に配置し，述語呼び出し命令の書き換えが完了した状態とした☆．実験には 1 CPU 構成の Sun V880 を使用した．

6.1.2 実験 2（SMP システムにおける並列性）

SMP システムにおける各プログラムコード表現の並列性を評価するために，CPU 構成を変えながらベンチマークプログラムの経過時間を計測した．CPU 構成は 1〜4（Sun V880）とし，1〜8 スレッドを生成して所定の回数だけ，同一プログラムを並列に実行した．

6.1.3 実験 3（知識処理サーバのスケーラビリティ）

知識処理サーバの SMP システムにおけるスケーラビリティを評価するために，これを J2EE に適用し，複数クライアントに対する推論処理の経過時間を計測した．アプリケーションは，病院情報システム CAFE をモデルとし，**図 8** に示すシステム構成とした．知識ベースインタフェースには CAFE 用 Session Bean (2) を用い，知識ベースシステムには臨床判断支援システムの推論機構 (3) を用いた．実験用アプリケーション (6) は，約 10 個のフィールドを持つ Entity Bean☆☆を提供し，その内容をデータベース (8) に格納する．実験用クライアント (1) は，まずこの Entity

☆ 他の実験も同様である．
☆☆ Entity Bean は病院情報システム CAFE における処方や注射等のオーダに対応する．

Bean を 20 個☆作成し，次に Session Bean を通して実験用知識ベース (9) に格納された推論ルールを呼び出す．この際，知識ベースアダプタは，Java の String 型から Prolog の文字列型に変換されたルール名をさらにアトム型に変換する処理や，例外発生時に例外情報を説明用文字列に変換する処理等を行う．推論ルールは，Java で記述されたデータ参照インタフェースを通して，この 20 個の Entity Bean を検索し，そのフィールド値を参照する．

実験では，アプリケーションサーバを 1〜4 CPU 構成の Sun V880 に，またデータベースを 2 CPU 構成の Sun Ultra 60 に，それぞれ配置した．推論機構は 1〜8 スレッドとし，実験用クライアントは 1〜32 個を最大 8 台の計算機上で動かした．経過時間は，Session Bean から Prolog サーバのメソッドを呼び出す箇所（図 8 中○印）で計測した．

6.1.4 実験 4（知識処理サーバのスケーラビリティ）

知識処理サーバの SMP システムにおけるスケーラビリティを評価するには，1〜4 CPU では必ずしも十分ではないため，32 論理 CPU 構成の Sun T2000 を用いて 6.1.3 項同様の実験を行った．クライアント数，スレッド数および論理 CPU 数は 1〜32 とした．なお，Sun T2000 は，8 つの CPU コアと 1 つの FPU☆☆を持ち，CMT（Chip Multi-Threading）により 1 コアあたり 4 スレッドを並列処理できるとされる．したがって，この実験結果を物理 CPU を持つ SMP システムにそのまま適用することはできないが，およその傾向は確認できると考えられる．

6.2 実験結果と考察

6.2.1 マルチスレッド化のオーバヘッド

実験 1 の結果について，マルチスレッド版による経過時間をシングルスレッド版に対する比として**図 9** に示す．この結果から，約 20%のオーバヘッドに収まっていることが分かる．これは，文献 10) に示された結果とほぼ同等である．また，その原因も，文献 10) の場合と同様に，排他制御，およびスレッド制御データの導入によりスタック等の管理が大域変数からポインタによる間接参照に変更されたことによるものと考えられる．

6.2.2 SMP システムにおける並列性

実験 2 の結果について，8 queens と boyer の interpretive および static プログラムの実行結果をそれぞれ**図 10**，**図 11**，**図 12** に示す．経過時間は，各 CPU 構成における 1 スレッドによる経過時間を基準とした比で表している．

8 queens の interpretive プログラム（図 10）は，スレッド数増加にともない経過時間が一様に増加している．また，3 CPU 以上では経過時間が 1 CPU 時より

