Rational. SDL

IBM

SDL Suite 設計手法のガイド

Methodology Guidelines

IBM Rational SDL Suite 6.3

Methodology Guidelines

日本語版

本書は、IBM Rational SDL Suite 6.3 および IBM Rational TTCN Suite 6.3 および新しい版で明記されていない限り、以降のすべてのリリースおよびモディフィケーションに適用されます。

著作権表示

本書は米国 IBM が提供する製品およびサービスについて作成したものであり、本書に記載の製品、サービス、または機能が日本においては提供されていない場合があります。日本で利用可能な製品、サービス、および機能については、日本 IBM の営業担当員にお尋ねください。本書で IBM 製品、プログラム、またはサービスに言及していても、その IBM 製品、プログラム、または サービスのみが使用可能であることを意味するものではありません。これらに代えて、IBM の知的所有権を侵害することのない、機能的に同等の 製品、プログラム、またはサービスを使用することができます。ただし、IBM 以外の製品とプログラムの操作またはサービスの 評価および検証は、お客様の責任で行っていただきます。

IBM特許権

IBM は、本書に記載されている内容に関して特許権 (特許出願中のものを含む) を保有している場合があります。本書の提供は、お客様にこれらの特許権について 実施権を許諾することを意味するものではありません。実施権についてのお問い合わせは、書面にて下記宛先にお送りください。

〒106-8711
東京都港区六本木 3-2-12
日本アイ・ビー・エム株式会社
法務・知的財産
知的財産権ライセンス渉外

本プログラムのライセンス保持者で、(i) 独自に作成したプログラムと その他のプログラム (本プログラムを含む) との間での情報交換、および (ii) 交換された情報の相互利用を可能にすることを目的として、本プログラムに関する情報を必要とする方は、製造元に連絡してください。

```
Intellectual Property Dept. for Rational Software|
IBM Corporation
1 Rogers Street
Cambridge, Massachusetts 02142
U.S.A.
```

本プログラムに関する上記の情報は、適切な使用条件の下で使用することができますが、有償の場合もあります。

本書で説明されているライセンス・プログラムまたはその他のライセンス資料は、IBM 所定のプログラム契約の契約条項、IBM プログラムのご使用条件、またはそれと同等の条項に基づいて、IBM より提供されます。

保証の不適用

以下の保証は、国または地域の法律に沿わない場合は、適用されません。IBM およびその直接または間接の子会社は、本書を特定物として現存するままの状態で提供し、商品性の保証、特定目的適合性の保証および法律上の瑕疵担保責任を含むすべての明示 もしくは黙示の保証責任を負わないものとします。国または地域によっては、法律の強行規定により、保証責任の制限が禁じられる場合、強行規定の制限を受けるものとします。

この情報には、技術的に不適切な記述や誤植を含む場合があります。本書は定期的に見直され、必要な変更は本書の次版に組み込まれます。IBM は予告なしに、随時、この文書に記載されている製品またはプログラムに対して、改良または変更を行うことがあります。

本書において IBM 以外の Web サイトに言及している場合がありますが、便宜のため記載しただけであり、決してそれらの Web サイトを推奨するものではありません。それらの Web サイトにある資料は、この IBM 製品の資料の一部ではありません。それらの Web サイトは、お客様の責任でご使用ください。

この文書に含まれるいかなるパフォーマンス・データも、管理環境下で決定されたものです。そのため、他の操作環境で得られた結果は、異なる可能性があります。一部の測定が、開発レベルのシステムで行われた可能性がありますが、その測定値が、一般に利用可能なシステムのものと同じである保証はありません。さらに、一部の測定値が、推定値である可能性があります。実際の結果は、異なる可能性があります。お客様は、お客様の特定の環境に適したデータを確かめる必要があります。

IBM 以外の製品に関する情報は、その製品の供給者、出版物、もしくはその他の公に利用可能なソースから入手したものです。IBM は、それらの製品のテストは行っておりません。したがって、他社製品に関する実行性、互換性、またはその他の要求については確証できません。IBM 以外の製品の性能に関する質問は、それらの製品の供給者にお願いします。

機密情報

IBM は、お客様が提供するいかなる情報も、お客様に対してなんら義務も負うことのない、自ら適切と信ずる方法で、使用もしくは配布することができるものとします。

本書には、日常の業務処理で用いられるデータや報告書の例が含まれています。より具体性を与えるために、それらの例には、個人、企業、ブランド、あるいは製品などの名前が含まれている場合があります。これらの名称はすべて架空のものであり、名称や住所が類似する企業が実在しているとしても、それは偶然にすぎません。

追加の法的通知が、本書で説明するライセンス付きプログラムに付随する「プログラムのご使用条件」に含まれている場合があります。

サンプルコードの著作権

本書には、様々なオペレーティング・プラットフォームでのプログラミング手法を例示するサンプル・アプリケーション・プログラムがソース言語で掲載されています。お客様は、サンプル・プログラムが書かれているオペレーティング・プラットフォームのアプリケーション・プログラミング・インターフェースに 準拠したアプリケーション・プログラムの開発、使用、販売、配布を目的として、いかなる形式においても、IBM に対価を支払うことなくこれを複製し、改変し、配布することができます。このサンプル・プログラムは、あらゆる条件下における完全なテストを経ていません。従って IBM は、これらのサンプル・プログラムについて信頼性、利便性もしくは機能性が あることをほのめかしたり、保証することはできません。

それぞれの複製物、サンプル・プログラムのいかなる部分、またはすべての派生的創作物にも、次のように、著作権表示を入れていただく必要があります。

© (お客様の会社名) (西暦年). このコードの一部は、IBM Corp. のサンプル・プログラムから取られています。© Copyright IBM Corp._(西暦年)

IBMの商標

IBM および関連の商標については、**www.ibm.com/legal/copytrade.html**. をご覧ください。これは、IBM が現在所有する米国における商標の最新リストです。以下は、International Business Machines Corporation の米国およびその他の国における商標です。

このページには、IBM が使用しているすべてのコモン・ロー商標は掲載されていません。IBM が販売している製品は多数あるため、コモン・ロー商標のうち、最も重要な商標のみを掲載しております。このページに商標が掲載されていなくても、それは IBM がその商標を使用していないということではなく、その製品が現在販売されていない、または関連する市場で、その製品が重要ではないということを意味するものではありません。

他社の商標

Adobe、Adobe ロゴ、PostScript は、Adobe Systems Incorporated の米国およびその他の国における登録商標または商標です。

Java およびすべての Java 関連の商標およびロゴは、Sun Microsystems, Inc. の米国およびその他の国における商標です。

Linux は、Linus Torvalds の米国およびその他の国における商標です。

Microsoft、Windows、Windows 2003、Windows XP、Windows Vista および / またはその他の Microsoft 製品は、Microsoft Corporation の米国およびその他の国における商標または登録商標です。

Pentium は、Intel Corporation の商標です。

はじめに

このマニュアルについて

このマニュアル*Methodology Guidelines*では、SDL Suiteを使ったソフトウェア開発においてSDLやその他の仕様を記述する際に役立つ設計手法を紹介しています。

SDL-92のオブジェクト指向の概念について説明し、SDL-88のシステムをSDL-92のシステムに変換する方法を紹介します。また、IBM Rational専用の拡張機能について説明し、SDL SuiteにおけるSDL、C言語、ASN.1のデータ型の使用方法を紹介します。さらに、複数のメンバーが存在するプロジェクトにおいて、ダイアグラム ファイルを管理する方法について説明し、事例を挙げます。

ドキュメント構成

ドキュメント構成に関する情報は、Release Guide日本語版のviiiページ、「ドキュメント」を参照してください。

表記規則

ドキュメントで使用している表記規則については、Release Guide日本語版のxページ、「表記規則」を参照してください。

カスタマー サポートへのお問い合わせ

IBM Rationalカスタマー サポートへの問い合わせに関する情報は、Release Guide日本語版のivページ、「IBM Rationalソフトウェア・サポートへの問い合わせ」を参照してください。

第 1 章 SDLを使用したオブジェクト指向設計

この章では、1992年度版のSDL言語に導入されたオブジェクト指向SDLの概念について説明します。Access Controlシステムという比較的簡単な事例を使って、仕様設計から最終的なSDL設計までの各工程を順次説明します。最初にSOMT方式に従った簡単なオブジェクト指向分析を行い、次にSDLを使用したオブジェクト指向設計を行います。

オブジェクト指向SDLの概念は、何種かのAccess Controlシステムのバージョンを開発することによって段階的に取り入れていきます。最初のバージョンでは、オブジェクト指向の概念であるブロック タイプとプロセス タイプのみを利用します。そして、最終のバージョンでは、継承、特殊化、仮想タイプ、タイプ ライブラリ、パッケージ ダイアグラムなどの高度なオブジェクト指向の概念を使用します。

この章では、『User's Manual』の第 69 章「SOMT Methodology Guidelines」で説明されている SOMT 方式の一部、主に、SOMT の設計アクティビティをベースにしたオブジェクト指向 SDL の使用方法に絞って説明します。

Access Controlシステムの要求条件

ここでは、システムを設計するための予備知識として要求条件分析の概要を説明します。したがって、要求条件分析に必要な手順をすべて説明するものではありません。

構築するシステムの説明

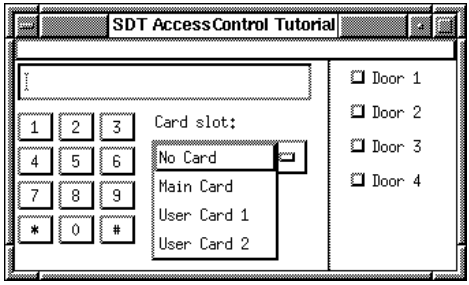
このアプリケーションを選択した理由は、組み込み型システムの好例であり、SDLと1992年版のSDLで導入されたオブジェクト指向の拡張仕様によって、仕様を記述するのに適しているためです。

Access Controlシステムは、ビルディングへのアクセスを制御するシステムです。ユーザーが事務所に入る際には、登録カードと個人コード（4桁）が必要です。カードと個人コードを入力するためのデバイスは、カード リーダー、キーパッド、ディスプレイで構成されます。

このシステムの特性は以下のようになります。

- 中程度のリアルタイム処理。
- 信号主体の処理。
- 単純なデータ表現。
- 単純な外部環境とのハードウェア インターフェイス。
- 非分散システム。
- 外部環境へのインターフェイスを変更することなく、新しいプログラム ロジックの追加によって、容易にシステムへの新機能の追加が可能。
- グラフィカルなユーザー インターフェイスを使ったシステムのシミュレーションが、ホスト環境で実施可能。図1を参照してください。

UNIXの場合



Windowsの場合

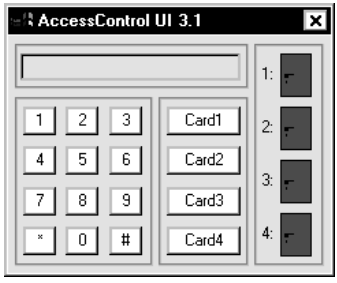


図1: Access Controlシステムへのグラフィカル インターフェイス

テキストによる要求条件の記述

ここでは、初期の工程で作成する要求条件について説明します。個々の要求条件は、収集して標準化された形式で再作成する方法が一般的です。これによって、要求条件分析と個々の要求条件の参照が容易になります。ただし、ここで説明する例は、要求条件の最初の部分のみを示します。

また、ここでは*機能に関する要求条件*のみを説明し、パフォーマンス、信頼性、可用性などの*機能以外の要求条件*については省略します。

基本的な要求条件

ハードウェア デバイスは、以下のコンポーネントで構成します。

- 8751マイクロプロセッサ
- 64 KBのプログラム メモリ（RAMまたはROM）

- 64 KBのデータ メモリ（RAM）
- クレジット カード用のカード リーダー
  カード リーダーは、トラック2を読み出します。データは、最も標準的な40ワード（1ワード: 5ビット）で保存します。
- キーパッド
  キーの配列は、標準の電話形式です。有効なキーは、0から9までの数字です。基本バージョンでは、機能キーの*と#は認識されません。
- ディスプレイ ユニット
  ディスプレイ ユニットは、各行16文字で2行を表示できます。
- 4つのLEDランプ
  4つの発光ダイオードが制御ドアの状態を示します。オフはドアが閉じている状態、オンは開いている状態を示します。

システムは、ユーザーに対する機能として以下の処理が可能でなければなりません。

- 標準的なクレジット カードの裏面に記録されたコードの読み取り。
- キーパッドから入力された4桁の個人コードの読み取り。
- カードと個人コードについて登録の有無の照会。
- システムが複数のドアを制御する場合、カードとコードが照会された後に、ユーザーが開くドアを選択する機能。

システムは、システム管理者に対する機能として以下の処理が可能でなければなりません。

- ユーザー カードと個人コードの登録。この際、各ユーザー カードに登録可能なコードは1種類のみ。
- システム起動時の管理者カードの登録。各システムで利用可能な管理者カードは1枚のみ。

また、システムは以下の共通の要求条件を満たす必要があります。

- システムの起動時に、1か所から4か所までのドアの制御を簡単に設定できること。

追加要求条件

システムは、ユーザーに対する機能として以下の処理が可能でなければなりません。

- 時刻の表示
- 現在有効な、カードのカテゴリ（以下参照）の表示

システムは、システム管理者に対する機能として以下の処理が可能でなければなりません。

- 特定のドアの開閉を禁止する機能（設定後は、管理者のみが開閉可能）
- すべてのドアの開閉を禁止する機能（設定後は、管理者のみが開閉可能）
- 1つまたはすべてのドアに対する開閉禁止の解除
- 1つまたはいくつかのドアに対する自由開閉の許可
- カードへのカテゴリの定義によって、24時間の時間帯内でさまざまなアクセス方法を指定する機
- 時刻の表示
- 現在の時刻の設定
- 特定のユーザー カードに対する使用禁止設定
- ユーザー カードへの使用禁止設定の解除

使用パターン

機能要求条件のうちで最も重要な部分は、複数の使用パターンを使って記述します。使用パターンによって、システムと外部環境との間の相互作用を表現することができます。また、機能要求条件を、より形式化された方法によって表現できます。

システムと通信する外部のエンティティは、通常、使用パターンの*アクタ*と呼ばれます。以下に、一般的なアクタを示します。

- 人
- 他のシステム
- ハードウェア

Access Controlシステムには、注目すべき2つのアクタ、すなわちユーザーと*管理者*が存在します。（ここでは検討を簡単にするために、ハードウェアをアクタの対象にしません。）

- ユーザー用の機能は、すべてのユーザーが利用可能なサービスです。サービスには、カード コードの読み取りや、4桁の個人コードの読み取りなどがあります。
- 管理者用の機能は、管理者など、適切な権限を持った人のみが利用できます。管理者用の機能には、新しいカードやコードの登録などのサービスがあります。

使用パターンは、テキストかMSC、または、それら両方を組み合わせて記述できます。ユーザーをアクタとして定義した使用パターンには、Open Door使用パターンがあります。図2のOpenDoor MSCを参照してください。この使用パターンは、「ドアを開く」という目的が達成されると終了します。

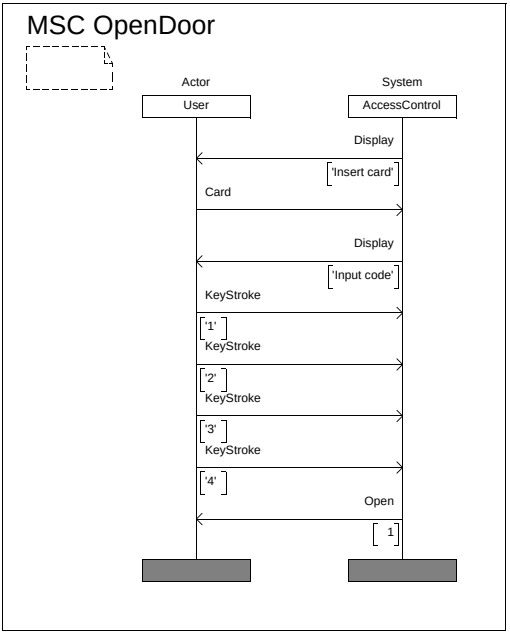


図2: OpenDoor 要求条件使用パターン

一般的に、使用パターンの要求条件を記述する際は、アクタとシステム間の相互作用のみを示します。これにより使用パターンを後で詳しく定義するときに、システム内部の振る舞いを追加記述することができます。

オブジェクト モデル

要求条件オブジェクト モデルは、比較的簡単なオブジェクト モデルであり、Access Controlシステムや外部環境の問題領域に存在する識別済みのエンティティを関連付けるために使用します。システムの外部環境には、*問題領域を理解するために必要になる*すべての事項が、その対象になります。通常は、システムの振る舞いを表現する使用パターンのアクタを、外部環境として定義します。オブジェクト モデルの目的は、問題の詳細に踏み込まずに簡単な概略を示すことにあります。

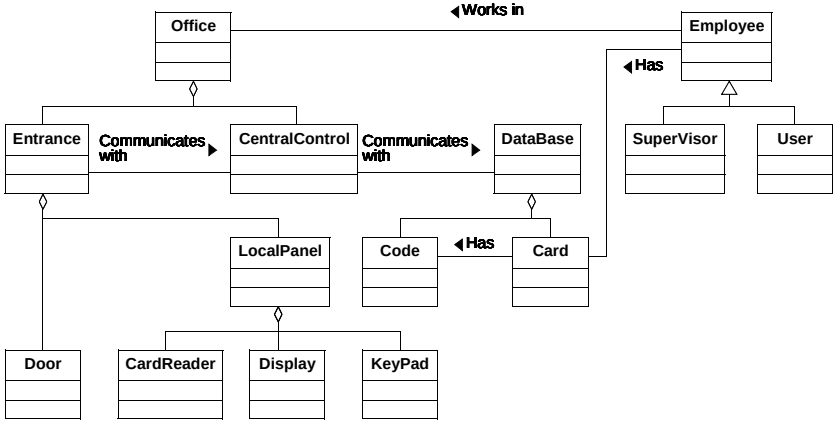


図3: オブジェクト モデルの要求条件

要求条件オブジェクト モデルから分析オブジェクト モデルを作成する際は、オブジェクト モデルに関係する実世界での特性を考えるよりも、要求条件のシステムの特性に注目します。要求条件のアクティビティでは、クラスがどのようなものになるか、またはモデル化する必要があるかどうかさえもわかりません。要求条件や作成するシステムを分析する段階で、クラスをソフトウェア エンティティやハードウェア エンティティに割り当てるか、または、割り当てないかを決定します。

Access Controlシステムのシステム分析

システム分析は、要求条件や問題領域など、高いレベルでの分析の結果をベースにします。システム分析モデルを作成する際は、システムの内部構造により重点を置きます。ただし、できるだけ設計上の決定は行わないようにします。

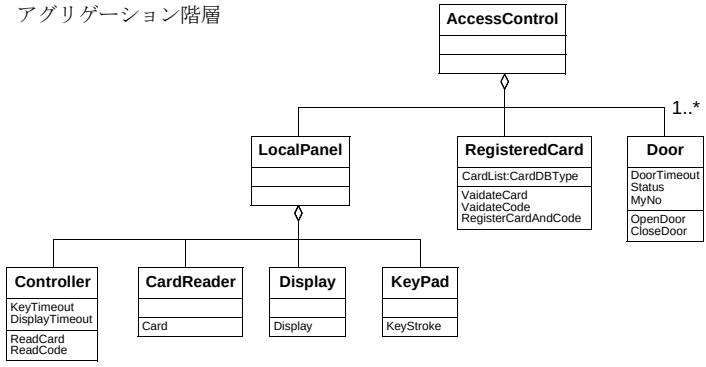
分析オブジェクト モデル: 基本バージョン

分析オブジェクト モデルの基本バージョンでは、モデル化する情報の構造が簡単なので、*継承*の概念は使用しません。図4を参照すると、クラス間や属性および操作間に*アグリゲーション*や*関連*のリレーションを見つけることができます。

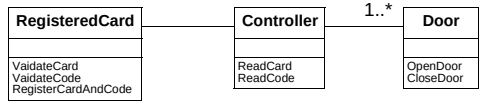
図4の分析オブジェクト モデルは、要求条件オブジェクト モデルから以下のような点が変更されています。

- わかりやすくするために、アクタをオブジェクト モデルから削除。
- SDL設計への割り当てを容易にするために、クラスを構造化。特に、アグリゲーションの構造はSDL設計への割り当てを意識して設計されています。基本的に、このような構造はSDLの設計でも維持されます。
- 問題の理解のみのために導入された要求条件オブジェクト モデルの冗長なクラスを削除。
- *アクティブ*と*パッシブ*オブジェクトを区別。アクティブ オブジェクトが振る舞いを表現するのに対して、パッシブ オブジェクトはデータ構造とデータ操作のみを表現します。アグリゲーション階層のクラスはすべてアクティブ クラスになります。また、情報構造のクラスはパッシブ オブジェクトになります。
- 属性と操作をクラスに追加。
- 分析オブジェクト モデルを、3つの部分で構成。クラス間のリレーションは、アグリゲーションの階層、通信構造、および情報構造の観点で分類しています。

アグリゲーション階層



通信構造



情報構造

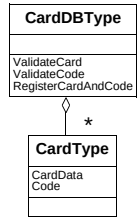


図4: Access Controlシステムの分析オブジェクトモデル（基本バージョン）

分析使用パターン モデル

以下のMSCは、ドアを開くための使用パターンを記述したものです。このMSCは、分析使用パターン モデルの一部分を表現しています。詳細さのレベルは、MSCインスタンスによって各末端のオブジェクトを表現する非常に詳細なレベルと、アグリゲーション構造の各サブシステムをインスタンスとして記述する全般的なレベルのいずれかで記述することができます。読みやすさと表現のしやすさのどちらを重視するかは、アプリケーションの分野や設計方法を考慮して決定します。この例では、サブシステムによる表現が選択されています（図5参照）。

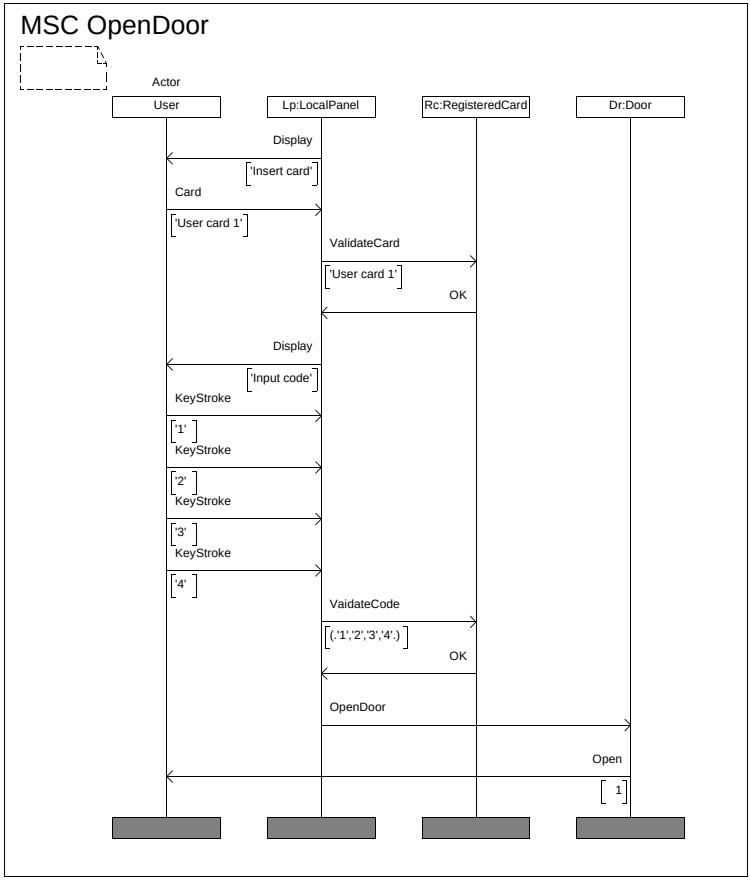


図5: OpenDoor分析使用パターン

MSCインスタンスは、実際のインスタンスであることに注意してください。つまり、MSCインスタンスは*オブジェクト*を表します。このため、インスタンス名には、オブジェクト名と対応するクラス名の両方を含めています。

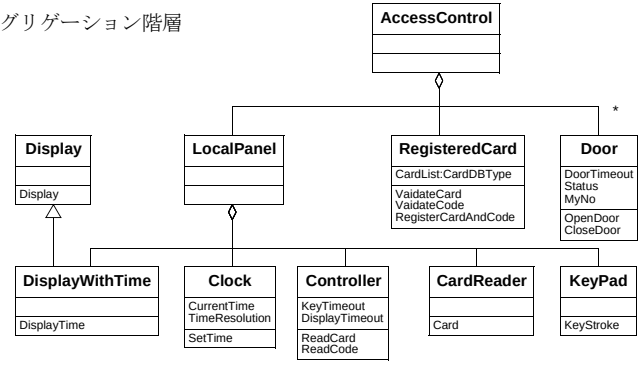
また、すべてのMSCメッセージが厳密にクラスの操作に割り当てられるわけではないことにも注意してください。場合によっては、操作が同期処理になることもあります。つまりリターン メッセージを必要とする場合もあります。このリターン メッセージは、図5のMSC使用パターンでも記述されています（例: *ValidateCard*という操作は、*ValidateCard*と*OK*というメッセージで表現されています)。