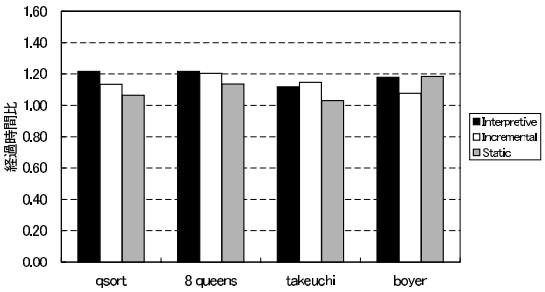


図 9 シングルスレッド版に対するマルチスレッド版の経過時間比
Fig. 9 Elapsed time ratios of multi-threaded version to single threaded version.

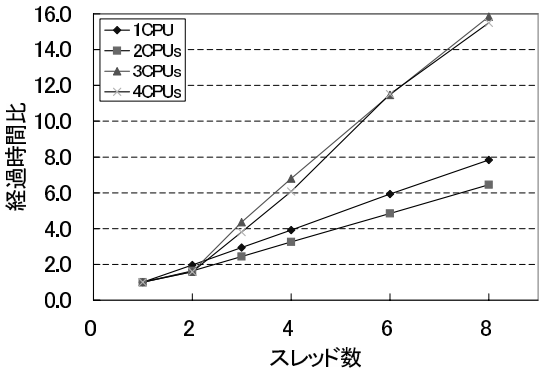


図 10 8 queens の経過時間比（Interpretive プログラム）
Fig. 10 Elapsed time ratios of the interpretive program of the 8-queens benchmark.

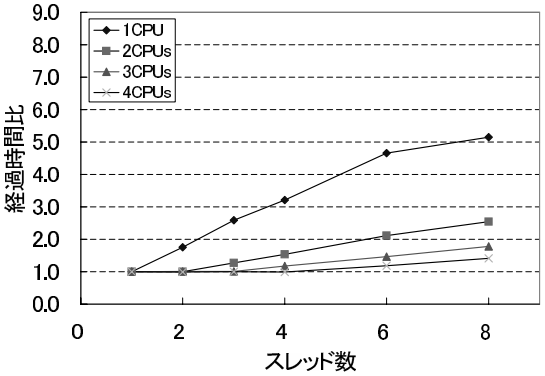


図 11 8 queens の経過時間比（Static プログラム）
Fig. 11 Elapsed time ratios of the static program of the 8-queens benchmark.

☆ Entity Bean の数は，中症度の患者に対して一日あたり 10〜20 薬剤のオーダが出されることが多いとされることから決めた．

☆☆ 知識ベースアダプタと推論機構の一部で浮動小数点演算を用いている．

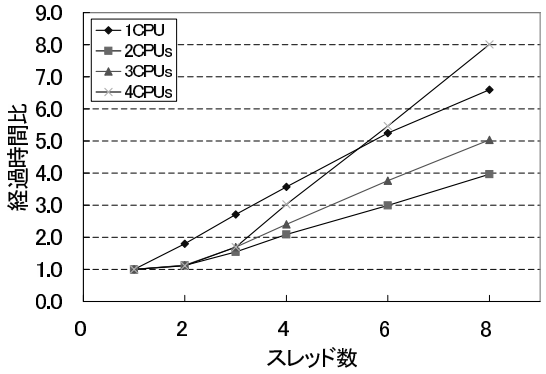


図 12 Boyer の経過時間比（Static プログラム）
Fig. 12 Elapsed time ratios of the static program of the boyer benchmark.

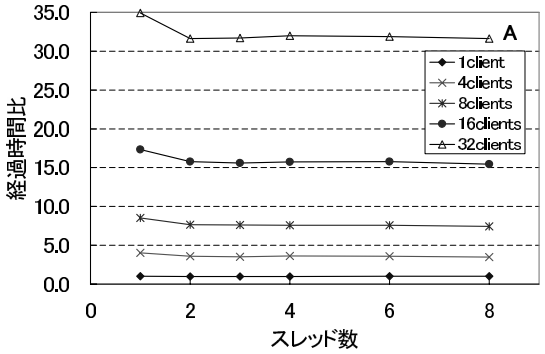


図 13 知識処理サーバにおける Session Bean の経過時間比（1 CPU）
Fig. 13 Elapsed time ratios of the Session Bean of the knowledge processing server on 1 CPU machine.

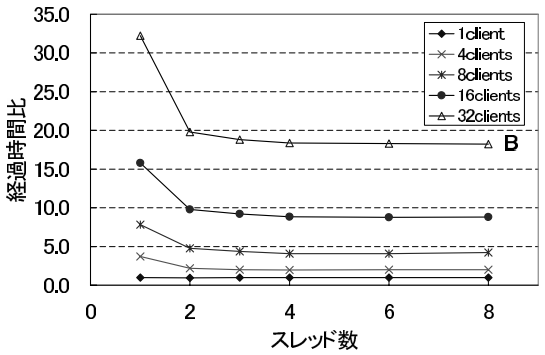


図 14 知識処理サーバにおける Session Bean の経過時間比（2 CPU）
Fig. 14 Elapsed time ratios of the Session Bean of the knowledge processing server on 2 CPUs machine.

も逆に増加している．これは，4.3 節で述べたように，interpretive プログラムがデータ領域を排他制御するためと考えられる．一方，static プログラム（図 11）では，スレッド数が同じなら CPU 数の増加にともない経過時間が減少している．特に，スレッド数が CPU 数以下の場合，経過時間に変化が見られず，高い並列性が確認できる．Incremental プログラムでも，最適化の違いによる実経過時間の増加を除いて，static プログラムとほぼ同様の結果が得られた．また，qsort や takeuchi でも同様である．

これに対し boyer の static プログラム（図 12）は，8 queens 等と異なり，interpretive プログラムと類似の結果となった．これは，boyer がシンボル表を操作する `functor/3` を多用するため，シンボル表の排他制御が原因で並列性が低下しているためと考えられる．

以上の結果から，incremental および static プログラムでは，排他制御をともなう述語の使用頻度が低ければ，SMP システムで高い並列性が得られることを確認できた．

6.2.3 知識処理サーバのスケーラビリティ

実験 3 の結果について，Session Bean が処理に要した経過時間を**図 13**，**図 14**，**図 15** に示す．経過時間は，1 クライアントに対する 1 スレッドによる経過時間（約 27 ミリ秒）を基準とした比で表している．同様にして，実験 4 の結果について，4 論理 CPU における経過時間を**図 16** に，32 クライアント，32 スレッドにおける各論理 CPU 数での経過時間を**図 17** に示す．

実験 3 の結果から，クライアント数の増加にともない経過時間は大きくなるが，スレッド数と CPU 数とを増やすことでスループットを改善できることが分かる．たとえば，32 クライアント，8 スレッドの場合，CPU 数を 2，4 と増やすことで 1 の場合（図 13 点 A）と比べて経過時間が約 0.51（図 14 点 B），0.28（図 15 点 C）倍にそれぞれ改善された．しかし，これは CPU 数に応じた改善率には及ばず，またスレッド数の増加による経過時間の改善効果も次第に低下している．実験に用いた推論機構と推論ルールとは，主に static プログラムであり☆，プログラム表現による排他制御を含まない．したがって，`functor/3` 同様の排他制御をともなう述語☆☆が並列性を妨げる一因と考えられるが，boyer のように CPU 数の増加にともない経過時間が増加する傾向は 4 CPU までは見られず，その影響は小さいものと考えられる．

一方，図 16 から，1 つの FPU と CMT による論理 CPU を持つ T2000 では，同数の物理 CPU を持つ V880 と比べると，スレッド数によるスループット

☆ 推論ルールの実行コードは，知識ベースの開発中は incremental プログラムで表現し，知識処理サーバ上に読み込む段階で static プログラムに翻訳している．