分析オブジェクトモデル: 拡張バージョン

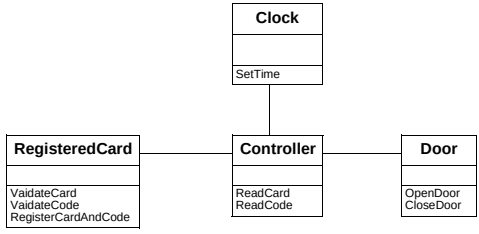
次に、時間処理の機能が追加された新しい要求条件の分析方法を説明します。現在の時刻を表示する時計機能の追加によって、主にDisplayクラスに新しい操作が追加されます。また、新しいクラスであるClockを導入します。このクラスはクロックを処理し、現在の時刻を更新します。図6に、Access Controlシステムの拡張された分析オブジェクト モデルを示します。

なお、実際に振る舞いを追加する際には、システムの内部について記述したテキスト要求条件や、要求条件分析とシステム分析の使用パターン モデルなどの他のモデルも拡張しなければなりません。

アグリゲーション階層



通信構造



情報構造

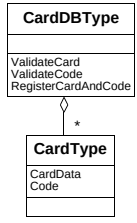


図6: Access Control システムの分析オブジェクト モデル
（時間処理と追加した拡張バージョン）

Access Controlシステムのオブジェクト指向設計

システム設計

システム設計アクティビティの目的は、設計アーキテクチャを作成することと、使用パターンを詳細設計に役立つレベルにまで詳細化することです。また使用パターンの詳細化には、SDLエクスプローラの*Verify MSC*機能を使って、設計検証を容易にする目的もあります。

本書では、設計で利用できるオブジェクト指向SDLの機能について説明することを主題としているため、システム設計に必要な手順のすべてを網羅してるわけではありません。

オブジェクト設計

ここでは、SDLの新しい概念について順を追って説明します。

- Access Controlシステムのバージョン1では、ブロックとプロセスのみに新しいタイプの概念を使用します。
- バージョン2では、リモート プロシージャ、返値プロシージャ、グローバル プロシージャなど、新しいプロシージャの概念を使います。また、パッケージ、特殊化、仮想の概念を導入します。

バージョン1: ブロック タイプとプロセス タイプ

バージョン1のAccess Controlシステムでは、分析オブジェクト モデルに基づいて、アグリゲーション階層の最上位クラスをSDLシステムに割り当てます。アグリゲーション階層の末端ノードはプロセスに割り当てます。また、最上位と末端ノード間の各クラスは、SDLブロックに割り当てます。なお、アグリゲーション構造の最上位クラスと末端クラスの間にクラスが存在しない場合でも、SDLプロセスはSDLブロック内に定義する必要があります。

バージョン1では、分析オブジェクト モデルで識別された、6つのプロセス（CardReader、Controller、Display、Door、KeyPad、RegisteredCard）を記述します。また、分析オブジェクト モデルのアグリゲーション構造を維持するために、LocalPanel、Doors、RegisteredCardの3つのブロックを作成します。図7を参照してください。

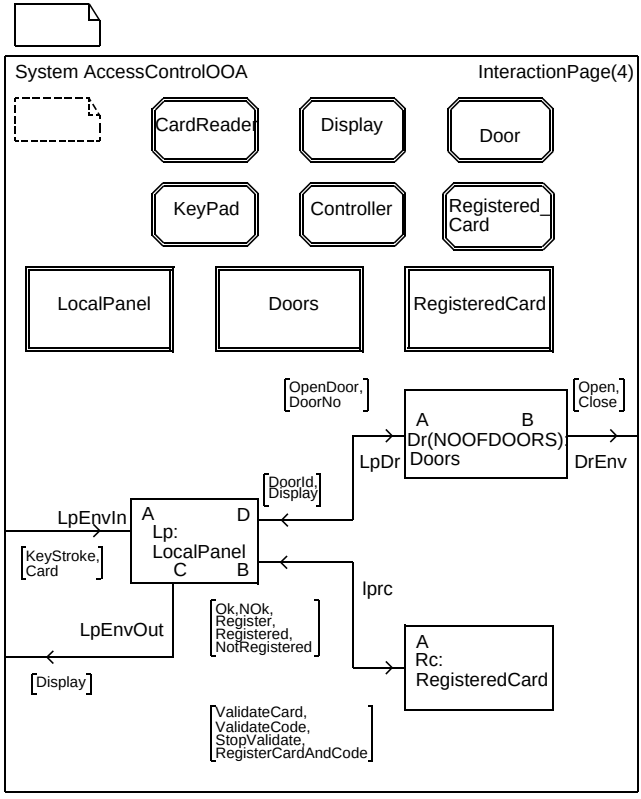


図7: AccessControlOOA システム ダイアグラム

ブロック タイプとプロセス タイプ

タイプの定義は、システムのあらゆる場所に配置できます。バージョン1では、最大の可視性を確保するためにシステム レベルに配置します。ただし通常は、個々のサブシステム（ブロック タイプ）を並行して編集、分析するために、別々のパッケージにタイプの定義を配置します。

システム内やパッケージ内など、上位の階層にタイプの定義を配置するとタイプの定義を参照できるすべての位置でインスタンス化することが可能になります。また、システム内のあらゆる場所で特殊化できます。

Block Type LocalPanel

各タイプが同じレベルに配置されている場合も、分析オブジェクト モデルに基づいてタイプをインスタンス化することで、構造は保持されます。したがって、CardReader、Display、KeyPad、Controllerプロセス タイプのインスタンスは、LocalPanelブロック タイプ内に作成します。図8を参照してください。

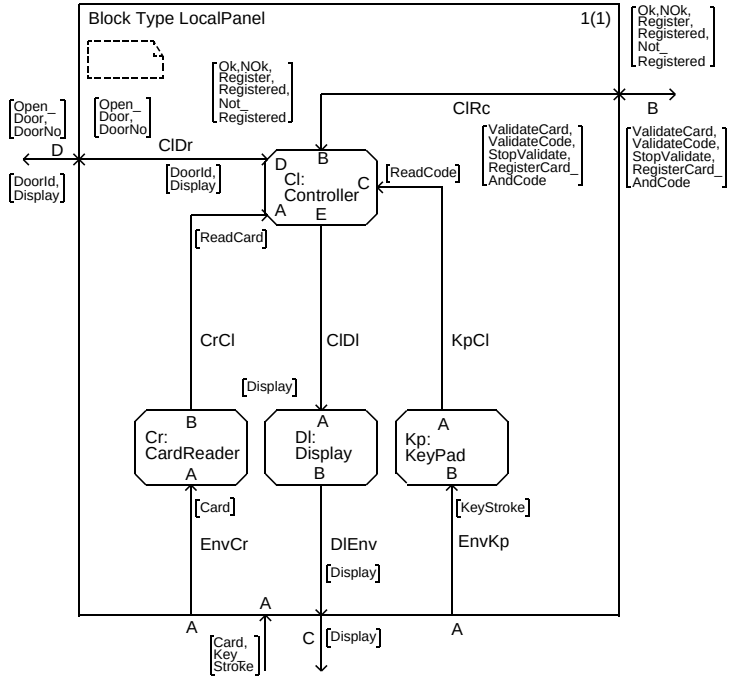


図8: LocalPanelブロック タイプ

Block Type RegisteredCard

RegisteredCardブロック タイプには、RegisteredCardプロセス タイプのインスタンスのみが存在します。SDLではブロック タイプとプロセス タイプに同じ名前を使っても、これらは異なるエンティティ クラスに属しているので問題になりません。図9を参照してください

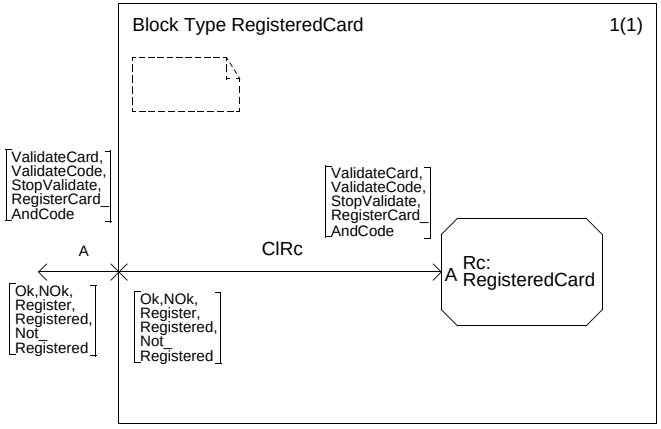


図9: RegisteredCardブロック タイプ

CardDBTypeクラスとCardTypeクラス

先に説明したように、クラスにはデータとデータ操作が主要な内容であるクラスが存在します。CardDBTypeやCardTypeなどのクラスはこのタイプに属し、設計時に抽象データ型として実装します。CardDbTypeデータ タイプには、カードとコードの照会や、新しいカードとコードの登録用に定義された演算子があります。これらの演算子は、C言語の関数としてインライン実装されます。図10を参照してください。

```
NEWTYPE CardType
STRUCT
  CardData Charstring;
  Code     CodeArray;
ENDNEWTYPE CardType;
NEWTYPE CardDbType
 array(Index,CardType)
ADDING
LITERALS
 NewDb;
OPERATORS
 ValidateCard:Charstring,CardDbType->ValCardResType;
 ValidateCode:CardType,CardDbType->ValCodeResType;
 ListFull:CardDbType->Boolean;
 RegisterCardAndCode:CardType,CardDbType->CardDbType;
/*#ADT(B)
#BODY
#ifndef XNOPROTO
extern #(CardDbType) #(NewDb) (void)
#else
extern #(CardDbType) #(NewDb) ()
#endif
{
 return(yMake_#(CardDbType)(yMake_#(CardType)("V¥0",
 yMake_#(CodeArray)('0'))));
}
......
.......
*/
ENDNEWTYPE;
```

図10: CardTypeとCardDbTypeデータ タイプ

CardDbTypeタイプのインスタンスは、RegisteredCardプロセス タイプに宣言します。また、RegisteredCardクラスのValidateCard、ValidateCodeおよびRegisterCardAndCodeの各操作は、CardDbTypeデータ タイプの演算子として実装します。図11を参照してください。

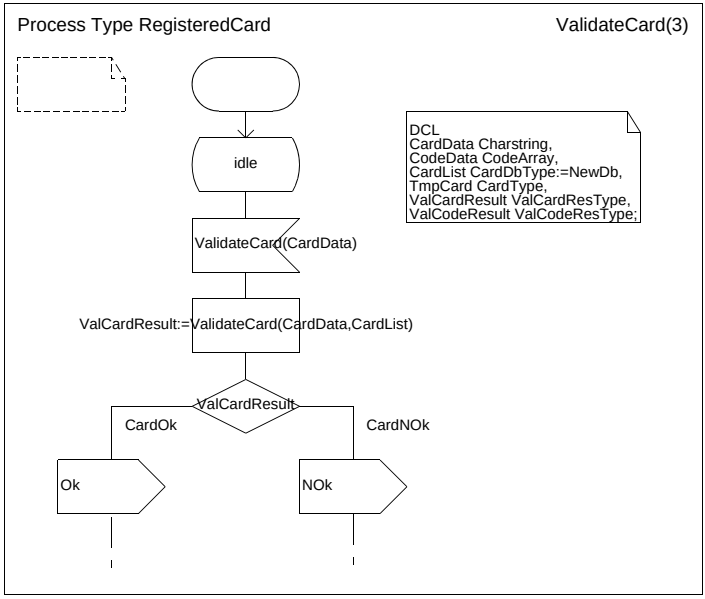


図11: RegisteredCardプロセス内での演算子の呼び出し

Doorsブロック タイプとDoorプロセス タイプ

Access Controlシステムの要求条件には、最大4か所のドアを制御することが記述されています。オブジェクト指向SDLの設計では、この要求条件をDoorsブロック タイプのブロック セットとして作成します。NOOFDOORSシノニムの既定値は1ですが、1から4までの任意の値を割り当てることができます。Doorsブロック タイプは、Doorプロセス タイプのインスタンスで構成します。オブジェクト指向分析のとおり、Doorプロセス タイプは、ドアが開いている時間（DoorTimeOut属性）とドアの開閉（OpenDoorとCloseDoor操作）を制御します。

バージョン2: プロシージャ、特殊化およびパッケージ

SDLプロセスの設計では、遷移の経路をできるかぎり短くすることが必要です。多くの場合、プロシージャを使用すると遷移を短くすることができます。

バージョン1でのプロシージャの使用

バージョン1では、プロシージャが頻繁に使用されています。ここでは、RegisterCardとReadCodeの2つのプロシージャについて説明します。この2つのプロシージャは共にControllerプロセスで宣言し、呼び出します。

Procedure RegisterCard

RegisterCardプロシージャは、ユーザーのカードまたは管理者のクレジット カードなどの、新しいカードを登録する必要があるときに呼び出されます。以下に、このプロシージャの機能を示します。

- まず、ReadCodeプロシージャを呼び出してユーザーの4桁のコードを読み取ります。
- ReadCodeプロシージャから正常に戻った場合は（ReadCodeResult=Successful）、RegisteredCardプロセスに新しいカードの登録要求（RegisterCardAndCode信号）を送ります。
- Registrationの結果（Registered信号かNotRegistered信号の返送）を受信するまで待ち、戻ります（図12参照）。

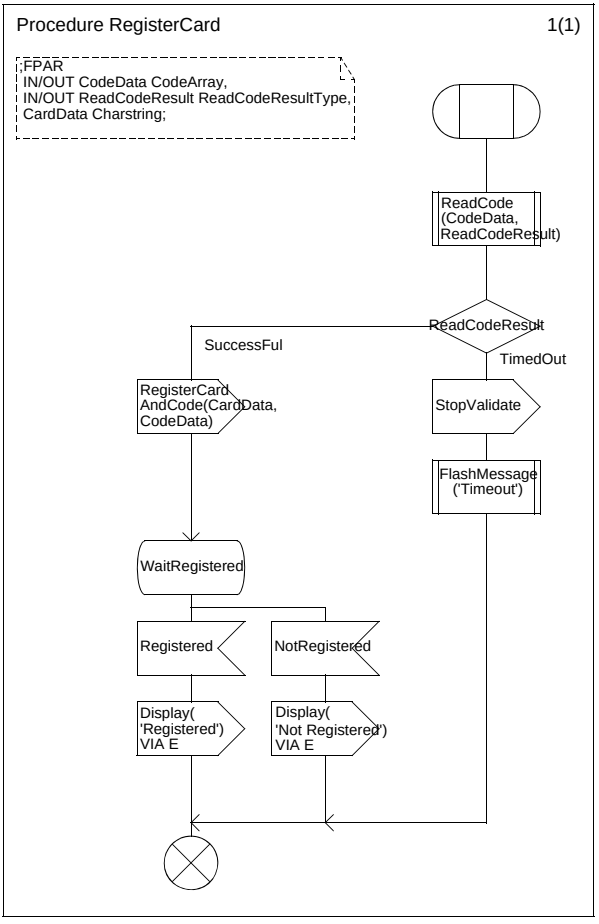


図12: RegisterCard プロシージャ

Procedure ReadCode

ReadCodeプロシージャは、キーパッドから入力された4桁のコードを読み込みます。読み込まれた値は、CodeDataという名前の配列に保存されます。4桁の数値を正常に受け取ると、ReadCodeResultにSuccessfulを割り当て、呼び出し側のプロセスまたはプロシージャに戻ります。

このプロシージャは、カードとコードの照会シーケンスにおいて、RegisterCard プロシージャとControllerプロセスの両方から呼び出されます。図13を参照してください。

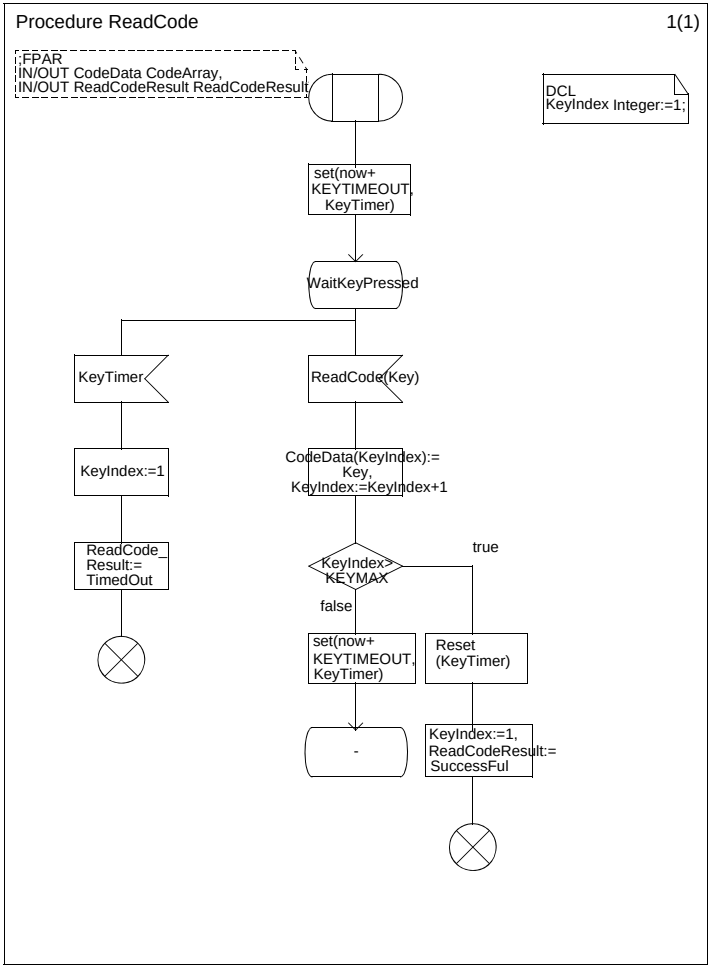


図13: プロシージャ ReadCode

リモート プロシージャと返値プロシージャ

バージョン2では、RegisterCardプロシージャをControllerプロセスからRegisterCardプロセスに移動することを検討します。これは、以下の2つの理由によるものです。

1. RegisterCardプロシージャは、カードが登録されるたびに呼び出されるので、RegisterCardプロセスに配置するのが最も自然です。
2. ControllerプロセスとRegisterCardプロセスの間で、登録手続きの開始と終了のタイミングを通知する信号をやり取りする必要がなくなります。

また、ReadCodeプロシージャも移動する必要があります。これは、RegisterCardプロシージャがReadCodeプロシージャを呼び出した場合、デッドロックが発生するためです。

> メモ：
>
> プロセスがリモート プロシージャを呼び出すと、そのプロセスはプロシージャ コールが実行されたことを通知する（暗黙の）リターン信号を待つために、新しい暗黙の状態に入ります。このとき、他のリモート プロシージャへの呼び出しなどの、すべての新しい信号が保持されます。したがって、デッドロック状態に入ってしまう可能性があります。

ReadCodeプロシージャはKeyPadプロセスに配置し、FlashMessageプロシージャ（およびRegisterCardに呼び出される他のプロシージャ）はDisplayプロセスに配置することができます。

SDL内のリモート プロシージャ

通常、プロシージャはそれを宣言するプロセスまたはプロシージャからのみ呼び出すことができますが、EXPORTEDとして宣言することで、システム内の任意のプロセスまたはプロシージャから呼び出せるようになります。リモート プロシージャの概念は、信号のやり取りとしてモデル化されます。

セーブの概念

サービス プロセス、つまりEXPORTEDプロシージャを持つプロセスは、ある特定の状態にあるときだけリモート プロシージャ コールを行うことができます。プロセスがリモート プロシージャ コールによって中断されてはならない場合、セーブ シンボルを使用してそのリモート プロシージャ コールを保持し、後でそれを処理することができます。これは、リモート プロシージャ コールがすぐに処理されるかどうかがわからないことを意味します。また、この呼び出しプロセスのモデルでは、各リモート プロシージャ コールによって、新しい暗黙の状態が導入されます。プロセスは、リモート プロシージャ コールの処理が終了されるまで暗黙の状態にとどまります。

EXPORTEDプロシージャの宣言方法

1. プロシージャのヘッダで、プロシージャをEXPORTEDとして宣言します。
2. リモート プロシージャを呼び出す各プロセスまたはプロシージャで、インポート プロシージャを指定します。
3. リモート プロシージャを定義して、エクスポートおよびインポートされるプロシージャの名前とシグネチャ（FPAR）を記述します。この定義は、システム ダイアグラム、ブロック ダイアグラム、パッケージ内に配置できます。この宣言を特定のスコープ内に置くことによって、リモート プロシージャの可視性が決定されます。

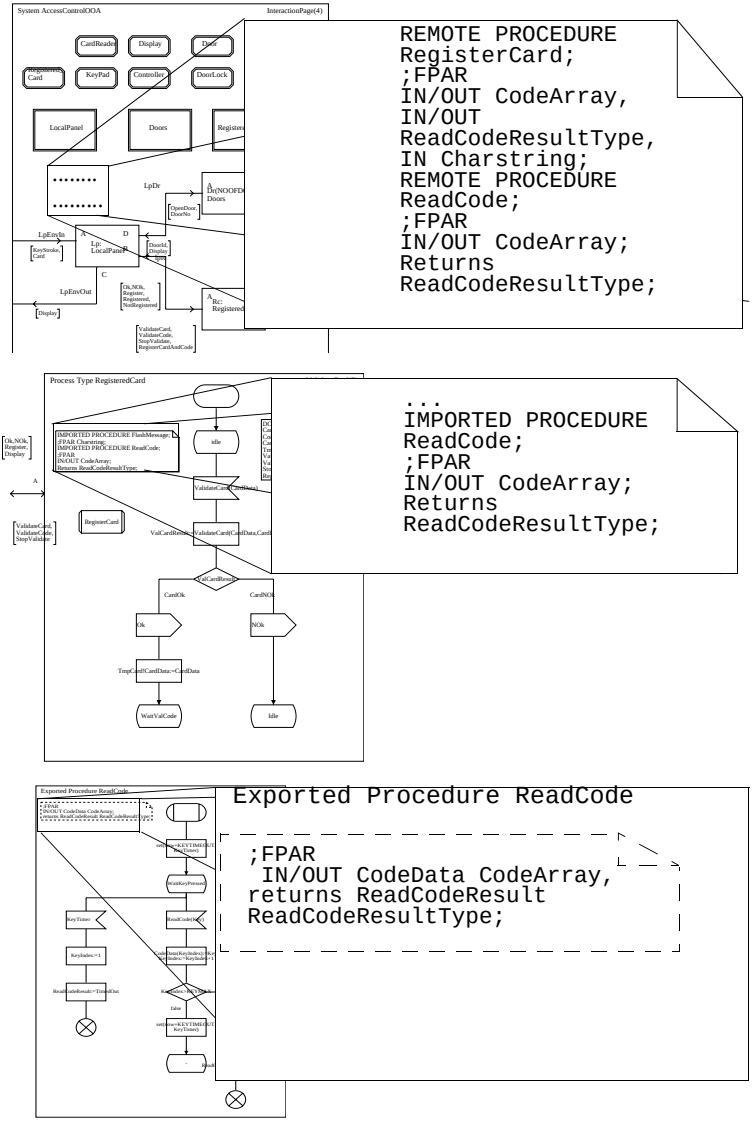


図14: リモートプロシージャの宣言

SDLの返値プロシージャ

最後のパラメータにIN/OUT型が定義されているプロシージャはすべて、返値プロシージャとして呼び出すことができます。しかし、返値プロシージャとして使用するプロシージャは、返値プロシージャとして宣言することが最も望ましい方法です。返値プロシージャへの呼び出しは、代入などの方法で式に直接記述できます。図15を参照してください。