☆☆ Java インタフェースと推論機構の一部が使用している．

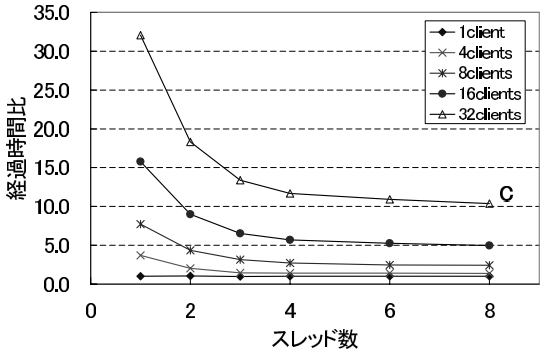


図 15 知識処理サーバにおける Session Bean の経過時間比（4 CPU）

Fig. 15 Elapsed time ratios of the Session Bean of the knowledge processing server on 4 CPUs machine.

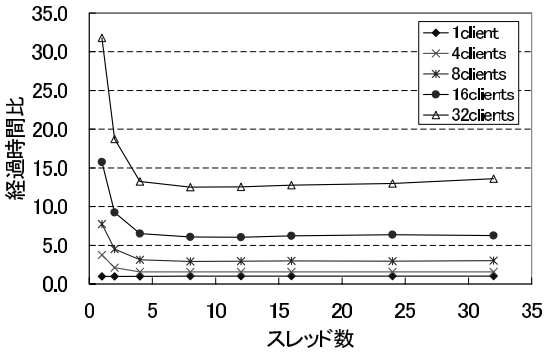


図 16 知識処理サーバにおける Session Bean の経過時間比（4 論理 CPU）

Fig. 16 Elapsed time ratios of the Session Bean of the knowledge processing server on 4 logical CPUs machine.

の改善効果が劣ることが分かる．このため，実験 4 の結果を同数の物理 CPU を持つ SMP システムにそのまま適用することはできない．しかし，図 17 において，論理 CPU と物理 CPU とが同数になる 8 まで顕著な改善効果が見られ，論理 CPU 数が 32 のときに 1 の場合と比べて経過時間が約 0.13 倍まで改善されている．したがって，これらの実験結果から，知識処理サーバが少なくとも 8〜16 CPU 程度までスケーラビリティを有していることが確認できた．

経過時間に対する排他制御の影響は，推論ルールにおけるデータベース検索等の待ち時間が長くなれば，相対的に小さくなる．実際に病院情報システム CAFE で運用中の推論ルールは，患者の処方歴☆や検体検査結果等を参照し，この処理が経過時間の多くを占める

☆ 処方する薬剤に依存するが，多くの場合 1〜14 日分，まれに 1〜3 カ月程度分のオーダを検索する必要がある．

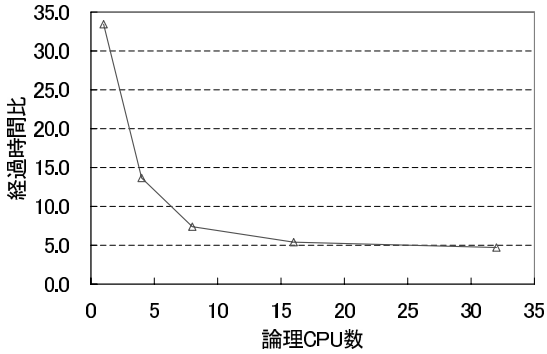


図 17 論理 CPU 数に対する Session Bean の経過時間比の推移

Fig. 17 Elapsed time ratios of the Session Bean of the knowledge processing server against the number of logical CPUs.

ことが分かっている．したがって，実際の臨床判断支援システムでは，この排他制御の影響はさらに小さいものと考えられる．

6.2.4 システムの頑健性

システムの頑健性をこれらの評価実験から定量的に示すことは難しい．そこで，病院情報システム CAFE における臨床判断支援システムの運用実績を基に評価する．CAFE は，32 CPU 構成の Sun Fire 15K（アプリケーションサーバ用ドメイン 24 CPU，データベース用ドメイン 8 CPU）上で，知識処理用アプリケーションサーバをクラスタリングされた 3 つのプロセスで構成し，各々は 5 つの Prolog エンジンスレッドを持つ．このシステム構成で，1 日あたり約 1000 発行される処方箋に対して，1 処方箋の禁忌検証をほぼ 1 秒以内で推論している．本システムは，この禁忌検証サービスを 1 年以上継続して提供しているが，知識処理サーバが原因の障害はこれまで 1 度も確認されていない．このことから，知識処理サーバは，実用上十分な頑健性を備えていると考えられる．

7. おわりに

本論文では，Prolog 上に開発された知識ベースシステムを様々な業務システムで運用するための枠組みとして，知識処理サーバを提案した．知識処理サーバは，新たに開発したマルチスレッド対応の Prolog 処理系と Java 処理系とを組み合わせることにより，業務システムとの相互運用性，トランザクション処理への対応，および運用管理の支援を実現し，知識ベースシステムの汎用的な運用環境を提供できることを示した．

Prolog 処理系のマルチスレッド機能，および知識処理サーバのスケーラビリティと頑健性を，実験と運用実績を通して評価した．実験の結果，Prolog 処理

系のマルチスレッド機能がオーバヘッドを約20%以内に抑えながらSMPシステムで高い並列性を得られること，知識処理サーバがSMPシステムにおいて8〜16 CPU程度までスケーラビリティを有することを確認した．また，病院情報システムCAFEにおける臨床判断支援システムの運用実績から，知識処理サーバが実用上十分な頑健性を備えていることも確認した．

病院情報システムCAFEでは，業務フローを扱うソフトウェアが大規模なために，Java VMとして業務フロー用64 bitsと知識処理サーバ用32 bitsの2種類を組み合わせて変則的に運用している．業務システムの大規模化に対応するには，Prolog処理系の64 bits化が必要である．また，知識処理サーバは，SOAP等を用いることで，J2EE同様に多用される.NETにも適用できる．しかし，業務システムとのインタフェースにC#を用いることができれば，.NETに基づいた業務システムへの適用がさらに容易になるものと考えられる．これらは今後の課題である．

謝辞 貴重なご助言とご協力を賜りました筑波大学附属病院副病院長五十嵐徹也教授，熊本大学医学部附属病院高田彰助教授，京都大学医学部附属病院長瀬啓介助教授，ならびに筑波大学附属病院関係者の皆様に深く感謝申し上げます．