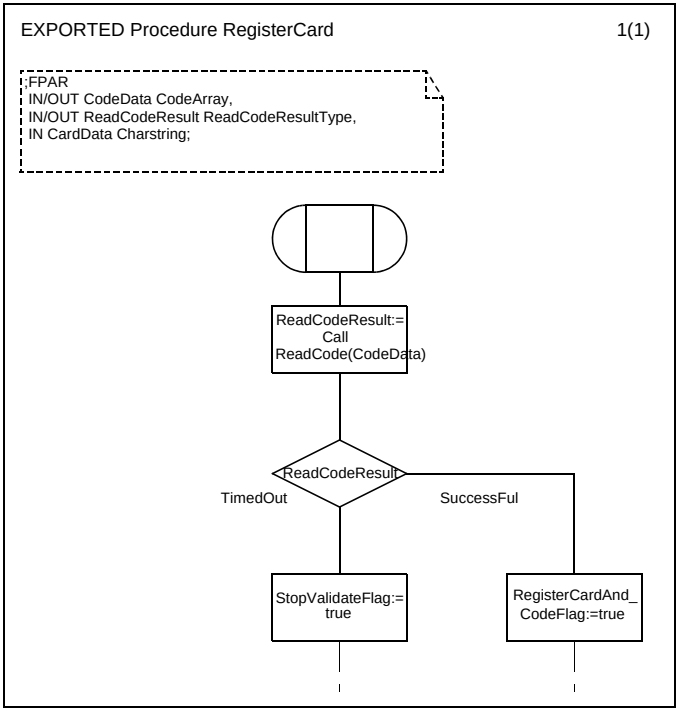


図15: 返値プロシージャの使用

バージョン2では、ReadCodeを返値プロシージャとして宣言します。このプロシージャは値としてReadCodeResultTypeを返します。ReadCodeResultTypeは、RegisterCardプロシージャからも返す必要があるので、変数の宣言はそのまま残します。図15を参照してください。

SDL内のグローバル プロシージャ

プロシージャは、SDL内でグローバルに定義することもできます。プロシージャをグローバルに定義すると、プロシージャの呼び出しの際に概念的なコピーがローカル プロセスに作成されると考えることができます。

信号送信のためのグローバル プロシージャ

メッセージを表示するほとんどの処理は、Controllerプロセスが行います。しかし、RegisteredCardプロセスのRegisterCardプロシージャとDoorプロセスもメッセージをディスプレイに送信します。

バージョン1では、ControllerプロセスがDisplay信号をDisplayプロセスに送る中間的な伝達機構として機能しました。グローバル プロシージャを宣言することによって、さまざまなメッセージをDisplay信号に使って送信できれば、より効率の良い設計が可能です。また、Displayプロセス、Doorプロセス、RegisterCardプロシージャからグローバル プロシージャを呼び出すことができれば、設計上有利です。ただし、この方法は実現できません。なぜなら、前述のとおり、グローバル プロシージャのコピーが暗黙的にローカルに作成されるためです。たとえば、Doorプロセスからグローバル プロシージャを呼び出すと、実際は呼び出し側のDoorプロセスからDisplay信号を送ることになります。したがって、信号の送信によって混乱すること以外には、特に得られるものはなく、信号は依然として出力チャンネルなどで宣言する必要があります。これに代わる方法としては、プロシージャをEXPORTEDプロシージャとして宣言し、リモート プロシージャとして呼び出す方法があります。しかし、リモート プロシージャの概念はあまり頻繁に使用すべきではなく、信号を隠すだけの目的で使用するのは適切ではありません。

各種のプロシージャの使い分け

ローカル プロシージャ

- 信号のやり取りをわかりやすくするために、遷移の経路を短くする目的に使用できます。
- ローカル ルーチンを記述するために使います。

リモート プロシージャと返値プロシージャ

- ローカル ルーチンをグローバルにアクセスできるようにします。
- 返値プロシージャを使用して式を単純化することができます。
- 信号の代わりにリモート プロシージャ コールを使用して、データへのアクセスと操作を行うことができます。

グローバル プロシージャ

- マクロの代わりに使うことができます。
- プロシージャの明確な所有者が存在しない場合、リモート プロシージャの代わりに使うことができます。

特殊化: プロパティの追加/再定義

オブジェクト指向言語の利点の1つは、オブジェクトを簡単かつ視覚的な方法で作成できることです。オブジェクトを作成する際は、既存のオブジェクトに新しい機能を追加したり、既存のオブジェクトの機能を再定義して新しいオブジェクトを作成することができます。この手法は、一般的に*特殊化*と呼ばれています。

SDLでは、以下の2つの方法でタイプを特殊化できます。

- サブタイプには、スーパータイプで定義されていない機能を追加できます。たとえば、プロセス タイプ[1] に新しい遷移を追加したり、ブロック タイプに新しいプロセスを追加することができます。
- サブタイプには、スーパータイプで定義されている仮想タイプと仮想遷移を再定義できます。たとえば、あるプロセス タイプ内で遷移の内容を再定義したり、ブロック タイプの内容や構造を再定義できます。

このような、機能を付加するメカニズムによって、振る舞いつまり遷移を追加することができます。たとえば、図16に示すTimeDisplayプロセス タイプは、新しい遷移を追加したDisplayのサブタイプです。INHERITSというキーワードによって、DisplayTimeタイプはDisplayのサブタイプとして定義されます。また、Displayプロセス タイプのすべての定義がDisplayTimeに継承されることを示しています。

ゲートAおよびBは点線で表示され、Displayプロセス タイプ内のゲート定義に対する参照であることを表します。また、DisplayTime信号が追加されていることを示しています。

1. SDLは、振る舞いの仕様を特殊化できるという点で、他の多くのオブジェクト指向言語と異なります。他のほとんどの言語では、新しい機能を追加する場合、サブクラス内で仮想メソッドを再定義する必要がありますが、SDLでは、プロセス タイプに新しい遷移を追加するだけで済みます。

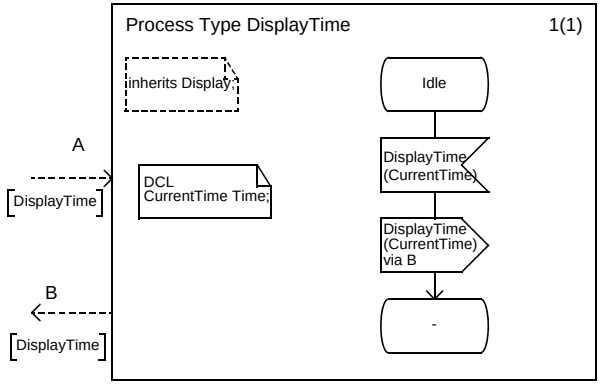


図16: TimeDisplay プロセス タイプ

場合によっては、プロパティを追加するだけでなく、スーパータイプの機能を再定義しなければならないこともあります。図16のDoorプロセス タイプは、新しいDoorOpenerプロセスにOpenDoor信号とCloseDoor信号を送信するように再定義しなければなりません。したがって、図17のように、Doorプロセス タイプの対応する遷移は仮想遷移として定義する必要があります。

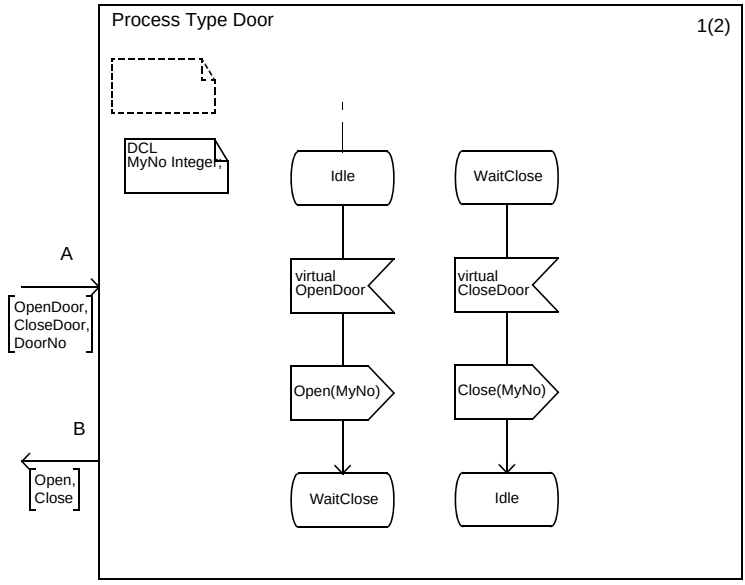


図17: 仮想遷移が定義されたDoorプロセス タイプ

新しいSpecialDoorブロック タイプの定義に対応して、Doorプロセス タイプ内の遷移は図18のように再定義されます。

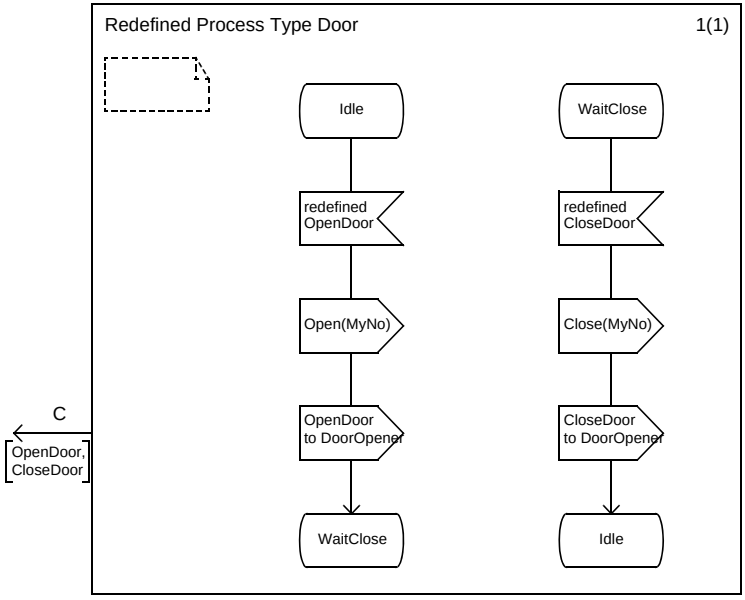


図18: 再定義されたDoorプロセス タイプ

仮想遷移に加えて、スタート遷移、セーブ、コンティニュアス信号、即時遷移、優先入力、リモート プロシージャ入力、リモート プロシージャ セーブは、仮想として指定できます。これらの概念はすべて仮想を定義すると、遷移の開始方法や、遷移を開始またはセーブするかどうかについて定義できるようになります。さらに、仮想セーブは、入力遷移として再定義することができ、逆に入力遷移から仮想セーブに再定義することもできます。

例: Access Controlシステムへの時計の追加

ここまでの説明で使用したAccess Controlシステムに、機能を拡張して現在の時刻を表示する時計機能を追加します。時刻は、ディスプレイに「HH:MM」の形式で表示します。またパネルから「#」を入力し、次に「HHMM」の書式で入力することで時刻を設定できるようにします。

以前に説明したオブジェクト指向分析などの方法で問題領域を分析すれば、ローカル パネルに時計を追加して時計の機能を実現することが最も簡単であることがわります。クロックから、1分ごとに現在の時刻をコントローラに送信し、コントローラによってディスプレイに時刻を表示させます。さらに、コントローラの機能を拡張して、キーパッドから時刻を設定できるようにします。

問題領域にSDLの特殊化の概念を適用するには、元のAccess Controlシステムの仕様を若干変更する必要があります。時計機能を持つ新しいAccess Controlシステムは、元のAccess Controlシステムの仕様に対して特殊化されたものと見なすことができます。つまり、新しいAccess Controlシステムを定義する際は、元のAccess Controlシステムの機能を継承することができるはずです。したがって、元のAccess Controlシステムを、図19のようにBaseAccessControlという名前のシステム タイプに定義します。

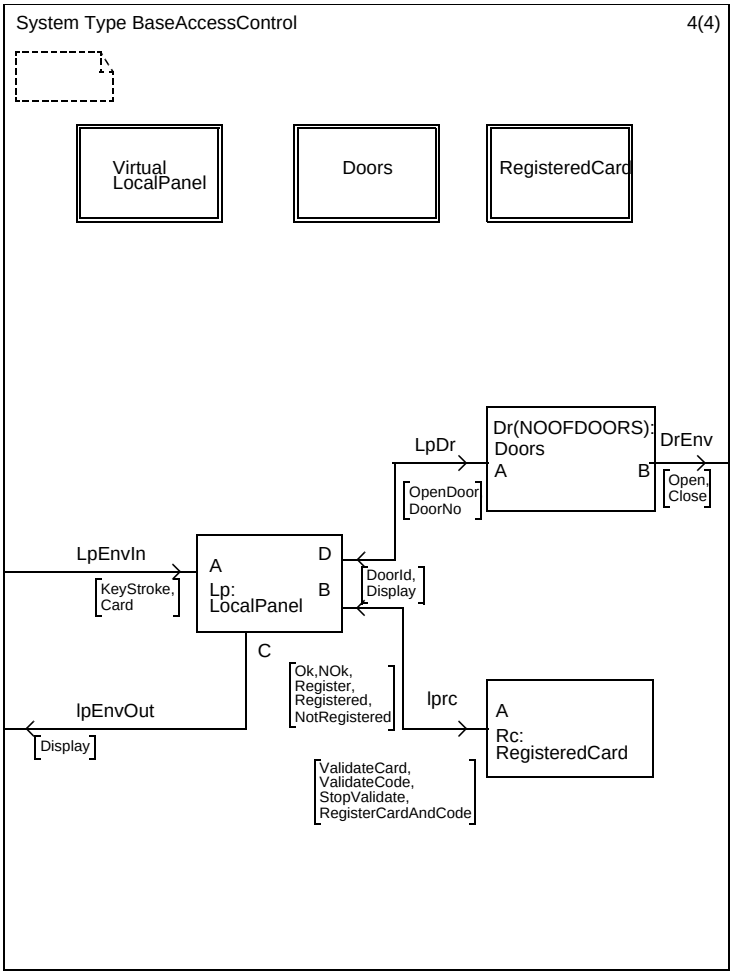


図19: システム タイプBaseAccessControl

BaseAccessControlの特殊化では、LocalPanelブロック タイプを変更するため、LocalPanelブロック タイプを仮想として定義します。同様に、LocalPanelブロック タイプで使用するCardReader、Display、KeyPad、Controllerプロセス タイプもすべて仮想として定義します。また、以前はシステム レベルに置かれていたプロセス タイプの定義は、明確化のためにそれらを使用するブロック タイプ内に配置します。

以上の修正によって、時計機能を持つAccess Controlシステム（TimeAccessControl）を定義する際に、BaseAccessControlシステム タイプからの継承が可能になります。図20を参照してください。

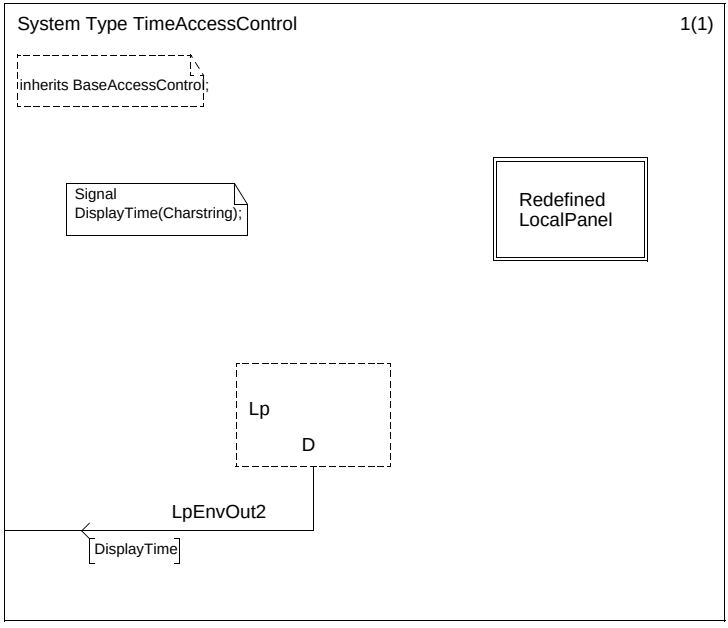


図20: TimeAccessControl システム タイプ

TimeAccessControlシステム タイプでは、Lpブロック（LocalPanelタイプ）から外部環境に送信されるDisplayTime信号を新たに追加し、BaseAccessControlを継承します。さらに図21に示す、LocalPanelブロック タイプをTimeAccessControl内に再定義します。

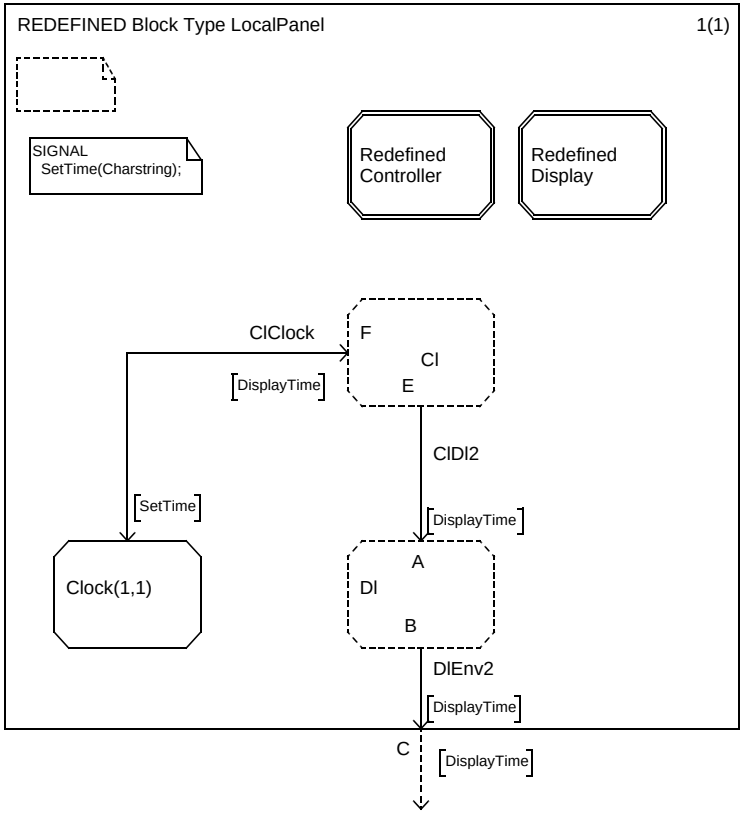


図21: 再定義されたブロック タイプLocalPanel

LocalPanelは、Clockプロセスを含むように再定義します。Clockプロセスは、DisplayTime信号をCl（Controllerタイプ）に送り、ClからSetTime信号を受け取ります。さらに、Clを拡張して、ClがDisplayTime信号をDl（Displayタイプ）に送り、DlからゲートCを介して環境にDisplayTime信号を送るようにします。

Displayの再定義は簡単です。図22に示されているように、DisplayTime信号に関する新しい遷移を追加します。

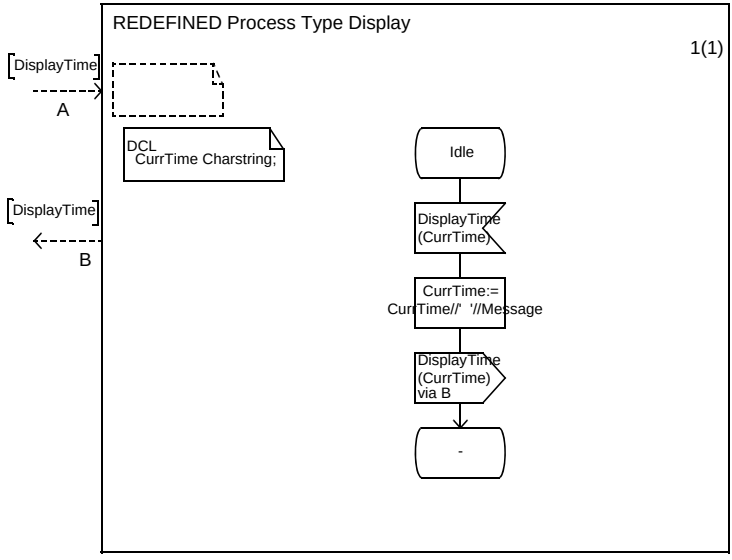


図22: 再定義されたDisplayプロセス タイプ

Controller（36ページの図23）の再定義では2つの機能を追加する必要があります。追加する機能は、DisplayTime信号の処理と、KeyPadから入力される新しい時刻を読み取って時計に設定する処理です。

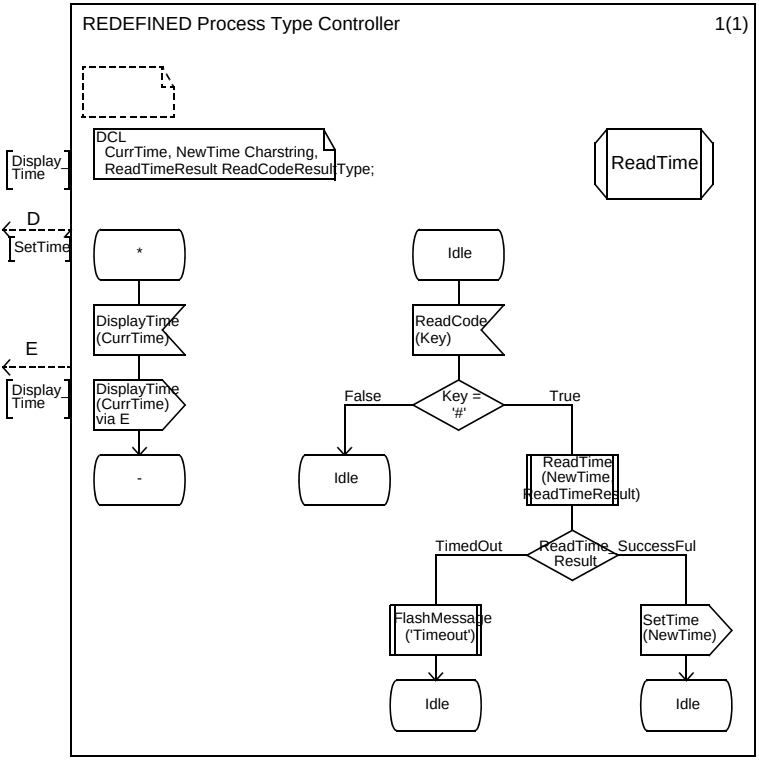


図23: Controller プロセス タイプの再定義

DisplayTime信号を処理するために、DisplayTimeを受信したときにゲートE経由でDlプロセスにDisplayTime信号を送信する、すべての状態に遷移を追加します。さらに時刻を設定できるように、ReadCode信号を処理する遷移をIdle状態に追加します。キーパッドで押されたキーが#の場合は、ReadTimeプロシージャで新しい時刻が読み取られます。新しい時刻を正常に読み取ることができた場合は、SetTime信号をClockプロセスに送ります。

最後にClockプロセスを図24のように定義します。

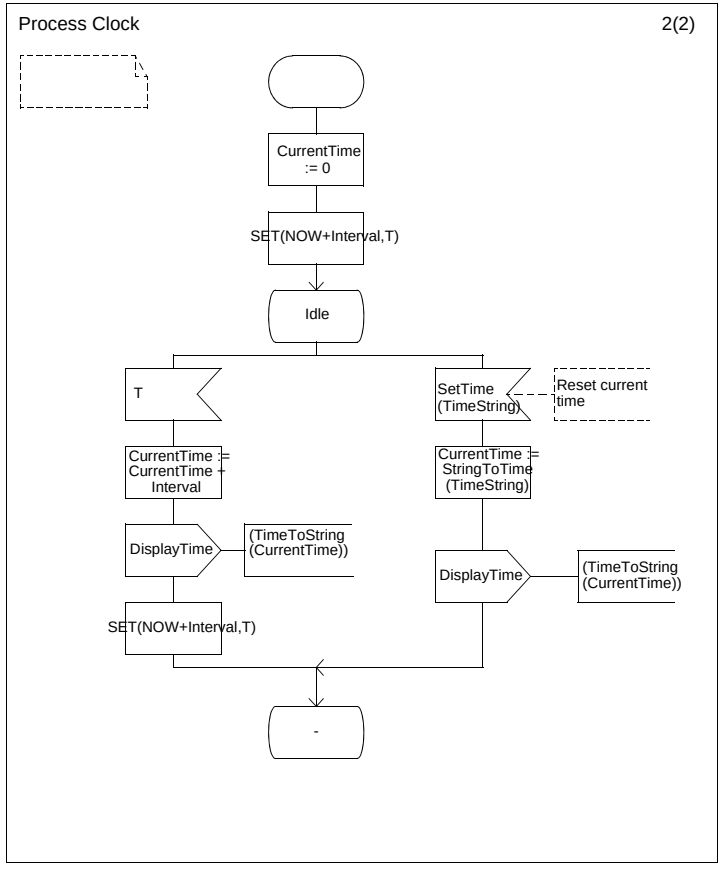


図24: Clockプロセス

時刻型のCurrentTime変数は、新しい時刻を分単位で格納します。毎分（時間間隔60）タイマの期限が切れると共に、CurrentTime変数が1増加し、DisplayTime信号がClプロセスに送られます。CurrentTime変数は、SetTime信号を受信すると新しい値に更新されます。Clockプロセスの外部で扱う時刻の値、つまりDisplayTimeとSetTime信号のパラメータは、文字列型なので、時刻型を文字列型に変換する関数とその逆の変換を行う関数が必要です。これらの関数は、以下のように定義できます。

```
NEWTYPE TimeOperators
  LITERALS Dummy;
  OPERATORS
    TimeToString : Time -> Charstring;
      /* 時刻型を文字列型に変換します。結果は
         'HH:MM' の形式になります */
      /*#OP (B) */

    StringToTime : Charstring -> Time;
      /* 文字列型を時刻型に変換します。ここで文字列は
         'HHMM' の形式になっていると仮定します */
      /*#OP (B) */
/*#ADT (B)
#BODY

SDL_Charstring #(TimeToString)(T)
SDL_Time T;
{
  SDL_Charstring result:=NULL;
  int Hours, Minutes;
  char tmp1[4], tmp2[4];

  Hours = (T.s/60/60)%24;
  Minutes = (T.s/60)%60;
  tmp1[0]='V';
  tmp2[0]='V';
  sprintf(&(tmp1[1]),"%2ld",Hours);
  sprintf(&(tmp2[1]), "%2ld",Minutes);
  xAss_SDL_Charstring(&result,tmp1,XASS);
xAss_SDL_Charstring(&result,xConcat_SDL_Charstring(r
esult,xMkString_SDL_Charstring(':')));
xAss_SDL_Charstring(&result,xConcat_SDL_Charstring(r
esult,tmp2));
  result[0]='V';
  if(Hours<10)
    result[1]='0';
  if(Minutes<10)
    result[4]='0';
  return result;
}

SDL_Time #(StringToTime)(C)
SDL_Charstring C;
{
  SDL_Time T;
  SDL_Charstring tmpstr;
  tmpstr=xSubString_SDL_Charstring(C,1,2);
  T.s = atoi(++tmpstr)*60*60;
  tmpstr=xSubString_SDL_Charstring(C,3,2);
  T.s = T.s + atoi(++tmpstr)*60;
  return T;
}
*/
ENDNEWTYPE;
```

パッケージ

パッケージの概念によって、複数のタイプをまとめることができるようになります。以下に、パッケージに含めることができる各種の定義を示します。

- システム タイプ、ブロック タイプ、プロセス タイプ、サービス タイプ、プロシージャなどのダイアグラム タイプ
- 抽象データ タイプとシノニム
- 信号と信号リスト

パッケージに定義された内容は、USE句を使ってシステムまたは別のパッケージで使用できるようにします。

SDLアナライザは、パッケージの意味解析をサポートします。これによって、大きなシステムを複数のパッケージに分割することが可能になり、プロジェクトの定義が容易になります。

また、パッケージに定義されたタイプは、必ず分析することができる点もパッケージの重要な特長の1つです。これは、あるプロセス タイプがパッケージ内に定義されている場合、そのプロセス タイプが使用するデータ タイプや信号も同じパッケージから必ず参照できることを意味します。図25に、パッケージの使用例を示します。

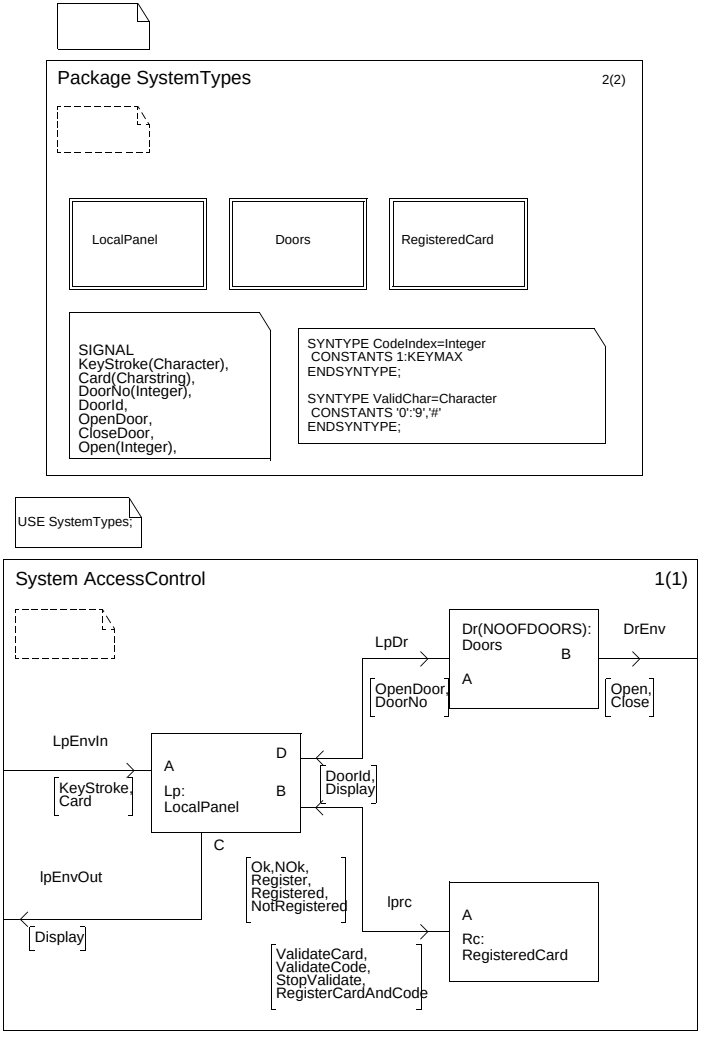


図25: Access Control システムでのパッケージの使用

第 2 章 データ型

この章では、SDL Suiteでの各データ型がどのように処理されるかについて説明します。SDLでサポートされているすべてのデータ型の概要を、例やガイドラインを挙げて説明します。また、C/C++言語やASN.1をSDL Suiteで使用する方法について解説します。

はじめに

システム内のデータの処理方法を決定することは、設計や実装作業の重要な要素であり、難しい作業の1つです。

SDL Suiteでは、以下のようなデータを利用できます。

- SDL固有データ型の使用。
- C/C++言語のデータ型や関数にアクセス。
- ASN.1データ型の使用。

この章では、SDTで利用可能な全てのデータ型について概要説明し、いろいろなデータ型の使用方法に関するガイドラインや例を示します。

SDLデータ型の使用

ここでは、SDLで利用できるデータ型の概要について説明します。SDLには、さまざまな定義済みのデータ型があります。これらの定義済みのデータ型をベースにすれば、ユーザー固有のデータ型を定義することもできます。データ型は、SDLでは「ソート」と呼び、`newtype`、`endnewtype`キーワードを使用して定義します。

例1: Newtypeの定義

```
newtype example1 struct
  a integer;
  b character;
endnewtype;
```

newtypeの定義を使うと、他のデータと互換性のない独自のデータ型を新たに定義できます。したがって、上記の`example1`とまったく同じ内容の`otherexample`をnewtypeで定義した場合は、`example1`の値を`otherexample`の変数に代入することはできません。

また、データ型には*シンタイプ*を定義することもできます。シンタイプは、あるデータ型をベースにして定義するため、ベースのデータ型との互換性を持ちますが、使用できる値の範囲に制限があります。シンタイプの定義には、`syntype`、`endsyntype`キーワードを使います。

例2: シンタイプの定義 ――――――――――――――――――――――――――――――――――――

```
syntype example2 = integer
  constants 0:10
endsyntype;
```

――

example2のシンタイプは整数型ですが、このデータ型で定義された変数は指定された0～10の範囲の値しか保持できません。上記のconstantsの行は、*範囲条件*と呼ばれます。範囲のチェックは、SDLシステムの解釈処理の際に実行されます。シンタイプの定義に範囲条件がなければ、単に同じソートに新しい名前が定義されたデータ型になります。

SDLで定義されているすべてのソートやシンタイプには、常に以下の演算子を使用することができます。

- :=（代入）
- =（等価のテスト）
- /=（非等価のテスト）

上記の演算子は、以降の項で説明する使用可能な演算子の中には入っていません。SDLの演算子は、以下のように、代数の1種として定義されます。

```
"+" : Integer, Integer -> Integer;
num : Character -> Integer;
```

+記号を囲むダブル クォーテーション マークは、中置演算子であることを示します。上記の+は、2つの整数パラメータを取り、整数値を返します。2番目の演算子numは、パラメータとして文字を1つ取り、整数値を返す前置演算子です。たとえば上記の演算子は、taskステートメントの式で次のように記述することができます。

```
task i := i+1;
task n := num('X');
```

ここで、iとnは整数の変数を想定しています。また、中置演算子は、以下の記述によって前置演算子として記述することもできます。

```
task i := "+"(i, 1);
```

この式は、i:= i+1と等価です。

定義済みソート

SDLの定義済みソートは、SDL Z100勧告の付録で定義されています。また、SDL使ったASN.1の使用方法を記述している、Z105勧告によって新しい定義済みソートが追加されています。ただし、SDLシステムをZ.100に準拠させなければならない場合は、Z105勧告のデータ型を使用すべきではありません。さらにSDL Suiteでは、いくつかのデータ型に対してIBM Rational固有の演算子がサポートされます。これらの演算子も、SDLシステムをZ.100に準拠させる場合は使用すべきではありません。ここでは、すべての定義済みソートについて説明します。特に記述がないソートはすべて、Z.100勧告で定義されているものです。

Bit

定義済みの`Bit`は、2つの値、すなわち`0`と`1`のみを取ります。Bitは、Z.105勧告で定義されていますが、Z.100勧告には含まれていません。以下に、Bitで定義した値に使用できる演算子を示します。

```
"not" : Bit -> Bit
"and" : Bit, Bit -> Bit
"or"  : Bit, Bit -> Bit
"xor" : Bit, Bit -> Bit
"=>"  : Bit, Bit -> Bit
```

これらの演算子は、以下のように定義されています。

- `not` :
  ビットを反転します。0は1になり、1は0になります。したがって、`not 0`は`1`を表し、`not 1`は`0`を表します。
- `and` :
  2つのパラメータが1の場合は1になり、それ以外の場合は0になります。たとえば`0 and 0`は`0`で、`0 and 1`は`0`、`1 and 1`は`1`となります。
- `or` :
  2つのパラメータが0の場合は0になり、それ以外の場合は1になります。たとえば`0 or 0`は`0`で、`0 or 1`は`1`、`1 or 1`は`1`となります。
- `xor` :
  2つのパラメータが異なる場合は1になり、それ以外の場合は0になります。たとえば`0 xor 0`は`0`で、`0 xor 1`は`1`、`1 xor 1`は`0`となります。
- `=>`(含蓄):
  最初のパラメータが1で、第2のパラメータが0の場合は0になり、それ以外の場合は1になります。たとえば`0 => 0`は`1`で、`1 => 0`は`0`、`0 => 1`は`1`、`1 => 1`は`1`となります。

Bitデータ型の機能は、後述のBooleanデータ型とほとんど同じです。`0`を`false`、`1`を`true`で置き換えれば、2つのソートは等価です。

BitとBooleanは、オン/オフなどの2つの値を取るシステムの性質を表すために使います。表現する性質がビットに関するものである場合や、`0`と`1`のほうが`false`と`true`で表現するよりも適している場合を除いて、通常はBooleanを使います。

Bit_string

`Bit_string`定義済みソートは、文字列や複数のBitのシーケンスを表すために使います。Bit_stringは、ASN.1の`BIT STRING`データ型をサポートするためにZ.105で定義されたものです。したがって、Bit_stringはZ.100には含まれていません。Bit_stringの要素の数に制限はありません。

Bit_stringに定義されている演算子を以下に示します。

```
mkstring  : Bit                    -> Bit_string
length    : Bit_string             -> Integer
first     : Bit_string             -> Bit
last      : Bit_string             -> Bit
"//"      : Bit_string, Bit_string -> Bit_string
substring : Bit_string, Integer, Integer
                                   -> Bit_string
bitstr    : Charstring             -> Bit_string
hexstr    : Charstring             -> Bit_string
"not"     : Bit_string             -> Bit_string
"and"     : Bit_string, Bit_string -> Bit_string
"or"      : Bit_string, Bit_string -> Bit_string
"xor"     : Bit_string, Bit_string -> Bit_string
"=>"      : Bit_string, Bit_string -> Bit_string
```

これらの演算子は、以下のように定義されています。

- `mkstring`：
  Bitの値を取って、1ビット長のBit_stringに変換します。
  `mkstring (0)`は、要素として0を持ったBit_stringを返します。
- `length`：
  Bit_stringをパラメータに取り、ビット数を返します。
  `length (bitstr('0110')) = 4`

- `first`：
  Bit_stringをパラメータに取り、第1ビットの値を返します。Bit_stringのビット長が0の場合に、first演算子を呼び出すとエラーになります。
  ```
  first (bitstr ('10')) = 1
  ```

- `last`：
  Bit_stringをパラメータに取り、最終ビットの値を返します。Bit_stringのビット長が0の場合に、last演算子を呼び出すとエラーになります。
  ```
  last (bitstr ('10')) = 0
  ```

- `//` (連結):
  演算結果は、第1パラメータ内の全要素に続いて、第2パラメータの全要素が連結された値を持つBit_stringになります。
  ```
  bitstr('01')//bitstr('10') = bitstr('0110')
  ```

- `substring`：
  第1パラメータにBit_stringを取り、部分的なコピーを返します。コピーの先頭位置は、第2パラメータに与えるインデックス値で指定します（注：先頭ビットのインデックス値は0です）。コピーのビット長は第3パラメータで指定します。第1パラメータのビット長を超える要素にアクセスするとエラーになります。
  ```
  substring (bitstr('0110'), 1, 2) = bitstr('11')
  ```

- `bitstr`：
  この演算子は、IBM Rational専用であり、0と1の2つの文字のみからなる文字列を、同じ長さのBit_stringに変換します。このとき各ビット要素には、文字列に対応する値が設定されます。

- `hexstr`：
  この演算子は、IBM Rational専用であり、HEX値（0-9、A-F、a-f）を含む文字列をBit_stringに変換します。各HEX値は、4つのビット要素からなるBit_stringに変換されます。
  ```
  hexstr('a') = bitstr('1010'),
  hexstr('8f') = bitstr('10001111')
  ```

- `not`：
  演算結果は、パラメータと同じビット長のBit_stringとなり、各要素にはBitソートのnot演算子が適用されます。つまり、各要素は反転された値になります。
  ```
  not bitstr ('0110') = bitstr ('1001')
  ```

- `and :`
  演算結果は、2つのパラメータの長い方のビット長に一致するBit_stringになります。各ビット要素の出力は、入力パラメータの対応するビット要素にBitソートのand演算子が適用された演算結果になります。演算は短い方のパラメータのビット長に達するまで実行されます。残りのビットがある場合は、0に設定されます。
  ```
  bitstr('01101') and bitstr('101') = bitstr('00100')
  ```

- `or :`
  演算結果は、2つのパラメータの長い方のビット長に一致するBit_stringになります。各ビット要素の出力は、入力パラメータの対応するビット要素に、Bitソートのor演算子が適用された演算結果になります。演算は短い方のパラメータのビット長に達するまで実行されます。残りのビットがある場合は、1に設定されます。
  ```
  bitstr('0110') or bitstr('00110') = bitstr('01111')
  ```

- `xor:`
  演算結果は、2つのパラメータの長い方のビット長に一致するBit_stringになります。各ビット要素の出力は、入力パラメータの対応するビット要素に、Bitソートのxor演算子が適用された演算結果になります。演算は短い方のパラメータのビット長に達するまで実行されます。残りのビットがある場合は、1に設定されます。
  ```
  bitstr('10100') xor bitstr('1001') = bitstr('00111')
  ```

- `=>` (含蓄):
  演算結果は、2つのパラメータの長い方のビット長に一致するBit_stringになります。各ビット要素の出力は、入力パラメータの対応するビット要素に、Bitソートの=>演算子が適用された演算結果になりますす。演算は短い方のパラメータのビット長に達するまで実行されます。残りのビットがある場合は、1に設定されます。
  ```
  bitstr ('1100') => bitstr ('0101') = bitstr ('0111')
  ```

Bit_string変数にインデックス値を指定することで、Bit_stringのビット要素にアクセスすることもできます。例えば、BというBit_string変数に対して、以下のように記述できます。

```
task B(2) := B(3);
```

上記の式は、変数Bのビット番号2にビット番号3の値が代入されることを意味します。Bit_stringのビット長を超えたインデックス参照を行うとエラーになります。

> メモ：
>
> Bit_stringの第1ビットのインデックス値は0ですが、SDLの他のほとんどの文字列型のインデックス値は1から始まります。

Boolean

`Boolean newtype`は、`false`と`true`の2つの値のみを取ります。以下に、Boolean値で利用できる演算子を示します。

```
"not" : Boolean -> Boolean
"and" : Boolean, Boolean -> Boolean
"or"  : Boolean, Boolean -> Boolean
"xor" : Boolean, Boolean -> Boolean
"=>"  : Boolean, Boolean -> Boolean
```

これらの演算子は、以下のように定義されています。

- `not`:

  値を反転します。

  ```
  not false = true
  not true  = false
  ```

- `and`:

  2つのパラメータがtrueの場合はtrueとなり、それ以外の場合はfalseとなります。

  ```
  false and false = false
  false and true  = false
  true  and false = false
  true  and true  = true
  ```

- `or`:

  2つのパラメータがfalseの場合はfalseとなり、それ以外の場合はtrueとなります。

  ```
  false or false = false
  false or true  = true
  true  or false = true
  true  or true  = true
  ```

- `xor`:
  2つのパラメータが異なる場合はtrueとなり、それ以外の場合はfalseとなります。

```
false xor false = false
false xor true  = true
true  xor false = true
true  xor true  = false
```

- `=>` (含蓄):
  第1パラメータがtrueで第2パラメータがfalseの場合はfalseとなり、それ以外の場合はtrueとなります。

```
false => false = true
false => true  = true
true  => false = false
true  => true  = true
```

前述のBitソートの機能のほとんどは、Booleanソートと共通です。`0`と`false`、`1`と`true`を置き換えると、2つのソートは等価です。通常は、Bitではなく Booleanを使用することをお勧めします。詳細については、44ページの「Bit」を参照してください。

Character

`character`ソートは、ASCII文字を表すために使用します。文字を表示する場合は以下の表記方法のリテラルによって表します。

```
'a'  '-'  '?' '2'  'P'  ''''
```

シングルクォート(')文字をリテラルとして表現する場合は、2つ続けて記述することに注意してください。印刷不能文字のために、Characterソートには特別なリテラル名が定義されています。このようなリテラル名を次に示します。以下の各文字列は、文字番号0～31と対応します。

```
NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
CAN,  EM,   SUB,  ESC,  IS4,  IS3,  IS2,  IS1
```

また、次の文字列は文字番号127に対応します。

```
DEL
```

以下に、Characterソートで利用できる演算子を示します。

```
"<"  : Character, Character  -> Boolean;
"<=" : Character, Character  -> Boolean;
">"  : Character, Character  -> Boolean;
">=" : Character, Character  -> Boolean;
num  : Character             -> Integer;
chr  : Integer               -> Character;
```

これらの演算子は、以下のように定義されています。

- <, <=, >, >= :
  これらの比較演算子は、ASCIIテーブルの文字番号を基準にして演算が適用されます。
- num :
  Character値を対応する文字番号に変換します。例：num('A') = 65
- chr :
  整数値を対応する文字に変換します。パラメータが負であったり、255より大きい場合は、まず256の剰余が取られます（Integerソートのmod演算子を使用します）。例：chr(65) = 'A'

Z.100勧告では文字番号が0～127の文字のみ扱うことになっています。しかしIBM Rationalでは、0～255の文字番号を扱えるよう拡張しています。そのため、次の2点に注意してください。

演算子numは、256の剰余を取ってから文字に変換します。Z.100勧告では128の剰余を取ることになっていました。

文字番号128～255に対応して、Characterソートに次のようなリテラルが追加されています。

```
E_NUL, E_SOH, E_STX, E_ETX, E_EOT, E_ENQ, E_ACK, E_BEL,
E_BS,  E_HT,  E_LF,  E_VT,  E_FF,  E_CR,  E_SO,  E_SI,
E_DLE, E_DC1, E_DC2, E_DC3, E_DC4, E_NAK, E_SYN, E_ETB,
E_CAN, E_EM,  E_SUB, E_ESC, E_IS4, E_IS3, E_IS2, E_IS1,
' ',   '¡',   '¢',   '£',   '¤',   '¥',   '¦',   '§',
'¨',   '©',   'ª',   '«',   '¬',   '',   '®',   '¯',
'°',   '±',   '²',   '³',   '´',   'µ',   '¶',   '·',
' ',   '¹',   'º',   '»',   '¼',   '½',   '¾',   '¿',
'À',   'Á',   'Â',   'Ã',   'Ä',   'Å',   'Æ',   'Ç',
'È',   'É',   'Ê',   'Ë',   'Ì',   'Í',   'Î',   'Ï',
'Ð',   'Ñ',   'Ò',   'Ó',   'Ô',   'Õ',   'Ö',   '×',
'Ø',   'Ù',   'Ú',   'Û',   'Ü',   'Ý',   'Þ',   'ß',
'à',   'á',   'â',   'ã',   'ä',   'å',   'æ',   'ç',
'è',   'é',   'ê',   'ë',   'ì',   'í',   'î',   'ï',
'ð',   'ñ',   'ò',   'ó',   'ô',   'õ',   'ö',   '÷',
'ø',   'ù',   'ú',   'û',   'ü',   'ý',   'þ',   'ÿ';
```

Charstring

Charstringソートは、文字列または文字シーケンスを表すために使用します。Charstring値の長さに制限はありません。Charstringリテラルは、2つのシングルクォート('abc')で囲まれた文字シーケンスとして記述します。Charstringにシングルクォート(‘)が含まれるときは、2つ続けて記述します。

```
'abcdef 0123'
'$%@^&'
'1''2''3' /* 1'2'3を表します */
''        /* 空のCharstring */
```

以下に、Charstringで利用できる演算子を示します。

```
mkstring  : Character              -> Charstring;
length    : Charstring             -> Integer;
first     : Charstring             -> Character;
last      : Charstring             -> Character;
"//"      : Charstring, Charstring -> Charstring;
substring : Charstring, Integer, Integer
                                   -> Charstring;
```

これらの演算子は、以下のように定義されています。

- mkstring:
  Characterの値を取って、長さ1のCharstringに変換します。たとえば、Characterデータ型の変数cは、mkString(c)によって文字cの要素を持つCharstringを表します。

- length:
  Charstringをパラメータに取り、文字数を返します。
  length ('hello') = 5

- first:
  Charstringをパラメータに取り、最初のCharacter値を返します。Charstringの長さが0の場合にfirst演算子を呼び出すとエラーになります。
  first ('hello') = 'h'

- last:
  Charstringをパラメータに取り、最後のCharacter値を返します。Charstringの長さが0の場合にlast演算子を呼び出すとエラーになります。
  last ('hello') = 'o'

- // (連結):
  演算結果は、第1パラメータの全要素に続いて、第2パラメータの全要素を連結したCharstringとなります。
  'he' // 'llo' = 'hello'

- `substring:`
  第1パラメータとしてCharstringを取り、部分的なコピーを返します。コピーの先頭位置は、第2パラメータに与えるインデックス値で指定します（注: 先頭文字のインデックス値は1です）。コピーの長さは、第3パラメータで指定します。第1パラメータの長さを超える要素にアクセスするとエラーになります。
  `substring ('hello', 3, 2) = 'll'`

Charstring変数にインデックス値を指定することで、Charstringの文字要素にアクセスすることもできます。例えば、`C`というCharstring変数に対して、以下のように記述できます。

```
task C(2) := C(3);
```

上記の式は、変数`C`の文字番号2に文字番号3の値が代入されることを意味します。

> メモ：
>
> Charstringの先頭の文字のインデックス値は1です。

IA5String、NumericString、PrintableString、VisibleString

これらの文字列型は、すべてZ.105専用であり、値に含まれる文字を制限するCharstringのシンタイプです。これらのソートは、主に同名のASN.1データ型に対応するデータ型として使います。以下に各文字列型の制限を示します。

- `IA5String:`
  NULからDELまで、すなわち文字番号が0～127の文字のみが使用可能です。
- `NumericString:`
  `'0'`から`'9'`と`' '`（スペース）のみが使用可能です。
- `PrintableString:`
  `'A':'Z', 'a':'z', '0':'9', ' ', '''':')', '+':'/', ':', '=', '?'`のみが使用可能です。
- `VisibleString:`
  `' ':'~'`のみが使用可能です。

これらのデータ型は、ASN.1やTTCN用に記述するコードでのみ使用することをお勧めします。それ以外の場合は、Charstringを使用してください。

Duration、Time

`Time`と`Duration`ソートは主に、タイマとともに利用します。`Set`ステートメントの第1パラメータには、タイムアウトの時刻を指定します。この時刻の値は、Timeソートでなければなりません。

TimeとDurationのリテラルの表現は、以下のように実際の値と同じです。

```
245.72  0.0032  43
```

以下に、Durationソートで利用できる演算子を示します。

```
"+"  : Duration, Duration -> Duration;
"-"  : Duration           -> Duration;
"-"  : Duration, Duration -> Duration;
"*"  : Duration, Real     -> Duration;
"*"  : Real,     Duration -> Duration;
"/"  : Duration, Real     -> Duration;
">"  : Duration, Duration -> Boolean;
"<"  : Duration, Duration -> Boolean;
">=" : Duration, Duration -> Boolean;
"<=" : Duration, Duration -> Boolean;
```