参 考 文 献

1) Kaplan, B.: Evaluating informatics applications — clinical decision support systems literature review, *International Journal of Medical Informatics*, Vol.64, No.1, pp.15–37 (2001).
2) 上野晴樹：エキスパート・システム概論，情報処理，Vol.28, No.2, pp.147–157 (1987).
3) 小林重信：プロダクションシステム，情報処理，Vol.26, No.12, pp.1487–1496 (1985).
4) Toussaint, A.: Java Rule Engine API, *JSR-94* (2003).
5) YASU Technologies: *QuickRules* (2005). http://yasutech.com/products/quickrulesse/index.htm
6) ILOG, Inc.: ILOG JRules (2006). http://www.ilog.com/products/jrules
7) Drools Project: Drools (2006). http://drools.org
8) Eskilson, J. and Carlsson, M.: SICStus MT — A Multithreaded Execution Environment for SICStus Prolog, *Principles of Declarative Programming: 10th International Symposium*, Lecture Notes in Computer Science, Vol.1490, pp.36–53, Springer-Verlag GmbH (1998).
9) Carro, M. and Hermenegildo, M.: Concurrency in Prolog Using Threads and a Shared Database, *International Conference on Logic Programming*, pp.320–334 (1999).
10) Wielemaker, J.: Native Preemptive Threads in SWI-Prolog, *Logic Programming*, Lecture Notes in Computer Science, Vol.2916, pp.331–345, Springer Science+Business Media (2003).
11) Denti, E., Omicini, A. and Ricci, A.: tuProlog: A Light-Weight Prolog for Internet Applications and Infrastructures, *Proc. Practical Aspects of Declarative Languages, 3rd International Symposium, PADL 2001*, Lecture Notes in Computer Science, Vol.1990, pp.184–198, Springer-Verlag GmbH (2001).
12) Tarau, P.: Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog, *Proc. 4th International Conference on the Practical Applications of Intelligent Agents and Multi-agent Technology*, pp.109–124 (1999).
13) Katamine, K., Umeda, M., Nagasawa, I. and Hashimoto, M.: Integrated Development Environment for Knowledge-Based Systems and Its Practical Application, *IEICE Trans. Information and Systems*, Vol.E87-D, No.4, pp.877–885 (2004).
14) 手越義昭，長澤 勲，前田潤滋，牧野 稔：建築物設計における小規模な組合せ選択問題の一解法，日本建築学会計画系論文集，No.405, pp.157–165 (1989).
15) 長澤 勲，前田潤滋，手越義昭，牧野 稔：建築設計支援システムにおける小規模な組合せ選択問題のためのプログラミング手法，日本建築学会構造系論文集，No.417, pp.157–166 (1990).
16) Umeda, M., Nagasawa, I. and Higuchi, T.: The Elements of Programming Style in Design Calculations, *Proc. 9th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pp.77–86 (1996).
17) 古川由美子，上野道雄，長澤 勲：健康管理支援システムHCSの開発と運用，医療情報学，Vol.10, No.2, pp.121–132 (1990).
18) 古川由美子，長澤 勲，上野道雄：健康管理支援システム，情報処理，Vol.34, No.1, pp.88–95 (1993).
19) Umeda, M. and Nagasawa, I.: Project Structure and Development Methodology Toward The IT Revolution — Lesson from Practice, *Proc. 4th Joint Conference on Knowledge-Based Software Engineering*, pp.1–8 (2000).
20) ISO/IEC: 13211-1 Information technology — Programming Languages — Prolog — Part 1: General core (1995).
21) 長瀬啓介，高田 彰，五十嵐徹也，大内隆信，網野

貴文，大野国弘：Java 2 Enterprise Edition を用いた推論エンジンを有する病院情報システムの開発，第 23 回医療情報学連合大会論文集，pp.1-G-2-2 (2003).
22) 長瀬啓介，長澤　勲，梅田政信，五十嵐徹也，高田彰，大野国弘，大内隆信，安光正則：医療サービスの質的・経営的変革を誘導することを意図した臨床判断支援論理の実装と運用，第 24 回医療情報学連合大会論文集，pp.2-E-1-3 (2004).
23) Ohno, K., Umeda, M., Nagase, K. and Nagasawa, I.: Knowledge Base Programming for Medical Decision Support, *Proc. 14th International Conference on Applications of Prolog*, pp.202–210 (2001).
24) 大野国弘，長澤　勲，梅田政信，長瀬啓介，高田彰，五十嵐徹也：臨床判断支援システムのための知識ベースの開発，第 23 回医療情報学連合大会論文集，pp.2-D-5-6 (2003).
25) 梅田政信，長澤　勲，大野国弘，片峯恵一：臨床判断支援のための知識ベース開発環境と推論エンジンの J2EE 実装，第 23 回医療情報学連合大会論文集，pp.3-C-1-5 (2003).
26) Bromley, H. and Lamson, R.（著），長谷川純一（訳）：LISP マシンプログラミング技法，マグロウヒル (1991).
27) Uchida, S., Yokota, M., Yamamoto, A., Taki, K. and Nishikawa, H.: Outline of the Personal Sequential Inference Machine: PSI, *New Generation Computing*, Vol.1, No.1, pp.75–79 (1983).
28) Ait-Kaci, H.: *Warren's Abstract Machine*, The MIT Press (1991).
29) Neng-Fa, Z.: Global Optimizations in a Prolog Compiler for the TOAM, *J. Logic Programming*, No.15, pp.265–294 (1993).
30) 片峯恵一，廣田豊彦，周　能法，長澤　勲：Prolog プログラムの C への変換，情報処理学会論文誌，Vol.37, No.6, pp.1130–1137 (1996).
31) 梅田政信，長澤　勲，伊藤公俊：標準部品にかかわる技術情報流通のための知識表現モデル，情報処理学会論文誌，Vol.38, No.10, pp.1905–1918 (1997).
32) Li, X.: A New Term Representation Method for Prolog, *The Journal of Logic Programming*, Vol.34, No.1, pp.43–57 (1998).
33) Clark, K., Robinson, P. and Hagen, R.: Multi-threading and message communication in Qu-Prolog, *Theory and Practice of Logic Programming*, Vol.1, No.3, pp.283–301 (2001).
34) IF Computer: MINERVA (2005). http://www.ifcomputer.co.jp/MINERVA
35) 福田　晃：並列オペレーティング・システム，情報処理，Vol.34, No.9, pp.1139–1149 (1993).
36) Carriero, N. and Gelernter, D.: Linda in Context, *Comm. ACM*, Vol.32, No.4, pp.444–458 (1989).