以下に、Timeソートで利用できる演算子を示します。

```
"+"  : Time,     Duration -> Time;
"+"  : Duration, Time     -> Time;
"-"  : Time,     Duration -> Time;
"-"  : Time,     Time     -> Duration;
"<"  : Time,     Time     -> Boolean;
"<=" : Time,     Time     -> Boolean;
">"  : Time,     Time     -> Boolean;
">=" : Time,     Time     -> Boolean;
```

これらの演算子は、実際の数値に対して使用する、一般的な算術演算子に対応しているので、理解は容易です。SDLには、Time値を返す、パラメータを取らない演算子nowがあります。nowは、現在のグローバル システム タイムを返します。

“時刻”を表す場合はTimeを使用し、“時間間隔”を表す場合はDurationを使います。SDLでは、時間の単位を指定しません。SDL Suiteでは通常、時間の単位は1秒です。

例3: SDLのタイマ ——————————————————————————

```
SET (now + 2.5, MyTimer)
```

このステートメントを実行すると、その時点から2.5単位時間（通常は秒）後に、`MyTimer`というSDLタイマがタイムアウトになります。

——————————————————————————————————————

SDLでは、TimeとDuration（およびReal）は、実数を算術的に処理します。ただし実装においては、これらの値の範囲や精度は制限されます。

Integer、Natural

SDLの`Integer`ソートは、整数を表すために使います。`Natural`は、0以上の整数のみを許容するIntegerのシンタイプです。

Integerのリテラルは、以下のように通常の整数を表現する構文を使用して定義します。

```
0  5  173  1000000
```

負の整数は、単項演算子（-）を使用して表します。以下に、Integerソートで利用できる演算子を示します。

```
"-"   : Integer           -> Integer;
"+"   : Integer, Integer  -> Integer;
"-"   : Integer, Integer  -> Integer;
"*"   : Integer, Integer  -> Integer;
"/"   : Integer, Integer  -> Integer;
"mod" : Integer, Integer  -> Integer;
"rem" : Integer, Integer  -> Integer;
"<"   : Integer, Integer  -> Boolean;
">"   : Integer, Integer  -> Boolean;
"<="  : Integer, Integer  -> Boolean;
">="  : Integer, Integer  -> Boolean;
float : Integer           -> Real;
fix   : Real              -> Integer;
```

これらの演算子は、以下のように定義されています。

- `-` (単項演算子、つまり、単一のパラメータを取ります):
  負の値を表します。例：-5.
- `+`, `-`, `*`:
  これらの演算子は、算術演算子に対応します。

- `/`:
  整数の除算です。例：`10/5 = 2, 14/5 = 2, -8/5 = -1`
- `mod`, `rem`:
  整数の除算の法と剰余です。`mod`は常に正の値を返します。また、`rem`は負の値を返すことがあります。例：
  `14 mod 5 = 4, 14 rem 5 = 4, -14 mod 5 = 1, -14 rem 5 = -4`
- `<`, `<=`, `>`, `>=`:
  これらの演算子は、算術演算子に対応します。
- `float`:
  この演算子は、整数値を対応する実数に変換します。例：
  `float (3) = 3.0`
- `fix`:
  この演算子は、実数値を対応するInteger値に変換します。変換によって、実数値の小数部分が切り捨てられます。
  `fix(3.65) = 3,  fix(-3.65) = -3`

NULL

`NULL`は、Z.105でASN.1に対応するために定義されたソートです。ASN.1で定義された比較的古いプロトコルでは、NULLはかなりの頻度で使用されています。ASN.1では、その後の拡張によって、代替手段を利用できるため、通常はNULLを使用すべきではありません。NULLソートは、単一の値NULLのみを保持します。

Object_identifier

Z.105固有のソートである`Object_identifier`も、ASN.1から導入されたものです。オブジェクト識別子は通常、プロトコルやコード化アルゴリズムなどの広く知られている定義を識別するために使います。たとえば、あるプロトコルを使って、一方のアプリケーションが、他方のアプリケーションに“バージョンXプロトコルのサポート”を通知する場合など、オープンエンドアプリケーションにおいてオブジェクト識別子はよく使用されます。このとき“バージョンXプロトコル”は、オブジェクト識別子によって識別されます。

Object_identifier値は、Natural値のシーケンスによって表現します。このソートには、emptystringというリテラルがあります。emptystringは、長さが0のObject_identifierを表すために使います。以下に、Object_identifierソートで利用できる演算子を示します。

```
mkstring  : Natural           -> Object_identifier
length    : Object_identifier -> Integer
first     : Object_identifier -> Natural
last      : Object_identifier -> Natural
"//"      : Object_identifier, Object_identifier
                              -> Object_identifier
substring : Object_identifier, Integer, Integer
                              -> Object_identifier
append    : in/out Object_identifier, Natural;
(. .)     : * Natural         -> Object_identifier
```

これらの演算子は、以下のように定義されています。

- mkstring:
  Natural値を1つ取り、長さ1のObject_identifierに変換します。
  mkstring (8)は、1つの要素（つまり8）を持つObject_identifierを表します。

- length:
  Object_identifierをパラメータに取り、オブジェクト要素（つまりNatural値）の数を返します。
  length (mkstring (8)//mkstring(6)) = 2
  length (emptyString) = 0

- first:
  Object_identifierをパラメータに取り、の最初のNatural値を返します。
  Object_identifierの長さが0の場合にfirst演算子を呼び出すとエラーになります。
  first (mkstring (8)//mkstring(6)) = 8

- last:
  Object_identifierをパラメータに取り、最後のNatural値を返します。
  Object_identifierの長さが0の場合にlast演算子を呼び出すとエラーになります。
  last (mkstring (8)//mkstring(6)) = 6

- // (連結):
  演算結果は、第1パラメータの全要素の後に、第2パラメータの全要素を連結したObject_identifierになります。
  mkstring (8) // mkstring (6)は、8に6が連結された2つの要素を持つObject_identifierを表します。

- `substring:`
  第1パラメータにObject_identifierを取り、部分的なコピーを返します。コピーの先頭位置は、第2パラメータに与えるインデックス値で指定します（注：先頭のNatural値のインデックス値は1です）。コピーの長さは、第3パラメータで指定します。第一パラメータの長さを超える要素にアクセスするとエラーになります。
  ```
  substring(mkstring(8)//mkstring(6),2,1) =mkstring(6)
  ```

- `append:`
  IBM Rationalが拡張した演算子で、あるObject_identifierの末尾に新しいコンポーネントを追加します。第1パラメータに1つの変数、第2パラメータにNatural値を取ります。演算の結果、第1パラメータの変数は、Object_identifier値の末尾に第2パラメータを追加した値に更新されます。この演算子を追加したのは、たとえば
  ```
  task append(V, 12);
  ```
  と書いた方が、
  ```
  task V := V // mkstring(12);
  ```
  よりも効率よく処理できるからです。

> 注意！
>
> `append` 演算子を使用しても、ストリングに対するサイズの制約はチェックされません。
>
> 範囲チェックを実行する場合は、`append` 演算子ではなく、`concat` 演算子を使用する必要があります。

- `(. .):`
  IBM Rationalが拡張した式で、演算子の形はしていませんが、暗黙のmake演算子を適用することを表します。make演算子はNatural値のシーケンスをパラメータとし、各値を順に並べたObject_identifierを返します。たとえば
  ```
  Obj_id_var := (. 1, 2, 3 .)
  ```
  とすると、1、2、3を順に並べたObject_identifierが返されます。

Object_identifier変数にインデックス値を指定することで、Object_identifierのNatural要素にアクセスすることもできます。例えば、Cというobject_identifier変数に対して、以下のように記述できます。

```
task C(2) := C(3);
```

上記の式は、変数Cのインデックス2のNatural値にインデックス3のNatural値が代入されることを意味します。Object_identifierの先頭のNatural値のインデックス値は、1であることに注意してください。Object_identifierの長さを超えたインデックスを指定するとエラーになります。

Octet

Z.105固有のソートであるOctetは、0～255までの8ビットの値を表すために使います。このソートは、C言語ではunsigned charに該当します。Octetソートには、明示的なリテラルはありません。しかし、以下に説明するi2oとo2i変換演算子を使用すれば簡単に値を定義できます。

以下に、Octetソートで利用できる演算子を示します。

```
"not"     : Octet                    -> Octet;
"and"     : Octet, Octet             -> Octet;
"or"      : Octet, Octet             -> Octet;
"xor"     : Octet, Octet             -> Octet;
"=>"      : Octet, Octet             -> Octet;
"<"       : Octet, Octet             -> Boolean;
"<="      : Octet, Octet             -> Boolean;
">"       : Octet, Octet             -> Boolean;
">="      : Octet, Octet             -> Boolean;
shiftl    : Octet, Integer           -> Octet;
shiftr    : Octet, Integer           -> Octet;
"+"       : Octet, Octet             -> Octet;
"-"       : Octet, Octet             -> Octet;
"*"       : Octet, Octet             -> Octet;
"/"       : Octet, Octet             -> Octet;
"mod"     : Octet, Octet             -> Octet;
"rem"     : Octet, Octet             -> Octet;
i2o       : Integer                  -> Octet;
o2i       : Octet                    -> Integer;
bitstr    : Charstring               -> Octet;
hexstr    : Charstring               -> Octet;
```

これらの演算子は、以下のように定義されています。

- not, and, or, xor, =>:
  Octetの各8ビットに対して、対応するビット演算子を適用します。例：
  `not bitstr ('00110101') = bitstr ('11001010')`
- <, <=, >, >=:
  Octet値で使用する通常の比較演算子です。

- `shiftl, shiftr:`
  これらの演算子はIBM Rational専用であり、C言語の左右のシフト演算子と同様に定義できます。つまり、`shiftl(a,b)`はC言語の`a<<b`に相当します。
  ```
  shiftl (bitstr('1'), 4) = bitstr('10000')
  shiftr (bitstr('1010'), 2) = bitstr ('10')
  ```
- `+, -, *, /, mod, rem:`
  これらの演算子は、対応する算術演算子に一致します。しかし、すべての演算には、256の剰余が適用されます。
  ```
  i2o(250) + i2o(10) = i2o(4), o2i(i2o(4)-i2o(6)) = 254
  ```
- `i2o:`
  この演算子は、IBM Rational専用であり、整数値を対応するOctet値に変換します。
  ```
  i2o (128) = hexstr ('80')
  ```
- `o2i:`
  この演算子は、IBM Rational専用であり、Octet値を対応する整数値に変換します。
  ```
  o2i (hexstr ('80')) = 128
  ```
- `bitstr:`
  この演算子は、IBM Rational専用であり、8つのBit値（“0”と“1”）を持つ文字列をOctet値に変換します。
  ```
  bitstr('00000011') = i2o(3)
  ```
- `hexstr:`
  この演算子は、IBM Rational専用であり、2つのHEX値（“0” - “9”、“a” - “f”、“A” - “F”）を持つ文字列をOctet値に変換します。
  ```
  hexstr('01') = i2o(1), hexstr('ff') = i2o(255)
  ```

また、Octet変数にインデックスを付けることで、Octet値の各ビットを読み取ることができます。インデックスは、0～7の範囲にする必要があります。

Octet_string

Z.105固有のソートである`Octet_string`は、`Octet`値のシーケンスを表します。シーケンスの長さに制限はありません。以下に、Octet_stringソートで利用できる演算子を示します。

```
mkstring     : Octet           -> Octet_string;
length       : Octet_string    -> Integer;
first        : Octet_string    -> Octet;
last         : Octet_string    -> Octet;
"//"         : Octet_string, Octet_string
                               -> Octet_string;
substring    : Octet_string, Integer, Integer
                               -> Octet_string;
bitstr       : Charstring      -> Octet_string;
hexstr       : Charstring      -> Octet_string;
bit_string   : Octet_string    -> Bit_string;
octet_string : Bit_string      -> Octet_string;
```

これらの演算子は、以下のように定義されています。

- `mkstring`:
  この演算子は、Octet値を取り、長さ1のOctet_stringに変換します。
  `mkstring (i2o(10))`は、要素を1つ持ったOctet_stringを表します

- `length`:
  Octet_stringをパラメータに取り、Octet値の数を返します。
  ```
  length (i2o (8)//i2o (6)) = 2
  length ( hexstr ('0f3d88')) = 3
  length ( bitstr ('')) = 0
  ```

- `first`:
  Octet_stringをパラメータに取り、最初のOctet値を返します。Octet_stringの長さが0の場合にfirst演算子を呼び出すとエラーになります。
  ```
  first ( hexstr ('0f3d88')) = hexstr('0f') (= i2o(15))
  ```

- `last`:
  Octet_stringをパラメータに取り、最後のOctet値を返します。Octet_stringの長さが0の場合にlast演算子を呼び出すとエラーになります。
  ```
  last ( hexstr ('0f3d88')) = hexstr('88') (= i2o(136))
  ```

- `//` (連結):
  演算結果は、第1パラメータ内の全要素に続いて、第2パラメータの全要素が連結された値を持つOctet_stringになります。
  ```
  hexstr('0f3d')//hexstr('884F') = hexstr('0f3d884f')
  ```

- `substring:`
  第1パラメータにOctet_stringを取り、部分的なコピーを返します。コピーの先頭位置は、第2パラメータに与えるインデックス値で指定します。コピーの長さは第3パラメータで指定します。第1パラメータの長さを超える要素にアクセスしようとすると、エラーになります。
  ```
  substring(hexstr('0f3d889C'), 3, 2) = hexstr('889c')
  ```

- `bitstr:`
  この演算子は、IBM Rational専用であり、0と1の2つの文字のみからなる文字列を、同じ長さのOctet_stringに変換します。このときOctet_stringの各Octet要素には、文字列の8つのビット シーケンスで定義された値が設定されます。文字列の長さが8の倍数でない場合は、0が補填されます。
  ```
  bitstr ('101') = bitstr ('10100000')
  ```

- `hexstr:`
  この演算子は、IBM Rational専用であり、HEX値（0-9、A-F、a-f）を含む文字列をOctet_stringに変換します。各HEX値のペアが、Octet_string内で1つのOctet要素に変換されます。文字列の長さが2の倍数でない場合は、0が補填されます。
  ```
  hexstr ('f') = hexstr ('f0')
  ```

- `bit_string and octet_string:`
  この2つの演算子は、Bit_stringとOctet_string間で値の変換を行います。

Octet_string変数にインデックス値を指定することで、Octet_stringのOctet要素にアクセスすることもできます。例えば、CというOctet_string変数に対して、以下のように記述できます。

```
task C(2) := C(3);
```

上記の式は、変数Cのインデックス2のOctet値にインデックス3のOctet値が代入されることを意味します。Octet_stringの長さを超えたインデックス値を指定するとエラーになります。

> メモ：
>
> Octet_stringの先頭のOctetのインデックス値は1です。

Pid

Pidソートは、プロセスインスタンスを処理するための参照として使います。Pidは、Nullというリテラルを1つ持ちます。これ以外の値は、SDLの定義済みのデフォルトの変数Self、Sender、Parent、Offspringから得ることができます。

Real

Realは、数学的な実数値を表すために使います。ただし、実装ではサイズや精度が制限されます。Realリテラルの例を以下に示します。

```
2.354  0.9834  23  1000023.001
```

以下に、Realソートで利用できる演算子を示します。

```
"-"  : Real          -> Real;
"+"  : Real, Real    -> Real;
"-"  : Real, Real    -> Real;
"*"  : Real, Real    -> Real;
"/"  : Real, Real    -> Real;
"<"  : Real, Real    -> Boolean;
">"  : Real, Real    -> Boolean;
"<=" : Real, Real    -> Boolean;
">=" : Real, Real    -> Boolean;
```

上記の演算子はすべて、一般的な算術演算を表します。

ユーザー定義のソート

これまでに説明したすべての定義済みソートとシンタイプは、変数の宣言などで直接使用できます。しかし、システムに新しいソートやシンタイプを定義して、特定のプロパティを表現しなければならない場合もあります。ユーザーが定義したソートやシンタイプは、定義したユニットとその中のすべてのサブユニットで使用できます。

シンタイプ

*シンタイプ*を定義すると、ベースのデータ型と完全に互換性のある新しいデータ型の名前を使用できます。したがって、ベースとなるデータ型の変数を使用できる任意の場所でシンタイプの変数を使用できます。ベースのデータ型とシンタイプが異なる唯一の点は、シンタイプに対して範囲チェックが実行されることです。ただし、仮の入出力パラメータに対応する実パラメータは、仮パラメータと同じシンタイプでなければなりません。そうでなければ範囲チェックが正しく実行されないことになります。

シンタイプは、以下の用途に使用します。

- 既存のデータ型に新しい名前を定義する
- 既存のデータ型と同じ機能を持ち、値の範囲を制限した新しいデータ型を定義する
- 配列を使ってインデックス ソートを定義する

例4: シンタイプの定義

```
syntype smallint = integer
  constants 00:10:00
endsyntype;
```

この例では、smallintが新しいデータ型の名前であり、Integerがベースのデータ型、そして0:10が範囲条件になります。範囲条件は、さらに複雑な定義も可能であり、以下のような条件を複数個定義できます（Xは、条件値です）。

- =X　単一の値Xが許容されます。
- X　=Xと同じです。
- /=X　X以外の任意の値が許容されます。
- >X　>Xの任意の値が許容されます。
- >=X　>=Xの任意の値が許容されます。
- <X　<Xの任意の値が許容されます。
- <=X　<=Xの任意の値が許容されます。
- X:Y　>=Xおよび<=Yの任意の値が許容されます。

例5: シンタイプの定義

```
syntype strangeint = integer
  constants <-5, 0:3, 5, 8, >=13
endsyntype;
```

この例で定義されたデータ型では、<-5, 0, 1, 2, 3, 5, 8, >=13に対応する値が使用可能です。

シンタイプの範囲チェックは、以下の場合にテストが実行されます（使用する変数、信号パラメータ、形式パラメータはシンタイプで定義されているものと仮定します）。

- 変数への代入
- 出力信号（およびインポート プロシージャ コールとリモート プロシージャ コールで使用される暗黙の信号）の信号パラメータへの値の代入
- プロシージャ コールでのINパラメータへの値の代入
- 生成要求アクションでのプロセス パラメータへの値の代入

- 入力信号の変数への値の代入
- 演算子パラメータへの値の代入（および演算子の結果の出力）
- セット、リセット、またはアクティブ時のタイマ パラメータへの値の代入

列挙ソート

*列挙ソート*は、列挙される値のみを含むソートです。システムのプロパティが取る値の種類が比較的少なく、それぞれの値が名前を持つ場合は、これらの要素を定義するために列挙ソートを使用することが最適の方法と考えられます。以下に、オフ、スタンバイ、サービス モードの3つのステータスを持つキーを記述するのに適したソートを示します。

例6: 列挙ソート

```
newtype KeyPosition
  literals Off, Stand_by, Service_mode
endnewtype;
```

`KeyPosition`ソートの変数は、`literals`で指定した3つの値のいずれかを取ることができますが、それ以外の値は取ることができません。

構造体

SDLにおいて*構造体*の概念は、データをまとめて、1つの集合として定義するために使います。ほとんどのプログラミング言語にもこれに相当する機能が用意されています。C言語では同様に構造体と呼ばれています。また、Pascalではレコードの概念がこれに相当します。たとえば、名前、住所、電話番号など、多くのプロパティ（または属性）を持つ人物は、以下のように記述できます。

```
newtype Person struct
  Name        Charstring;
  Address     Charstring;
  PhoneNumber Charstring;
endnewtype;
```

構造体には、複数のコンポーネントを定義することができ、それぞれのコンポーネントには名前とデータ型を定義することができます。上記の構造体の定義に対して、以下の変数名を定義すると、構造体全体の値を、代入や等号の評価などによって直接処理することができます。

```
dcl p1, p2 Person;
```

また、構造体変数内の個々のコンポーネントを選択したり、変更することができます。

```
task p1 := (. 'Peter', 'Main Road, Smalltown',
```

```
                ’+46 40 174700’ .);
task BoolVar := p1 = p2;
task p2 ! Name := ’John’;
task CharstringVar := p2 ! Name;
```

最初のタスクは、構造体レベルの代入です。右側の式(..)は、すべての構造体に存在する暗黙のmake演算子です。make演算子は、最初のコンポーネント ソートの値を取り、続いて第2およびそれ以降のコンポーネント ソートの値を取ります。そして、個々のコンポーネントの値で構成された構造体値を返します。上記の例では、p1変数のNameコンポーネントに値’Peter’が代入されます。第2のタスクは、2つの構造体式の等価テストを表しています。第3と第4のタスクは、構造体のコンポーネントにアクセスする方法を示しています。コンポーネントは、以下の記述方法によって指定できます。

< 変数名 > ! < コンポーネント名 >

上記のコンポーネントの指定は、式（このとき通常、*extract*が呼び出されます）や代入の左辺（このとき通常、*modify*が呼び出されます）で実行できます。

ビット フィールド

*ビット フィールド*は、構造体コンポーネントのビット サイズを定義します。この機能はSDL勧告には含まれていませんが、C言語のビット フィールドの生成をSDLでサポートするためにIBM Rationalによって導入されました。これは、ビット フィールドの構文や意味がC言語のビット フィールド機能に対応することを意味します。

例7: ビット フィールド ———————————————————————

```
newtype example struct
  a Integer     : 4;
  b UnsignedInt : 2;
  c UnsignedInt : 1;
                : 0;
  d Integer     : 4;
  e Integer;
endnewtype;
```

———————————————————————————————————————

ビット フィールドには、以下の規則が適用されます。

- ビット フィールド サイズ、つまり": X"（Xは整数）の意味は、C言語と同じです。SDLからCコードを生成する場合、SDL 構造体から生成された": X"がそのままC構造体にコピーされます。
- SDLの": 0"は、C言語では"int : 0"に翻訳されます。
- C言語のビット フィールド コンポーネントで使用できるのは、intおよびunsigned intのみですが、同じ規則がSDLにも適用されます。つまり

SDLでは、ctypesパッケージのIntegerとUnsignedIntのみが使用可能です。

ビット フィールドは、SDLからC言語のビット フィールドを生成する必要がある場合にのみ使います。ビット フィールドは、定数句を持ったシンタイプの代用として使用すべきではありません。SDL Suiteでは、ビット フィールドのサイズの違反はチェックされません。

Optionalとデフォルト値

ASN.1データ型からSDLソートへの翻訳を簡単にするために、2つの新しい機能が構造体に導入されています。構造体のコンポーネントには、*オプション*を指定することができ、*デフォルト値*を定義することができます。これらの機能は主にASN.1のデータ型と共に利用します。これらはSDL-96の標準機能ではないので、他の目的に使うことは推奨できません。

例8: Optionalおよびデフォルト値

```
newtype example struct
  a Integer     optional;
  b Charstring;
  c Boolean     := true;
  d Integer     := 4;
  e Integer     optional;
endnewtype;
```

コンポーネントcとdは、定義されたデフォルト値によって初期化されます。

*オプション*コンポーネントは、構造体値内に存在するかどうかを指定できます。初期状態では、オプション コンポーネントは存在しません。オプション コンポーネントの存在は、値が代入された後に生じます。存在しないコンポーネントにアクセスするとエラーになります。オプション コンポーネントが存在するかどうかを確認するには、<*コンポーネント名*>presentという暗黙の演算子を呼び出します。

上記の例では、apresent(v)とepresent(v)を呼び出すことによって、変数vに格納されている値の中にaとeコンポーネントが存在するかどうかをテストできます。存在するコンポーネントは、<コンポーネント名>absentという暗黙の演算子を呼び出して、存在しないことにすることができます。

上記の例では、aabsent(v)やeabsent(v)を呼び出すことによって、コンポーネントが存在しないことにすることができます。なお、absent演算子には戻り値がありません。

オプションコンポーネントと同様に、デフォルト値を持つコンポーネントにはpresentとabsentの演算子があります。ただし、この場合にはオプションコンポーネントと違った意味になります。なぜなら、デフォルト値を持つコンポーネントには値が常にあります。その場合におけるpresentとabsentはASN.1値のエンコードとデコードに関わりがあります。エンコードスキームによっては、デフォルト値を持つコンポーネント（つまり、absentとなるもの）がエンコードされません。

デフォルト値を持つコンポーネントはそのデフォルト値で初期化されますが、presentを呼び出すとfalse値が返ります。presentは実際には、「明示的に何らかの値が与えられた」かどうかをテストする演算子なのです。したがって、代入文などで値が与えられるとtrueになります(その値がたまたまデフォルト値と同じであっても同様です)。一方、absent演算子を適用すると、コンポーネントが「存在しない」状態に戻ります。詳しく言うと、コンポーネントの値はデフォルト値になり、present演算子はfalseを返すようになります。