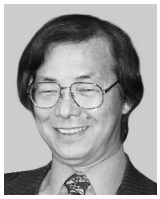
(平成 18 年 4 月 12 日受付)
(平成 19 年 2 月 1 日採録)

梅田　政信（正会員）

1982 年九州大学理学部物理学科卒業．1984 年同大学大学院工学研究科情報工学専攻修士課程修了．1984 年富士通（株）．1989 年長崎県工業技術センター．1990 年九州工業大学情報工学部助手．現在，同大学大学院情報工学研究科助手．設計支援システム，医療情報システム，知識ベースシステムのための開発支援環境等の研究開発に従事．精密工学会，プロジェクトマネージメント学会各会員．

片峯　恵一（正会員）

1992 年九州工業大学情報工学部機械システム工学科卒業．1994 年同大学大学院情報工学研究科修士課程修了．1994 年九州工業大学情報工学部助手．2002 年同大学大学院情報工学研究科助手．現在に至る．情報システムの仕様化，仕様記述言語，プログラムの自動生成，ソフトウェア開発支援環境等の研究開発に従事．電子情報通信学会，プロジェクトマネジメント学会各会員．



長澤　　勲（正会員）

1967 年九州大学工学部電子工学科卒業．1972 年同大学大学院工学研究科博士課程単位取得退学．1972 年九州大学中央計数施設講師．1989 年九州工業大学情報工学部教授．現在，同大学大学院情報工学研究科教授．工学博士．知識情報処理の立場から CAD/CAM，ロボット，医療システム等の研究開発に従事．人工知能学会，日本建築学会，精密工学会，電子情報通信学会，日本機械学会，日本設計工学会，日本ロボット学会，プロジェクトマネージメント学会各会員．

橋本 正明（正会員）

1968 年九州大学工学部電子工学科卒業．1970 年同大学大学院工学研究科修士課程修了．同年日本電信電話公社研究所入社．大型汎用計算機システム開発プロジェクトやソフトウェア工学研究に従事．1993 年九州工業大学情報工学部教授．現在，同大学情報工学研究科教授．ソフトウェア工学やプロジェクトマネジメント等の研究に従事．工学博士．プロジェクトマネジメント学会，電子情報通信学会，ソフトウェア科学会，人工知能学会，IEEE 各会員．

高田 修（正会員）

1983 年名古屋大学大学院工学研究科情報工学専攻修士課程修了．同年（株）豊田中央研究所入社．2001 年九州工業大学情報工学部助教授，現在，同大学大学院情報工学研究科．知識工学やプロジェクトマネジメント等に関する研究に従事．博士（工学）．人工知能学会，プロジェクトマネジメント学会，精密工学会，IEEE 各会員．