Z.105勧告によると、構造体を構築するmake演算子では、オプション コンポーネントやデフォルト値を持つコンポーネントの値を与えることができません。オプション コンポーネントは常に「存在しない」状態になり、デフォルト値を持つコンポーネントは常にそのデフォルト値で初期化されます。例8に挙げた「struct example」の場合、オプション コンポーネントでもデフォルト値を持つコンポーネントでもない、1つのコンポーネントの値しか与えることができません。つまり、この型の変数vにmake演算子で値を与える場合、

```
task v := (. 'hello' .);
```

というように記述することになります。他のコンポーネントの値は、このあと、一連の代入文で設定してください。

もっと簡潔な記述で、構造体の各コンポーネントの値を設定できるよう、IBM Rationalではmake演算子の代替解釈機能を追加しました。この機能を適用したい場合、[生成]-[解析]-[詳細]-[意味の分析]の順に選択して表示される[実装演算子に省略可能なフィールドを含める]を使って指定してください。

代替make演算子には、パラメータとしてすべてのコンポーネントの値を与えます。存在しないコンポーネント、デフォルト値のままのコンポーネントにしたい場合は、該当箇所を空にしてください。明示的に値を記述すれば、対応するコンポーネントに値が代入されます。例8の構造体の場合、たとえば次のように記述します。

```
task v := (. 1, 'hello', , 10, .);
```

この場合、第1、第2、第4コンポーネントには明示的に値が与えられます。第3、第5コンポーネントは値が存在しないままです。

Choice

新しい*choice*概念は、ASN.1のCHOICE概念を表すためにSDLに導入されています。また、この概念はSDL固有のデータ型を作成するときにも有用です。SDLのchoiceは、C言語では暗黙のタグ フィールドを持った*union*と見なすことができます。

例9: Choice

```
newtype C1 choice
  a Integer;
  b Charstring;
  c Boolean;
endnewtype;
```

この例は、3つのコンポーネントを持ったchoiceを表しています。choiceデータ型の変数は、一度にいずれか1つのコンポーネントのみを持つことができるため、C1の値は、整数値、Charstring値またはBoolean値のいずれかになります。

例10: choiceデータ型の使用

```
DCL var C1, charstr Charstring;

TASK var := a : 5; /* コンポーネント a に代入 */
TASK var!b := 'hello'; /* コンポーネント b に代入
                          (a は存在しなくなります) */
TASK charstr := var!b; /* コンポーネント b を取得 */
```

この例は、choiceデータ型のコンポーネントの変更と抽出方法を示しています。choiceの値を示すためにa:5の表記を使用することと、struct値を示すために(. ... .）の表記を使用する点を除けば、choiceデータ型の変更と抽出方法は、structデータ型の操作と同じです。

存在しないchoiceデータ型のコンポーネントの抽出を試みると、実行時エラーが発生します。したがって、個々の値に対してどのコンポーネントがアクティブであるか確認できなければなりません。このため、choiceに暗黙の演算子が定義されています。

```
var!present
```

上記のvarはchoiceデータ型の変数です。v変数はアクティブ コンポーネントの名前を値として返します。これは、各choiceコンポーネントと同じ名前のリテラルで構成された列挙型が、暗黙的に生成されることによって実現されています。なお、この列挙型は暗黙的に生成されるため、明示的に使用することはできません。上記の例では、次のように以下の記述によってチェックが可能です。

```
var!present = b
```

図26に、コンポーネントをチェックするメカニズムを示します。

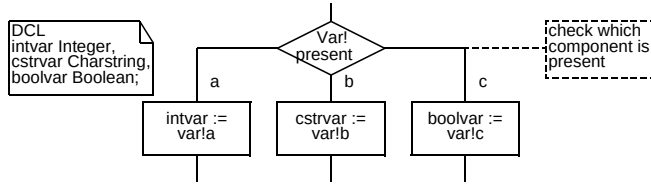


図26: choice内に存在するコンポーネントのチェック

暗黙の<*コンポーネント名*>`present`論理演算子を使用すれば、特定のコンポーネントがアクティブかどうかをテストできます。上の例のコンポーネントbが存在するかどうかをチェックするには、以下のように記述します。

```
bpresent(v)
```

`present`演算子を使用すると、アクティブなコンポーネントを表す情報にアクセスできますが、この情報を変更することはできません。この情報は、choice変数のコンポーネントに値が代入されると自動的に更新されます。

choiceを使用する目的は、メモリや帯域幅の節減にあります。値を保持するコンポーネントは一度に1つだけであることがわかっているため、コンパイラは上書きによってchoiceデータ型が使用する総メモリを節減できます。また、物理的な接続を経由してchoice値を送信する際の所要時間も、同等の構造体を送信するより短縮できます。

choiceの構成体はIBM Rational固有であり、Z.105勧告には含まれていません。ゆえに、移植性が要求されるSDLを記述する場合は、choiceを使用すべきではありません。choiceはSDL Suite #UNIONコード ジェネレータ ディレクティブと置き換えることができます。SDL Suiteでは、choiceに対してより優れたツールがサポートされているため、#UNIONディレクティブの代わりにchoiceを使用することをお勧めします。

Inherits

ソートは、別のソートから情報を*継承*して作成することができます。また、新しいソートに継承する演算子やリテラルを指定したり、新たに演算子やリテラルを追加することもできます。

継承を使う場合は、データ型そのものを変更することはできません。たとえば、構造体を継承しても、その構造体に新しいコンポーネントを追加することはできません。

いままでの継承を使った事例から判断すると、継承は当初考えられたほど有用ではないようです。継承を使用することによって、語義的に無効になる式が数多く発生するため、限定子が多くの場所で必要になります。

例11: Inherits

```
newtype NewInteger inherits Integer
  operators all;
endnewtype;
```

この例では、NewIntegerという新しいデータ型を導入しています。このデータ型は、Integerとは区別されます。つまり、NewIntegerの式や変数を記述しなければならない場所でIntegerを使うことはできません。また、Integerの式や変数を記述しなければならない場所でNewIntegerを使うことはできません。さらに、上記の例では、すべてのリテラルと演算子を継承しているので、整数リテラルである0、1、2…をNewIntegerのリテラルとしてすべて利用できます。したがって、パラメータや戻り値のデータ型がIntegerになっているすべての演算子は、コピーされると、IntegerパラメータがNewIntegerパラメータに置き換えられます。置き換えられる対象は、Integerソート用に定義されている演算子だけでなく、すべての演算子が含まれるため、以下のような予期しない影響が生じる可能性があります。

例12: 継承された演算子

以下の演算子は、パラメータや戻り値のデータ型にIntegerを持ちます。

```
"+" : Integer, Integer -> Integer;
"-" : Integer -> Integer;
"mod" : Integer, Integer -> Integer;
length : Charstring -> Integer;
```

前述のNewIntegerデータ型は、これらの演算子を継承するとともに、パラメータや戻り値のデータ型にIntegerを持つその他のすべての演算子を継承します。上記の演算子の中ではlengthがCharstringソートで定義されていることに注意してください。

```
"+" : NewInteger, NewInteger -> NewInteger;
"-" : NewInteger -> NewInteger;
"mod" : NewInteger, NewInteger -> NewInteger;
length : Charstring -> NewInteger;
```

NewIntegerの宣言によって、

```
decision length(Charstring_Var) > 5;
```

のようなステートメントは、SDLシステムでは正しく動作しなくなります。上の式内のデータ型を特定することはもはやできません。整数を返しIntegerリテラル

と比較されるlengthが存在するのと同時に、NewInteger値を返しNewIntegerリテラルと比較されるlengthも存在します。

継承する演算子を明示的に指定すれば、この種類の問題は回避できます。

例13: Inherits

```
newtype NewInteger inherits Integer
  operators ("+", "-", "*", "/")
endnewtype;
```

今回は、列挙されている演算子のみが継承されるため、先に説明したlengthの問題を回避できます。

他の型を継承するnewtypeは、元の型のデフォルト値は継承しません

定義済みジェネレータ

Array

定義済みのArrayジェネレータは、インデックス ソートとコンポーネント ソートの2つのジェネレータ パラメータを取ります。SDLでは、インデックス ソートとコンポーネント ソートに対する制限はありません。

例14: Arrayの実体化

```
newtype A1 Array(Character, Integer)
endnewtype;
```

上記の例は、インデックス ソートにCharacter、コンポーネント ソートにIntegerを取るArrayジェネレータの実体化を表しています。この定義は、各整数値に対応したCharacter値を格納することができるデータ構造を作成したことになります。したがって、特定のインデックス値に関連付けられたコンポーネントの値を取得するために、配列にインデックスを割り当てることができます。

例15: array型の使用

```
dcl Var_A1 A1;  /* この例では例 15 のソートを想定しています */

task Var_A1 := (. 3 .);
task Var_Integer := Var_A1('a');
task Var_A1('x') := 11;

decision Var_A1 = (. 11 .);
  (true) : ...
  ...
enddecision;
```

上記の例は、配列の使用方法を示しています。最初のタスクには、`(. 3 .)`という式が記述されています。この式では、すべての配列インスタンスで利用できる*make!*演算子を使用しています。この式の目的は、makeによって定義されているすべてのコンポーネントに対して、配列値を設定することにあります。最初のタスクでは、すべての配列コンポーネントに値3を代入しています。このタスクは、配列値全体に対する代入であることに注意してください。

第2のタスクでは、インデックス`'a'`の配列コンポーネントの値を抽出して、整数変数`Var_Integer`に代入しています。第3のタスクでは、インデックス`'x'`の配列コンポーネントの値を新しい値11に変更しています。第2と第3のタスクでは、すべての配列のインスタンスに存在する*extract!*演算子と*modify!*演算子が適用されています。extract!、modify!、make!演算子は、上記の例の用法でのみ使用できます。なお、これらの演算子の名前を直接使用することはできません。

最後のステートメントでは、2つの配列値の等価テストを行うdecisionが実行されています。等価と非等価は、代入と同様にSDLのすべてのソートに対して定義されています。

配列の典型的な使用方法に、一定数の同じソートの要素を定義する方法があります。Integerのシンタイプは、インデックス ソートとしてしばしば使います。以下の例では、11個のPidの配列が0から10のインデックス値で定義されています。

例16: 典型的な配列の定義

```
syntype indexsort = Integer
  constants 00:10:00
endsyntype;

newtype PidArray Array (indexsort, Pid)
endnewtype;
```

一般的なプログラミング言語と異なり、SDLのインデックス ソートにはいかなる制限もありません。ほとんどのプログラミング言語では、列挙を可能にするためにインデックス データ型に対して有限範囲を定義する必要があります。たとえばC言語では、配列のサイズを整数の定数として指定します。この場合、配列のインデックス値は0からそのサイズ-1の範囲で変化しますが、SDLには、このような制限はありません。

例17: 要素数に制限のない配列

```
newtype RealArr Array (Real, Real)
endnewtype;
```

上記の配列では、インデックス データ型にReal値を使用しています。つまり、要素数には制限がないことを意味します。それにもかかわらずこの配列は、ここまでに説明した他の配列と同じ機能を持ちます。このような高機能の配列は強力な概念であり、異なるエンティティ間のマッピング テーブルを実装する場合などに使用できます。

例18: マッピング テーブルを実装する配列

```
newtype CharstringToPid Array (Charstring, Pid)
endnewtype;
```

このデータ型は、名前を表すCharstringを、対応するプロセス インスタンスを表すPid値に割り当てるために使用することができます。

String

`String`ジェネレータは、空の文字列値の名前と、コンポーネント ソートの2つのジェネレータ パラメータを取ります。Stringデータ型の値は、コンポーネント ソートの値のシーケンスによって構成されます。このシーケンスの長さにはいかなる制限もありません。たとえばCharstring定義済みソートは、Stringジェネレータによって定義されています。

例19: Stringジェネレータ

```
newtype S1 String(Integer, empty)
endnewtype;
```

この例では、Integerコンポーネントを持つStringが定義されています。空の文字列、つまり長さゼロの文字列は`empty`リテラルによって表されます。

以下に、Stringのインスタンスで利用できる演算子を示します。

```
mkstring  : Itemsort                -> String
length    : String                  -> Integer
first     : String                  -> Itemsort
last      : String                  -> Itemsort
"//"      : String, String          -> String
substring : String, Integer, Integer -> String
append    : in/out String, Itemsort;
(. .)     : * Itemsort              -> String
```

これらの演算子の定義において、Stringはnewtypeの文字列すなわち、上記の例のS1に置き換える必要があり、Itemsortはコンポーネント ソートのパラメータすなわち、上記の例ではIntegerに置き換える必要があります。以下に、これらの演算子の振る舞いを説明するとともに、String(Integer,empty)データ型に基づいた例を示します。

- mkstring:
  Itemsort値を1つ取り、長さ1のStringに変換します。
  mkstring (-3)は、値-3の整数を1つ持った文字列を返します。

- length:
  Stringをパラメータとして取り、Itemsort値要素の数を返します。
  length (empty) = 0, length(mkstring (2)) = 1

- first:
  Stringをパラメータとして取り、最初のItemsort要素の値を返します。Stringの長さが0の場合にfirst演算子を呼び出すとエラーになります。
  first (mkstring (8) // mkstring (2)) = 8

- last:
  Stringをパラメータとして取り、最後のItemsort要素の値を返します。Stringの長さが0の場合にlast演算子を呼び出すとエラーになります。
  last (mkstring (8) // mkstring (2)) = 2

- // (連結):
  演算結果は、第1パラメータ内の全要素に続いて、第2パラメータの全要素が連結された値を持つStringになります。
  mkstring (8) // mkstring(2)は、8の次に2を連結した2つの要素から成る文字列を返します。

- `substring:`
  第1パラメータにStringを取り、部分的なコピーを返します。コピーの先頭位置は、第2パラメータに与えられるインデックス値で指定します（注: 先頭のItemsort要素のインデックス値は1です)。コピーの長さは第3パラメータで指定します。第1パラメータの長さを超える要素にアクセスしようとすると、エラーになります。
  ```
  substring (mkstring (8) // mkstring(2), 2, 1)
  = mkstring(2)
  ```
- `append:`
  IBM Rationalが拡張した演算子で、あるStringの末尾に新しいコンポーネントを追加します。第1パラメータに変数、第2パラメータにItemsort値を取ります。演算の結果、第1パラメータの変数は、IBM RationalString値の末尾に第2パラメータを追加した値に更新されます。この演算子を追加したのは、たとえば
  ```
  task append(V, Comp);
  ```
  と書いた方が、
  ```
  task V := V // mkstring(Comp);
  ```
  よりも効率よく処理できるからです。
- `(. .):`
  IBM Rationalが拡張した式で、演算子の形はしていませんが、暗黙のmake演算子を適用することを表します。make演算子はItemsort値のシーケンスをパラメータとし、各値を順に並べたStringを返します。たとえば
  ```
  String_var := (. 1, 2, 3 .)
  ```
  とすると、1、2、3を順に並べたStringが返されます。

String変数にインデックス値を指定することで、StringのItemsort要素にアクセスすることもできます。たとえば、CというString実体化変数に対して、以下のように記述できます。

```
task C(2) := C(3);
```

上記の式は、変数CのItemsort要素2にItemsort要素3の値を代入することを意味します。なお、Stringの最初の要素のインデックス値は1であることに注意してください。Stringの長さを超えたインデックスを指定するとエラーになります。

Stringジェネレータは、同じデータ型の項目をまとめたリストを作成するために使用できます。ただし、新しい要素をリストの途中に挿入するなど、一般的なリスト操作を行うには、コンピュータにかなりの処理能力が必要です。

Powerset

Powersetジェネレータは、ジェネレータ パラメータにアイティム ソートを取り、そのアイティム ソートをパワーセットとして実装します。パワーセットの値は、アイティム ソートが取ることのできるそれぞれの値に対して、パワーセットのメンバであるかどうかを表していると見なすことができます。

Powersetは、多くの場合、より単純な他のデータ型の抽象データ型として使うことができます。32ビット ワードをビット パターンとしてモデル化する場合は、0から31の範囲を定義したIntegerのシンタイプに対してパワーセットを定義すれば実現できます。たとえば7がパワーセットのメンバーである場合は、7ビット目がセットされます。

例20: Powersetジェネレータ

```
syntype SmallInteger = Integer
  constants 0:31
endsyntype;

newtype P1 Powerset(SmallInteger)
endnewtype;
```

powersetソートの唯一のリテラルは、要素を含まないpowersetを表すemptyです。powersetソートでは、以下の演算子を利用できます。以下の演算子の定義に上記の例を適用する場合は、Powersetをnewtype名のP1に、またItemsortをItemsortパラメータであるSmallIntegerにそれぞれ置き換えます。

```
"in"   : Itemsort,  Powerset  -> Boolean
incl   : Itemsort,  Powerset  -> Powerset
incl   : Itemsort,  in/out Powerset;
del    : Itemsort,  Powerset  -> Powerset
del    : Itemsort,  in/out Powerset;
length : Powerset             -> Integer
take   : Powerset             -> Itemsort
take   : Powerset,  Integer   -> Itemsort
"<"    : Powerset,  Powerset  -> Boolean
">"    : Powerset,  Powerset  -> Boolean
"<="   : Powerset,  Powerset  -> Boolean
">="   : Powerset,  Powerset  -> Boolean
"and"  : Powerset,  Powerset  -> Powerset
"or"   : Powerset,  Powerset  -> Powerset
(. .)  : * Itemsort           -> Powerset
```

以下に、これらの演算子の定義を示します。演算子の説明に使っている例は、上記のP1 newtypeの定義をベースにしています。また、P1の変数v0_1_2には、0、1、2の要素があることを想定しています。

- in:
  この演算子は、ある値がpowersetのメンバであるかどうかをテストします。
  3 in incl (3, empty)はtrueを返します。
  また、3 in v0_1_2 はfalseを返し、0 in v0_1_2はtrueを返します。

- incl:
  ある値をパワーセットのメンバに設定します。演算結果は、指定したItemsortパラメータが反映されたPowersetパラメータのコピーです。既にpowersetのメンバになっている値を指定した場合は、何も変更されません。
  incl (3, empty)は、要素に3のみを持ったセットを返し、
  incl (3, v0_1_2)は要素0、1、2および3を持ったセットを返します。

- incl(2つ目の演算子):
  IBM Rationalが拡張した演算子で、標準のincl演算子を使いやすくしたものです。inclの演算結果で、(第2パラメータの)Powerset変数を新しいコンポーネント値に更新します。
  task incl(3, v0_1_2);は、
  task v0_1_2 := incl(3, v0_1_2);と同じ意味です。

- del:
  パワーセットからメンバを削除します。演算結果は、指定したItemsortパラメータが削除されたPowersetパラメータのコピーです。Powersetのメンバではない値を削除する場合は、何も変更されません。
  del (0, v0_1_2)は要素1、2を持ったセットを返します。
  del (30, v0_1_2) = v0_1_2

- del (second operator):
  IBM Rationalが拡張した演算子で、標準のdel演算子を使いやすくしたものです。Powerset変数からコンポーネント値を削除した値で、Powerset変数を更新します。
  task del(3, v0_1_2);は、
  task v0_1_2 := del(3, v0_1_2);と同じ意味です。

- length:
  パワーセット内の要素数を返します。
  length (v0_1_2) = 3, length (empty) = 0

- `take`(パラメータ数1):
  パワーセットの要素の1つを返しますが、どの要素を返すかは不確定です。`take (v0_1_2)`は0、1または2を返します。3つのどの値を返すかを指定することはできません。

- `take`(パラメータ数2):
  パワーセットの要素には、1からlength()までの番号が暗黙に付けられています。このtake演算子はIBM Rational専用であり、第2パラメータで指定した番号の要素を返します。この演算子は、図27に示されているように、セット内のすべての要素を参照するために、ループ制御で使用できます。

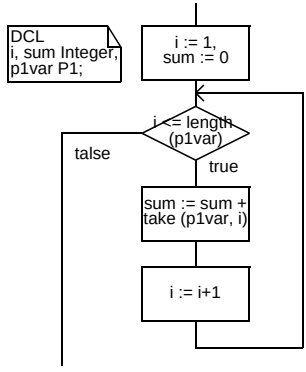


図27: Powersetの全要素の合計を算出

- `<`:
  A<Bは、AがBの真の部分集合であることを表します。
  `incl (2, empty) < v0_1_2 = true,`
  `incl (30, empty) < v0_1_2 = false`

- `>`:
  A>Bは、BがAの真の部分集合であることを表します。

- `<=`:
  A<=Bは、AがBの部分集合であることを表します。

- `>=`:
  A>=Bは、BがAの部分集合であることを表します。

- `and`:
  パワーセットの共通部分、つまり両方のパラメータに共通するメンバを要素に持つパワーセットを返します。
  `incl (2, incl (4, empty)) and v0_1_2`は、要素として2のみを持ったセットを返します。
- `or`:
  パラメータの和集合、つまり両方のパラメータのメンバを要素に持つパワーセットを返します。
  `incl (2, incl (4, empty)) or v0_1_2`は、要素として0、1、2、4を持ったセットを返します。
- (. .):
  IBM Rationalが拡張した式で、演算子の形はしていませんが、すべてのパワーセットにある暗黙のmake演算子を適用することを表します。make演算子はItemsort値のシーケンスをパラメータとし、各値を要素として持つPowersetを返します。たとえば
  `v0_1_2 := (. 1, 2, 3 .)`とすると、1、2、3を要素として持つPowersetが返されます。

PowersetはBag演算子に似ていますが、通常はPowersetの使用をお勧めします。詳細については、79ページの「Bag」を参照してください。

Bag

Z.105固有のジェネレータである`Bag`の機能は、Powersetとほとんど同じです。唯一の違いは、Bagが同じ値を複数個持つことができる点です。Powersetでは個々の値は、セットのメンバか非メンバのどちらかになります。Bagのインスタンスは`Powerset`のインスタンスと同じの演算子と、`empty`リテラルを持ち、同様の振る舞いをします。詳細は、76ページの「Powerset」を参照してください。

Bagには、他に以下の演算子が定義されています。

```
makebag : Itemsort      -> Bag
```

- `makebag`:
  Itemsort値を取り、この値を持つlength=1のBag値を返します。

値のインスタンス数に重要な意味がある場合以外は、Bagの代わりにPowersetを使用してください。Powersetは、Z.100勧告で定義されているため移植性が高くなります。Bagは、ASN.1の`SET OF`構成体をサポートするために定義されたデータ型の1つです。

Ref、Own、Oref、Carray
IBM Rationalが拡張したジェネレータです。Ref、Own、OrefはC言語のポインタ、CarrayはC言語の配列を扱うために使います。

OwnおよびOrefについては、第3章「SDLの拡張表現の使用」128ページの「OwnとORefジェネレータ」、RefおよびCarrayについては108ページの「C言語専用のctypesパッケージ」の「package ctypes」を参照してください。「package ctypes」には、Cの単純型に対応するSDLの型についても記述してあります。

リテラル

リテラル、すなわち名前付きの値は、newtypeに定義することができます。

例21: struct newtypeのリテラル

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    literals Origo, One;
endnewtype;
```

この構造体には、`Origo`と`One`という2つの名前付きの値（リテラル）があります。SDLで、リテラルによって値を定義する方法は、アクシオムを使用する方法のみです。アクシオムの定義は、newtype内で1行記述するだけです。ここでは、詳細について説明しません。リテラル値を使用する方法としては、その他にSDL to Cコンパイラを利用する方法があります。『User´s Manual』の第56章、「Cadvanced/Cbasic SDL to Cコンパイラ」を参照してください。

SDLアクション内でのリテラルの表現方法は、通常の式と同じです。

例22: リテラルの使用

```
dcl C1 Coordinates;

task C1 := Origo;
decision C1 /= One;
...
```

上述の例に示したリテラルと、列挙データ型64ページの「列挙ソート」の記述に現れるリテラルでは、解釈が異なるので注意してください。列挙データ型の場合、リテラルごとに別々の値が与えられます。また、その型の変数は、リテラルで与えられた以外の値を取ることができません。これに対して上述の構造体の例では、

型や取りうる値は構造体の定義で決まります。リテラルはある既存の値に名前を付けているだけのことです。

この説明で分かりにくければ、列挙データ型に使うか、あるいは上述の構造体の例のように、パラメータなしの演算子(IBM Rationalによる拡張)として使うという、2つの場合のみ考えると決めておけばよいでしょう。

演算子

*演算子*は、リテラルと同じ方法でnewtypeに追加できます。

例23: newtype構造体の演算子

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    operators
    "+" : Coordinates, Coordinates -> Coordinates;
    length : Coordinates -> Real;
endnewtype;
```

IBM Rationalでは演算子についてもいくつか拡張して、より柔軟な記述や、効率的な実装を可能にしました。次のような点が拡張されています:

- 入出力兼用パラメータ
- パラメータを取らない演算子
- 結果を返さない演算子

例24演算子

```
operators
  op1 : in/out Coordinates;
  op2 : -> Coordinates;
  op3 : ;
```

この例で、op1は入出力兼用パラメータを取り、結果は返しません。op2にはパラメータがなく、戻り値の型はCoordinatesです。op3にはパラメータも戻り値もありません。

演算子の振る舞いは、アクシオム内（リテラル値と同様）か、演算子ダイアグラム内に定義できます。演算子ダイアグラムは、状態を持たない返値プロシージャとほとんど同じ機能になります。また、SDL to Cコンパイラでは、ターゲット言語で記述されたコードを定義することもできます。演算子の実装をダイアグラムで記述する代わりに、テキストで記述することもできます。計算処理がほとんどで、プロセス制御やプロセス間通信が不要な場合に適切な方法です。第3章「SDLの拡張表現の使用」138ページの「複合ステートメント」に解説しているアルゴリズム拡張も併用すれば、さらに洗練された記述が可能になります。

例25: 演算子の実装

```
newtype Coordinates struct
    x integer;
    y integer;
  adding
    operators
    "+" : Coordinates, Coordinates -> Coordinates;
  operator "+" fpar a, b Coordinates
               returns Coordinates
  {
    dcl result Coordinates;
    result!x := a!x + b!x;
    result!y := a!y + b!y;
    return result;
  }
endnewtype;
```

SDL to Cコンパイラでは、ターゲット言語で記述されたコードをインクルードすることもできます。ただし、SDL to Cコンパイラが演算子をCに変換する方法について、詳しく知っていなければならないという問題があります。

デフォルト値

newtypeやシンタイプでは、データ型内のすべての変数に与えるデフォルト値を指定するdefault句を挿入できます。

例26: newtype構造体のデフォルト値

```
newtype Coordinates struct
    x integer;
    y integer;
  default (. 0, 0 .);
endnewtype;
```

`Coordinates`ソートのすべての変数には、変数宣言で明示的にデフォルト値が与えられている場合を除いて、初期値`(. 0, 0 .)`が与えられます。

例27: 変数宣言の明示的なデフォルト値

```
dcl
  C1 Coordinates := (. 1, 1 .),
  C2 Coordinates;
```

上記のC1には、起動時に代入されるデフォルト値が明示的に定義されています。C2には、newtypeで指定されているデフォルト値が与えられます。

他の型を継承するnewtypeは、元の型のデフォルト値は継承しません。

ジェネレータ

SDLでは、定義済みのジェネレータであるArray、String、PowersetおよびBagと同じような機能を持ったジェネレータを定義できます。ただし、これはかなり難しい作業であり、コード ジェネレータのサポートに制限がある場合もあるため、専門的な知識のない方がジェネレータを定義することはお勧めできません。

SDL to Cコンパイラで使用できるユーザー定義ジェネレータについての詳細は、『User´s Manual』の第56章「Cadvanced/Cbasic SDL to Cコンパイラ」の2698ページ、「ジェネレータ」を参照してください。

SDLでのC/C++言語の使用

はじめに

SDL仕様からCまたはC++の宣言にアクセスできるようにするために、C/C++からSDLへの翻訳規則が作成されました。この翻訳規則は、C/C++構文をSDLで記述する方法について指定しています。この翻訳規則は、SDL Suiteの CPP 2 SDLツールに実装されています。CPP2SDLでは、CとC++の宣言の両方に対する翻訳をサポートしています。

CPP2SDLを使用している場合、SDL内のC/C++宣言および定義にアクセスすることができます。図28では、CPP2SDLがC/C++ヘッダ ファイルのセットおよびインポート仕様を入力内容として処理する方法を示しています。インポート仕様は、CPP2SDLがコマンドラインから実行されたときのみのオプションになります。オーガナイザからユーティリティを使用すると、インポート仕様はデフォルトの構成を使用して作成されます。インポート仕様にはCPP2SDLオプションが含まれ、ヘッダ ファイル内のどの宣言を変換するかを指定することもできます。その後CPP2SDLは、ヘッダ ファイル内のC/C++宣言をSDL宣言に翻訳します。この結果翻訳されたSDL宣言は、生成されたSDL/PRファイルに保存されます。詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の752ページ、「Introduction」を参照してください。

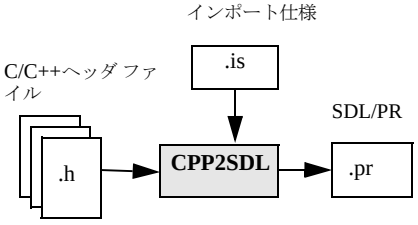


図28: CPP2SDL入出力

ワークフロー

CPP2SDLを使用する場合に含まれる通常のワークフローは、Access Controlシステムに基づく例で説明することができます。この例は、`IBM¥Rational¥SDL_TTCN_Suite6.3J¥sdt¥examples¥cpp_access`にあります。この例は、現時点ではWindowsのみで動作します。ただし、説明に含まれる原理はすべてのプラットフォームで同一です。

Access Controlシステムは、ビルディングへのアクセスを制御します。ビルディングには、ディスプレイ、カード リーダー、およびキーパッドから成るユーザー端末があります。このビルディングへアクセスするには、有効なカードを挿入し、正しい4桁のコードを入力する必要があります。

このバージョンのAccess Controlシステムでは、カードと有効なコードに関する情報は外部データベースに格納されます。データベースには、ODBC[1] によってアクセスできます。ODBCとは、別の種類のデータベースからデータにアクセスするときに一般的に使用されるC/C++ APIです。

この例の目的は、C/C++ Accessのツールを使用してSDLからC/C++ APIへアクセスする方法を示すことです。この例にはC/C++ Accessの使用法に関連する最も重要な問題が含まれ、より高度な試みの基礎として使用することができます。

下記の例は、オーガナイザ内からCPP2SDLを利用する方法の概略です。ここでは以下の段階について説明しています。

- PRシンボルがCentralプロセス ダイアグラムに追加されます。
- PRシンボルがインポート仕様になるよう変更されます。この処理を行うには、PRシンボルをダブルクリックしてドキュメント タイプを*C++インポート仕様*に設定します。
- TRANSLATEセクションがインポート仕様に追加されます。このセクションには、アクセスする必要があるすべてのC/C++宣言の名前を列挙します。
- インポート仕様がファイルに保存されます。これにより、インポート仕様シンボルは自動的にこのファイルに接続されます。
- ［CPP2SDLオプション］ダイアログ ボックスを使用して、インポート仕様の各種オプションを設定します。
- 最後に、変換するヘッダ ファイルを追加します。

1. ODBCはWindowsの事実上の標準ですが、Windows以外のプラットフォームにも実装されています。

編集

SDLからC/C++ APIにアクセスするには、まず、APIのC/C++宣言が使用されるSDL仕様内の場所にPRシンボルを挿入します。通常、PRシンボルはSDL/PRファイルが含まれていることを表します。C/C++ Accessでは、この機能はCPP2SDLによって生成されるSDL/PRファイルを挿入するために使用されます。

Access Controlの例では、ODBCという名前のPRシンボルをプロセスCentralに挿入します。このプロセスからODBC APIへは排他的にアクセスされます。これによって利用できる範囲を最も限られたものにします。

通常、インポート仕様は、インポート仕様によりインポートされた宣言が使用される最上位に配置する必要があります。ただし、C/C++変数をインポートする場合は、外部SDL変数を宣言できる範囲内にインポート仕様を配置する必要があります。

> メモ：
>
> 外部変数は、システムまたはブロック レベルでは宣言できません。これらの変数は、プロセス、プロシージャ、サービスあるいは演算子ダイアグラムでのみ宣言できます。

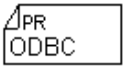


図29: SDLエディタ内のPRシンボル

SDLエディタに追加されたPRシンボルは、まずオーガナイザ内に未接続の参照として表示されます。図30を参照してください。

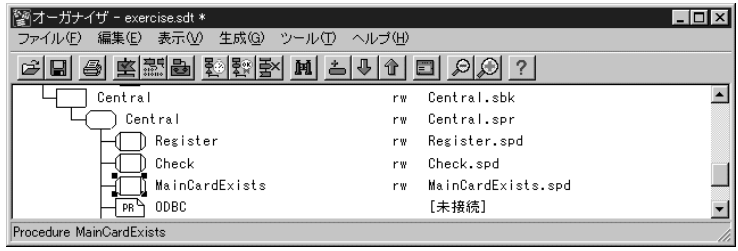


図30: オーガナイザ内の未接続のPRシンボル

デフォルトでは、オーガナイザは未接続のPRシンボルが通常のユーザー定義SDL/PRファイルに接続されると想定しています。この場合は、ユーザー定義SDL/PRファイルに接続しません。SDLエディタまたはオーガナイザでPRシンボルをダブルクリックすると、[ドキュメントの編集] ダイアログ ボックスが開きます。87ページの図31を参照してください。 ドキュメント タイプを*SDL/PR*から*C++インポート仕様*に変更する場合は、C++ヘッダ ファイルのセットからSDL/PRファイルが生成されるように指定します。

> メモ：
>
> 通常のPRシンボルはユーザー定義のSDL/PRファイルに接続されますが、インポート仕様シンボルは生成されたSDL/PRファイルに接続されます。

図31: PRシンボルのタイプを編集してC++インポート仕様にする

新しいインポート仕様を作成するために、[エディタで表示] チェック ボックスはオンのままにしておきます。使用するインポート仕様が既にある場合は、接続方法は2とおりあります。最初の方法は、すべてのチェック ボックスをオフにし、オーガナイザの [接続] コマンドを使用して既存のインポート仕様ファイルに接続します。2番目の方法は、[既存のファイルをコピー] チェック ボックスをオンにして、既存のインポート仕様を参照するか、既存のインポート仕様のパスを入力します。この方法を使用する場合は、テキスト エディタでファイルを表示し、そのファイルに接続するために保存する必要があります。

インポート仕様はテキスト エディタを使用して手動で編集できます。ただし、インポート仕様を空のままにし、後でオーガナイザ内からCPP2SDLオプションを設定することができます。これにより、CPP2SDLに対する別のオプションが格納される、CPP2SDLOPTIONSという名前のセクションが追加されます。ほとんどの場合、インポート仕様にはTRANSLATEセクションが含まれます。このセクションには、SDL内のアクセスしたいすべての宣言名の一覧が含まれます。詳細については、102ページの「インポート仕様」を参照してください。

ここでは、SDLからアクセスする必要があるすべてのODBC関数およびデータ型の名前を含むTRANSLATEセクションを追加します。図32を参照してください。

```
TRANSLATE {

  SQLHENV
  SQLHDBC
  SQLHSTMT
  SQLRETURN
  SQLCHAR
  SQLINTEGER
  SQLSMALLINT
  SQLPOINTER

  SQLAllocHandle
  SQLSetEnvAttr
  SQLSetConnectAttr
  SQLConnect
  SQLBindCol
  SQLExecDirect
  SQLFetch
  SQLCloseCursor
  SQLFreeHandle
  SQLDisconnect
  SQLGetDiagRec

  unsigned char.[6]
  unsigned char.[64]
  unsigned char.[256]

  char.[256]

  strcpy
  strcat
}
```

図32: ODBCインポート仕様のTRANSLATE セクション

インポート仕様がファイル（ODBC.is）に保存されると、オーガナイザは自動的にインポート仕様シンボルをそのファイルに接続します。

図33では接続されたインポート仕様シンボルを示しています。

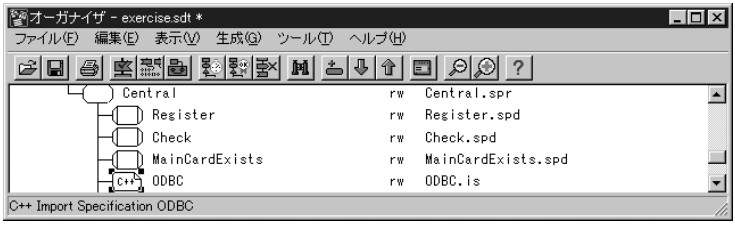


図33: オーガナイザ内の接続されたインポート仕様シンボル

次に、インポート仕様で指定されたC/C++宣言の翻訳に必要な適切なオプションを設定します。設定は［CPP2SDLオプション］ダイアログ ボックス（図34参照）を使用して行うのが最適です。オーガナイザでインポート仕様シンボルを右クリックすると、このダイアログ ボックスが開きます。CPP2SDLオプションの詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の751ページ、「The CPP2SDL Tool」を参照してください。

図34: ［CPP2SDL オプション］ダイアログ ボックス

以下のオプションを指定することができます。

- ［言語］

  入力言語を指定する言語オプションです。つまり、C宣言とC++宣言のいずれを変換するかを指定します。入力言語としてC言語を選択した場合、CPP2SDLは入力ヘッダ ファイルにはC++言語固有の構文がないものと見なします。

  このオプションは、インポート仕様がCインポート仕様またはC++インポート仕様のいずれであるかを決定します。使用するインポート仕様の種類を選択する場所については、87ページの図31を参照してください。

- ［固有表現］

  これらのチェック ボックスでは、どの種類のC/C++言語の固有表現がCPP2SDLでサポートされるかを指定することができます。すべてのチェック ボックスがオフになっている場合は、ANSI C/C++言語の固有表現がサポートされます。

  例ではMicrosoft Foundation ClassesのODBC実装を使用しているため、Microsoftの固有表現のサポートが必要になります。

- ［ランタイム データ型情報］

  このチェック ボックスがオンになっている場合、ランタイム データ型情報（RTTI）が想定され、動的キャストがサポートされます。

- ［オブジェクトスライスの許可］
  生成されるSDLキャスト演算子がC++オブジェクトのスライスをサポートする場合、このチェック ボックスをオンにします。
- ［入力でのSDLソートの識別］
  このチェック ボックスがオンになっている場合、入力でSDLソートが識別されます。
- ［プリプロセッサ］
  入力の前処理に使用するプリプロセッサをここで設定できます。プリプロセッサが設定されていない場合、CPP2SDLはWindowsではMicrosoft Visual C/C++コンパイラ（`cl`）を、UNIXでは標準のC/C++プリプロセッサ（`cpp`）を使用します。

> メモ：
>
> 通常は、一般的なプリプロセッサではなくコンパイラを使用して入力C/C++ヘッダを前処理することをお勧めします。これは、コンパイラが複数の有用なプリプロセッサ定義を設定できるためです。

- ［プリプロセッサ オプション］
  プリプロセッサ オプションはこのフィールドで設定できます。
- ［ポインタ］、［配列］、［テンプレート］、［キーワード］、［未完成］、［アンダースコア］
  これらのフィールドは、SDL翻訳でC/C++名を変更する必要があるときに使用する接頭辞と接尾辞を指定します。

- ［基本的なデータ型に対するSDL表現を生成する］
  基本的なC/C++のデータ型のSDL表現が翻訳に含まれる場合、このチェック ボックスをオンにします。これらのSDL表現はSDL/PRファイルで定義されます。詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の834ページ、「SDL Library for Fundamental C/C++ Types」を参照してください。

> メモ：
>
> SDLデータ型表現のオプションが複数のレベルで設定されている場合は問題が発生します。基本的なデータ型のSDL表現は、そのデータ型が使用される最上位にのみ含まれる必要があります。たとえば、システム内の2つのブロックがC/C++宣言にアクセスするためのインポート仕様を持っている場合、基本的なC/C++のデータ型のSDL表現はブロック内ではなくシステム内に挿入する必要があります。これを行うには、システム レベルに入力ヘッダを持たない空のインポート仕様を追加します。空のインポート仕様により、基本的なC/C++のデータ型のSDL表現が挿入されます。

- ［必要時にクラス ポインタ データ型のみを生成する］
  このチェック ボックスをオンにすると、CPP2SDLはクラス ポインタ データ型の生成を最適化します。

インポート仕様に対する適切なCPP2SDLオプションを設定しら、翻訳するC/C++ヘッダ ファイルを追加します。これを行うには、インポート仕様を選択し、［編集］メニューで［既存の追加］を選択します。追加されたヘッダ ファイルはオーガナイザのインポート仕様シンボルの下に表示されます。図35を参照してください。

図35: インポート仕様へのヘッダ ファイルの追加

インポート仕様には任意の数のヘッダ ファイルを追加できます。ヘッダ ファイルはすべてインポート仕様で指定されたオプションを使用して処理されます。Access Controlの例では、ヘッダ ファイルが1つだけ追加されています（`includes.h`）。

ヘッダ ファイルの内容を確認するには、オーガナイザでそのシンボルをダブルクリックします。するとヘッダ ファイルがテキスト エディタで開かれ、その内容が表示されます。`includes.h`ヘッダの内容を表示すると、実際にほかのヘッダ ファイルが複数挿入されているのを確認できます。インポート仕様の下の目的のヘッダ ファイルを直接追加する代わりに`includes.h`のようなラッパー ヘッダを使用するのは、これらのファイルのパスがハードコードされないようにするためです。`#include <`ファイル名`>`ステートメントを使用し、さらにMicrosoft Visual C++コンパイラでファイルを前処理すると、これらのファイルの場所がコンパイル時に通知されます。

ここで、ここまで例を使用して行ってきたことをまとめてみます。まず、PRシンボルを追加してSDLシステムを編集しました。次に、PRシンボルをインポート仕様に変更し、必要な宣言を`TRANSLATE`の一覧に追加しました。さらに、システムに接続するためにインポート仕様を保存し、オプション ダイアログ ボックスを使用してCPP2SDLを設定しました。最後に、ヘッダ ファイルをシステムに追加しました。

これらの処理をすべて行うと編集段階が終了します。次は、システムを分析します。

分析

CPP2SDLによって生成されるSDL宣言は、大文字と小文字が区別されるSDLとして分析する必要があります。アナライザを起動する前に、大文字と小文字の区別に関するオプションを設定してください。

- オーガナイザで［ツール］を選択し、［環境設定マネージャ］を起動します。
- ［環境設定マネージャ］でSDTシンボルをダブルクリックし、［CaseSensitive］を*オン*にします（デフォルトでは*オフ*に設定されています）。

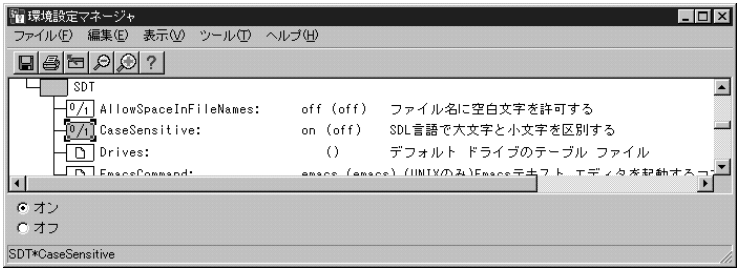


図36: 環境設定マネージャで大文字と小文字を区別する (Case Sensitive) SDL を設定する

アナライザは、C/C++インポート仕様を含むSDLシステムの分析時に3つの手順を行います。各手順を実行している間は、進捗状況を示すメッセージがオーガナイザ ログ ウィンドウに表示されます。図37を参照してください。

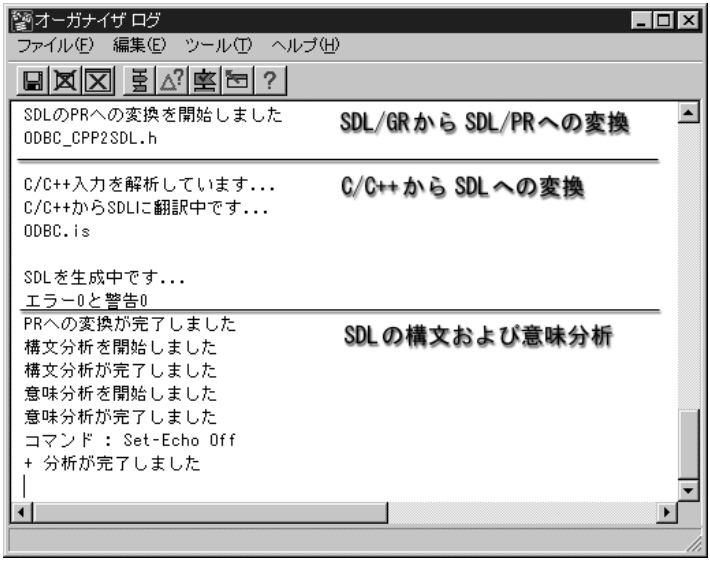


図37: 分析段階でのオーガナイザ ログ ウィンドウ

1. SDL/GRからSDL/PRへの変換
   アナライザはSDLエディタにSDL/GRからSDL/PRへの変換を行うよう要求します。つまり、すべてのグラフィカルなSDLシンボルがテキスト表現に変換されます。特に、すべてのPRシンボルは、SDL/PR内の`#include 'filename.pr'`によって表現されます。`filename.pr`は、対応するインポート仕様の接続先のファイル名です。

   したがって、例ではプロセスCentralのSDL/PR表現で`#include 'ODBC.pr'`を取得します。

2. C/C++からSDLへの変換
   この手順はシステム内のインポート仕様ごとに1度だけ実行されます。インポート仕様に関連付けられたヘッダ ファイルはCPP2SDLによって解析され分析されます。この段階で報告されるエラーは、たとえば、言語サポートの違いや不適切なプリプロセッサの設定によって発生します。このエラーが発

生した場合は、［CPP2SDLオプション］ダイアログ ボックスで、正しい言語や適切なプリプロセッサ オプションを設定します。また、ヘッダ ファイルの構文エラーや一部の意味エラーもこの段階でチェックされます。
CPP2SDLがエラーを処理する方法の詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の839ページ、「Example usage of some C/C++ functionality」を参照してください。
エラーが検出されなかった場合、CPP2SDLは翻訳の結果としてSDL/PRファイルを生成します。最後に、何らの理由で特定の宣言が翻訳されなかったことなどを通知する警告がいくつか表示されることがあります。

この例ではこの手順が完了すると、`ODBC.pr`という名のファイルを取得します。

> メモ：
>
> エラーの検出およびエラー報告に関して言えば、CPP2SDLよりC/C++コンパイラの方が優れています。したがって、SDLに変換する前にヘッダ ファイルをコンパイラで実行し、ヘッダ ファイルが意味論において正しいことを確認するよう強くお勧めします。

3. SDLの構文および意味分析
   すべてのSDL/PRコードが生成されると、SDLアナライザは通常どおり構文および意味エラーをチェックします。たとえば、環境設定マネージャで大文字と小文字が区別されるSDLが設定されていなければ、多くのエラーが報告される可能性が高くなります。図36を参照してください。エラーの一般的な原因は、基本的なデータ型のSDL表現がまったく含まれないか、SDLシステムの誤った場所に含まれているか、あるいは同じSDLスコープ エンティティに何度も挿入されていることです。

システムの完全な分析を行ったら、次にコード生成を行います。

生成

コード生成は、従来の［実装］ダイアログ ボックスか、より強力なツールである*SDL ターゲティング エキスパート*で行うことができます。CまたはC++インポート仕様を含むシステムのコードを生成する場合は、ターゲティング エキスパートの方が適しています。たとえば、翻訳されたヘッダ ファイルに属しているオブジェクト ファイルをリンクする場合は、ターゲティング エキスパートを使用すると簡単に行うことができます。近い将来［実装］ダイアログ ボックスはなくなり、ターゲティング エキスパートが使用されるようになります。

1つ以上のC++インポート仕様を含むシステムは、Cコードではなく C++コードに翻訳する必要があります。コード ジェネレータのオプションが、Cコードと C++コードのいずれが生成されるかを制御します。1つ以上のC++インポート仕様がオーガナイザ ビューに存在する場合、このオプションはアナライザによって自動的に設定されます。

ターゲティング エキスパートを起動するには、オーガナイザの［生成］メニューで［ターゲティング エキスパート］を選択します。すると、［ターゲティング エキスパート］ダイアログ ボックスが表示されます。図38を参照してください。ターゲティング エキスパートでサポートされているすべての設定およびオプションの詳細については、『User´s Manual』の第59章、「The Targeting Expert」を参照してください。

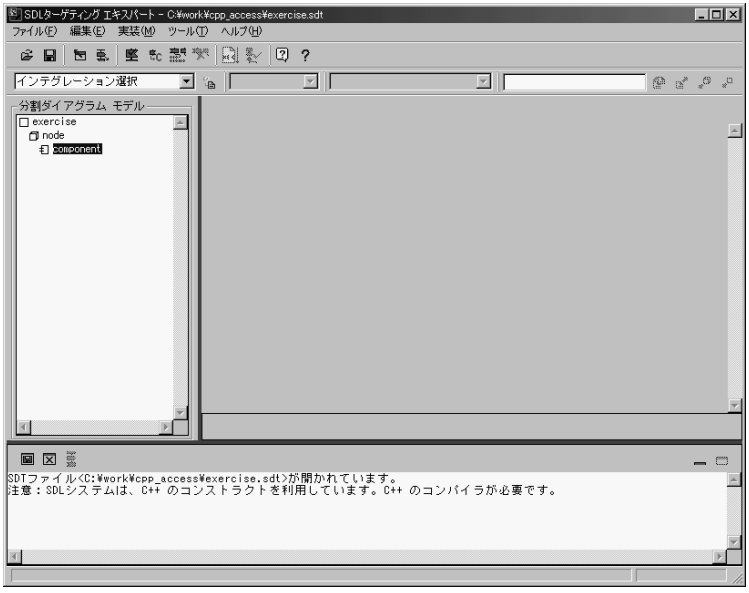


図38: SDL ターゲティング エキスパート

1つ以上のC++インポート仕様がSDLシステムに存在する場合、ターゲティング エキスパートは、生成されたコードをコンパイルするにはC++コンパイラが必要だという警告を発します（図38参照）。次に［コンポーネント］を右クリックし、［シミュレーション］-［シミュレーション］の順に選択します。［コンパイラ/リンカ/実装］アイコンをクリックし、［コンパイラ］タブの下にコンパイラ実行可

能ファイルを配置すると、コンパイラを設定できます。また、このタブではコンパイラ オプションおよびプリプロセッサの設定を指定することもできます。

> メモ：
>
> ［CPP2SDLオプション］ダイアログ ボックスで行ったプリプロセッサの設定がターゲティング エキスパートで行った設定と一致していることを確認してください。

Access Controlの例では、C++ Microsoft Simulation カーネルを使用できます。また、生成されたコードはMicrosoft Visual C++ コンパイラによってコンパイルされます。

リンク エラーによる負荷を防ぐには、いくつかの必要なMicrosoftライブラリ（Odbc32.lib ライブラリなど）をリンクしておく必要があります。これを行う場合は、図39に示した［リンカ］タブを使用します。ファイルの一覧にファイルを追加し、保存するだけで済みます。

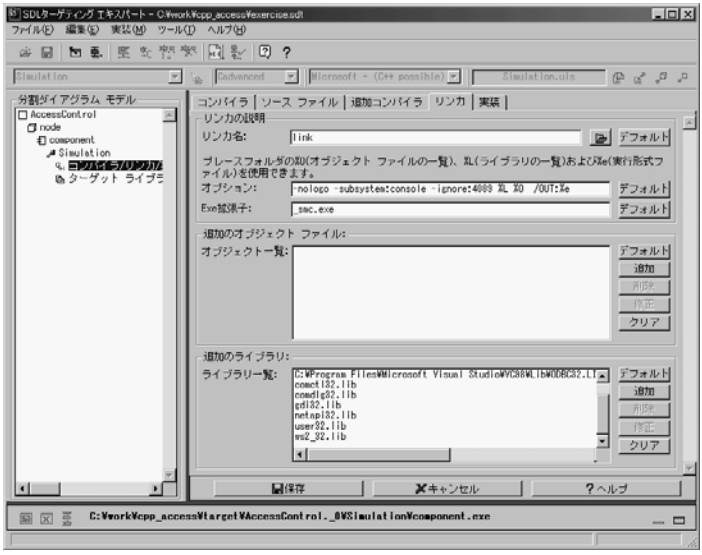


図39: ターゲティング エキスパートでのライブラリの追加

これで、コード生成のための準備が完了しました。［実装］ボタンまたは［完全な実装］ボタンをクリックすると、ターゲティング エキスパートによりアナライザがSDLシステムを分析し（93ページの「分析」参照）、続いてCまたはC++コード ジェネレータが起動します。最後に、生成されたコードが指定どおりにコンパイルおよびリンクされ、シミュレータ実行可能ファイルが作成されます。図40ではこの作業が完了したときのターゲティング エキスパートの状態を示しています。

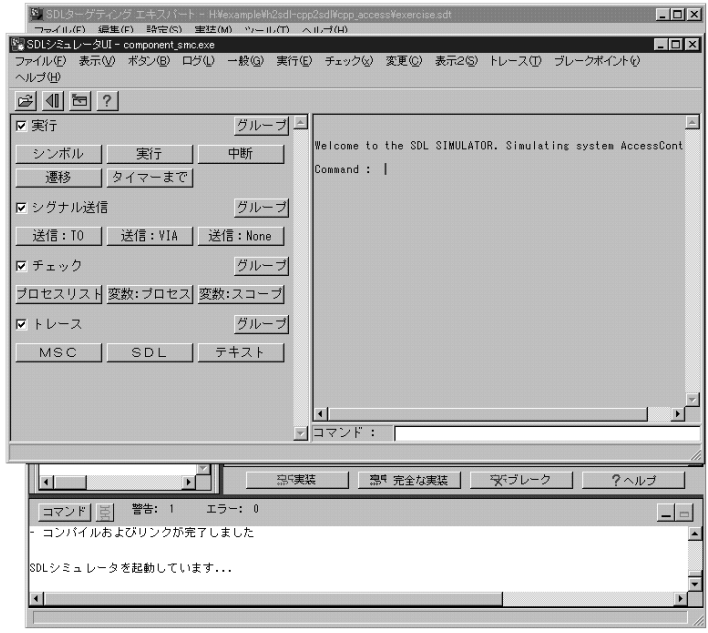


図40: ターゲティング エキスパートからのシミュレータの生成

シミュレーション

通常は、CまたはC++宣言を使用していても、SDLレベルでシステムをシミュレートおよびデバッグすることは可能です。標準のSDLシミュレータはこの処理に使用できます。

シミュレータは、ターゲティング エキスパートからの実装時に自動的に起動します。実装時以外にシステムのシミュレーションを開始するには、オーガナイザまたはターゲティング エキスパートの［ツール］メニューでSDLを選択します。続いて［シミュレータUI］を選択すると、SDLシミュレータ ユーザー インターフェイスが起動します。上の手順で生成されたシミュレータ実行可能ファイルをロードするには、SDLシミュレータUIで［ファイル］-［開く］の順に選択します。

Access Controlの例では、カスタマイズされた2つのボタンがシミュレータUIで使用できます。これらのボタンをロードするには［ボタン］-［ロード］の順に選択します。［GUI］ボタンをクリックすると、Access ControlシステムのGUIが起動し、GUIとの相互作用によってシステムを開始または検査するまで待機しま

す。[GUI+MSC] ボタンをクリックすると、GUIが起動し、さらにMSCトレースを生成します。図41はAccess ControlシステムのMSCトレースの例を示しています。

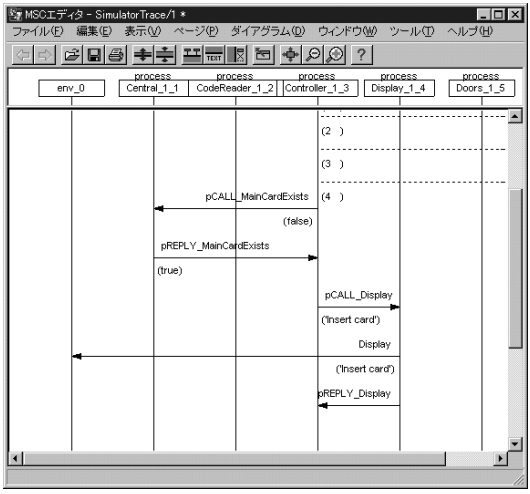


図41: Access ControlシステムのMSCトレース

シミュレータはC++クラスをC言語の構造体として扱いますが、クラスのコンストラクタを起動する可能性もあります。たとえば、SDLでインスタンス化されたC++クラスの値がシミュレータから変更されると、以下の手順が実行されます。

- シミュレータは、使用できるコンストラクタの一覧を示すダイアログ ボックスを表示します。たとえば、以下のようになります。

  ```
  0 /* No constructor */
  1 /* C() */
  ```

  また、ユーザー定義コンストラクタを持つクラスの場合は、以下のようになります。

  ```
  0 /* No constructor */
  2 /* C(int) */
  ```

  起動されるコンストラクタがある場合は、その数を入力します。

- コンストラクタが選択された場合、シミュレータはその実引数を要求します。

- 最後にシミュレータにより、SDLまたはASN.1構文のいずれかを使用してパブリック メンバ変数を明示的に設定することができます。たとえば以下のようになります。

```
(. 1, true, 'x' .)              SDL syntax
{mv1 1, mv2 true, mv3 'x'}       ASN.1 syntax
```

ASN.1構文はメンバ変数の名前を含んでいるため、より柔軟性があります。

シミュレータから、たとえばクラス タイプのパラメータを含む信号を送信することによってC++クラスをインスタンス化する手順は、上記の手順と類似しています。

Access Controlの例の要約

上に示したAccess Controlの例の手順は、C/C++ Accessを使用している場合の代表的なワークフローです。

- SDL仕様はPRシンボルを追加することによって編集され、PRシンボルは変更されてインポート仕様になります。C/C++ヘッダ ファイルは各インポート仕様の下に追加され、［CPP2SDLオプション］ダイアログ ボックスによって適切な翻訳オプションが設定されます。また、翻訳される宣言名を列挙しているインポート仕様にTRANSLATEセクションを追加することもできます。
- SDL仕様は大文字と小文字が区別されるSDLとして分析されます。C/C++ヘッダのエラーはCPP2SDL、SDL仕様のエラーはSDLアナライザによりそれぞれ検出されます。
- CまたはC++コードは［実装］ダイアログ ボックスまたはターゲティング エキスパートの使用により生成されますが、ターゲティング エキスパートの方が適しています。生成されたコードは追加のオブジェクト ファイルとともにコンパイルされリンクされます。
- SDL仕様はシミュレータUIを使用してシミュレーションできます。

図42は、Access Controlシステムの完了時にこのシステムがどのような状態になっているかを示すオーガナイザ ビューです。

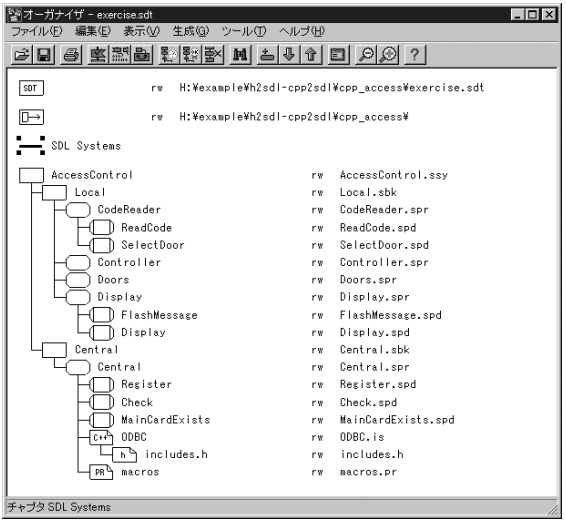


図42: Access Control システムのオーガナイザ ビュー

図でわかるように、Centralプロセスにはインポート仕様に変更されていないPRシンボルが1つあります。その代わり、このシンボルは一般的なSDL/PRファイルである`macro.pr`に接続されています。このファイルにはODBC APIの呼び出しで必要なC/C++マクロを表す外部SDLシノニムが含まれています。これらのシノニムのソートは`ODBC.is`インポート仕様によりインポートされます。#CODE演算子に基づくC/C++マクロにアクセスするためのこれ以外の方法については、104ページの「SDLからのC/C++マクロへのアクセス」を参照してください。

インポート仕様

インポート仕様は、簡単なC/C++形式の構文で記述されたテキストファイルです。インポート仕様を使用すると、入力ヘッダ ファイル内のどの宣言にアクセスするか正確に指定することができます。宣言の指定されたサブセットがCPP2SDLにより翻訳されます。また、インポート仕様により、クラスや関数テンプレートなどにアクセスすることができます。インポート仕様の詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の766ページ、「Import Specifications」を参照してください。

以下の例は、SDLで識別子a_int、i_arrおよびfuncが使用できる簡単なインポート仕様を示しています。

例28: 簡単なインポート仕様

```
TRANSLATE {
        a_int
        i_arr
        func
}
```

インポート仕様の識別子が別の宣言に依存する宣言を参照している場合、CPP2SDLはこれらの宣言もすべて同様に翻訳します。

これ以外にもインポート仕様で使用できる以下のような高度な構文があります。

- データ型宣言子
- Ellipsis関数のプロトタイプ

これらの構文の詳細については、『User´s Manual』の第14章「The CPP2SDL Tool」の768ページ、「Advanced Import Specifications」を参照してください。

テンプレート

CPP2SDLツールの使用により、テンプレート宣言のインスタンス化がサポートされます。

C++テンプレートをインスタンス化できるようにするには、CPP2SDLはそのテンプレート実引数に関する情報を必要とします。この情報はインポート仕様に記述されています。

C++テンプレート宣言はそれ自身ではSDLに翻訳されません。その代わりテンプレートのインスタンスがSDLに割り当てられます。

CPP2SDLで完全にはサポートされていないC/C++構文へのアクセス

SDLからのC/C++マクロへのアクセス

マクロは条件付きコンパイル用に使用されますが、以下のような目的にも使用できます。

- 定数の定義：`#define PI 3.1415`
- データ型の定義：`#define BYTE char`
- 関数の定義：`#define max(a,b) a>b?a:b`

マクロはC言語やC++言語の一部ではありません。したがってSDLには翻訳されません。その代わり、CPP2SDLが翻訳を行う前にプリプロセッサがすべてのマクロを展開します。

SDLからマクロ定数にアクセスできるようにする場合は、暗黙の#CODE演算子やSYNONYMを使用できます。以下の例をご覧ください。

例29: C++マクロとして定義された定数

```
C++:
#define PI 3.1415;

#CODE を使用する SDL:
dcl a double;

task a := #CODE('PI');

SYNONYM を使用する SDL:
SYNONYM PI double = EXTERNAL 'C++';

dcl a double;

task a := PI;
```

データ型または関数のマクロ定義にアクセスできるようにする場合は、マクロ`__CPP2SDL__`を使用できます。`__CPP2SDL__`マクロはCPP2SDLの実行時に定義されますが、それ以外では定義されません。このマクロは特別なヘッダ ファイル（下記の例の`x.h`）で使用されます。このヘッダ ファイルはCPP2SDLにより翻訳されるヘッダ ファイルのグループに含める必要があります。

以下の例は、C/C++ヘッダを変更してデータ型と関数のマクロ定義をSDLで使用できるようにするための`__CPP2SDL__`マクロの利用方法を示しています。

例30: C++ヘッダでのマクロ“データ型”

```
#define BYTE char
```

上記のC++フラグメントでは、マクロBYTEがデータ型であるかのように使用されています。プリプロセッサはすべてのBYTEを解決しようとし、SDLではBYTEを使用できないという結果になります。これを回避するために、BYTEの定義を以下のように変更することができます。

```
x.h:
#ifndef __CPP2SDL__
#ifdef BYTE
#undef BYTE
#endif
#define BYTE char
#else
typedef char BYTE;
#endif
```

C++からSDLへの翻訳時に__CPP2SDL__が定義されるため、マクロBYTEはSDLでデータ型として使用できるようになります。生成されたC++コードでは__CPP2SDL__は未定義なのでBYTEはマクロになります。

例31: C++ヘッダでのマクロ“関数”

```
#define max(a,b) a>b?a:b
```

maxをマクロとして定義すると、関数であるかのようにmaxを使用することができます。マクロmaxは>が定義されているすべてのデータ型で使用できます。以下の定義により、SDLでmaxをchar型およびint型で使用できるようになります。

```
x.h:
#ifndef __CPP2SDL__
#ifdef max
#undef max
#endif
#define max(a,b) a>b?a:b
#else
int max(int a,int b);
char max(char a, char b);
#endif
```

上記の定義では、__CPP2SDL__が定義されているため、maxはSDLシステムにより演算子と見なされます。SDLから生成されたC++コードがコンパイルされると、マクロ__CPP2SDL__が未定義になるため、C++プリプロセッサは“関数呼び出し”をmaxとして解決します。

関数ポインタ

関数ポインタはSDL内の型なしポインタptr_void(void*)に割り当てられます。これにより、関数ポインタをSDL内で表現できるようになります。ただし、このSDL表現を処理することはできません。たとえば、ポインタが示す関数を呼び出したり、関数ポインタをSDL演算子のアドレスに割り当てたりするには、以下の例に示すとおりに作業を行う必要があります。

例32: 関数ポインタの使用 ——————————————————————

```
C++:
int func1(int i, int j);
int con_sum(int a, int b, int (*F)(int,int));

インポート仕様 :
TRANSLATE {
func1
con_sum
}

SDL:
NEWTYPE global_namespace /*#NOTYPE*/
  OPERATORS
    con_sum : int, int, ptr_void -> int;
    func1 : int, int -> int;
ENDNEWTYPE global_namespace; EXTERNAL 'C++';

代替の #CODE を使用した SDL 1:
dcl
   sum int,
   pfunc ptr_void;

task {
   pfunc := #CODE('(void*) &func1');
   sum := con_sum(1,4,#CODE('(int (*)(int,int))
#(pfunc)'));
};

代替の #CODE を使用した SDL 2:
dcl
   sum int;

task {
   sum := con_sum(1,4,#CODE('&func1'));
};
```

——————————————————————————————————

サポートされないオーバーロード演算子

C++およびSDLのいずれにおいても、定義済みの演算子が無視される可能性があります。下記の表に、CPP2SDLがサポートするオーバーロードC++演算子を一覧表示しています。

C++演算子	説明	SDL演算子
+	(バイナリ)加算	+
-	(バイナリ)減算	-
*	(バイナリ)乗算	*
*	(単項接頭辞) デリファレンス	*>
/	(バイナリ)除算	/
%	(バイナリ)剰余	rem
!	(単項接頭辞)論理否定	not
<	(バイナリ)より小さい	<
>	(バイナリ)より大きい	>
<<	(バイナリ)左シフト	<
>>	(バイナリ)右シフト	>
==	(バイナリ)等しい	=
!=	(バイナリ)等しくない	/=
<=	(バイナリ)以下	<=
>=	(バイナリ)以上	>=
&&	(バイナリ)論理積	and
||	(バイナリ)論理和	or

C++のシフト演算子（<<、>>）とより小さい/より大きいを表す比較演算子（<、>）はいずれもSDLの"<"と">"に割り当てられます。この割り当ては、オーバーロードがSDLの"<"および">"または"<<"および">>"のいずれかでサポートされるということです。これらの演算子の組み合わせがいずれもオーバーロードされている場合、CPP2SDLは警告を発し、SDLで表すのに前者の組み合わせを選択します。

CPP2SDLではサポートされないオーバーロード演算子は、演算子#CODEを使用して処理することができます。

C言語専用のctypesパッケージ

IBM Rationalは、C言語に対応するデータ型とジェネレータが定義された`ctypes`という特別のパッケージを提供しています。`ctypes`についての詳しい情報は、『User´s Manual』の第62章、「The ADT Library」を参照してください。`ctypes`パッケージは、以下の場合に使用します。

- SDLでポインタを使用する場合
- 対応するSDLのソートが存在しない特殊なC言語のデータ型に一致するデータ型を必要とする場合（short intなど）
- SDLで直接Cコードのヘッダを使用する場合
  この場合は、必ず`ctypes`パッケージを使用しなければなりません。

次の表に、`ctypes`に定義されているデータ型とジェネレータ、および対応するC言語のデータ型とジェネレータを示します。

SDLのソート	対応するC言語のデータ型
`ShortInt`	`short int`
`LongInt`	`long int`
`UnsignedShortInt`	`unsigned short int`
`UnsignedInt`	`unsigned int`
`UnsignedLongInt`	`unsigned long int`
`Float`	`float`
`Charstar`	`char *`
`Voidstar`	`void *`
`Voidstarstar`	`void **`

SDLのジェネレータ	対応するC言語の宣言子
`Carray`	`C array, i.e. []`
`Ref`	`C pointer, i.e. *`

ここからは、上記データ型とジェネレータのSDLにおける使用方法について説明します。

Float型とさまざまなInt型

`ShortInt`、`LongInt`、`UnsignedShortInt`、`UnsignedInt`、`UnsignedLongInt`はすべてIntegerのシンタイプとして定義されるので、SDLの観点ではすべて同じデータ型になり、通常のInteger用の演算子をこれらのデータ型で使用できます。唯一の違いは、これらのデータ型から生成されるコードが異なる点です。また、`Float`は、Realのシンタイプとして定義されます。

Charstar、Voidstar、Voidstarstar

`Charstar`は、C言語の文字列（`char *`）を表します。Charstarは、SDLの定義済みデータ型であるCharstringとは異なります。Charstarは、`char *`を使用するC言語の関数やデータ型にアクセスする際に便利です。その他の場合は、Charstringを使用してください（51ページの「Charstring」参照）。CharstarとCharstring間の変換には後述の専用の演算子が用意されています。

`Voidstar`は、C言語の`void *`に対応します。このデータ型は、`void *`パラメータを使用するC言語の関数や、`void *`を返すC言語の関数にアクセスするときにのみ使用します。なお、この場合は、変換結果を直接別のデータ型に“割り当て”てください。

`Voidstarstar`は、C言語の`void **`に対応します。このデータ型は、113ページの「SDLでのポインタの使用」で説明する`Free`プロシージャと共に使います。また、C言語の関数へアクセスする際に、ときどきこのデータ型が必要になることがあります。

以下に、`ctypes`で利用できる変換演算子を示します。

```
cstar2cstring : Charstar    -> CharString;
cstring2cstar : CharString -> Charstar;
cstar2vstar   : Charstar    -> Voidstar;
vstar2cstar   : Voidstar    -> Charstar;
cstar2vstarstar : Charstar -> Voidstarstar;
```

以下に、これらの演算子の振る舞いを示します。

- `cstar2cstring`:
  C言語の文字列をSDLのCharstringに変換します。たとえば、Charstarデータ型の変数vがC言語の文字列`"hello world"`を含んでいる場合は、`cstar2cstring(v) = 'hello world'`になります。
- `cstring2cstar`:
  SDL CharstringをC言語の文字列に変換します。つまり`cstar2cstring`の逆の変換を行います。

- `cstar2vstar`:
  CharstarをVoidstarに変換します。この演算子は、`void *`パラメータを持ったC言語の関数を呼び出すときに有効です。
- `vstar2cstar`:
  VoidstarをCharstarに変換します。この演算子は、`void *`を返すC言語の関数の戻り値を`char *`に“変換する”場合に使用します。

Carrayジェネレータ

`ctypes`パッケージの`Carray`ジェネレータは、C言語の配列と同じ機能を持つ配列を定義するときに有用です。Carrayは、整数値とコンポーネント ソートの2つのジェネレータ パラメータを取ります。

例33: Carrayの実体化

```
newtype IntArr Carray(10, Integer)
endnewtype;
```

ここで定義されている`IntArr`データ型は、0から9までの10個のインデックスが定義された整数の配列です。このIntArrデータ型は、以下のC言語のデータ型に対応します。

```
typedef int IntArr[10];
```

Carrayのインスタンスでは、*modify!*と*extract!*の2つの演算子が利用できます。modify!は配列の要素の1つを変更するために使用し、extract!は配列の要素の1つの値を取り出すために使用します。これらの演算子の使用方法は、通常のSDL配列の使用方法と同じです。71ページの「Array」を参照してください。ただし、CArray全体の値を表す(. ... .)の表記はサポートされていません。

```
modify!  : Carray, Integer, Itemsort -> Carray;
extract! : Carray, Integer           -> Itemsort;
```

例34: SDLでのCarrayの使用

```
DCL v IntArr, i Integer;

TASK v(0) := 3; /* 要素の 1 つを変更 */
TASK i := v(9); /* 要素の 1 つを取得 */
```

演算子のパラメータとしてC言語の配列を使用すると、C言語の処理と同様にアドレスとしてそのC言語の配列が渡されます。これにより、実際のパラメータの内容を変更する演算子を定義できます。標準のSDLでは、このような定義は不可能です。

Refジェネレータ

ctypesパッケージのRefジェネレータは、ポインタ型を定義するために使います。次の例に、Refジェネレータの使用方法を示します。

例35: ポインタ型の定義

```
newtype ptr Ref(Integer)
endnewtype;
```

ptrソートは、Integerへのポインタです。

標準のSDLには、ポインタ型はありません。ポインタは、通常のSDLでは定義できない特性を持つため、慎重に使用する必要があります。Refジェネレータの使い方を説明する前に、SDLでポインタを使用する場合の注意点について触れておきます。

ポインタによるデータの不整合

複数のプロセスが、ポインタを使用して同じメモリの読み取りや書き込みを行った場合、データの不整合が発生する可能性があります。以下に、その例を示します。

- 飛行機予約システムで、座席が1つ残っている場合に2つの予約要求が同時に発生したと仮定します。座席の予約確認にポインタが使用されている場合、両方の要求が承認される可能性があります。これは「書き込み間の競合」と呼ばれます。
- あるプロセスによって配列変数が更新されるときに、他のプロセスがポインタを使用して同じ配列の変数を読み出すと、“新しい”要素を読み出す場合と“古い”要素を読み出す場合があり、最終的な結果が予測できなくなります。これは、一般的に「読み取りと書き込み間の競合」と呼ばれる問題です。

シミュレータやエクスプローラなどのツールは、ポインタに関する多数のエラーを検出できますが、これらのツールでも検出できない問題があります。この問題は、エクスプローラとシミュレータが、最悪でも一度に1つのSDLシンボルが処理されるという最小スケジュール単位が仮定されているためです。この仮定は、1つのプロセスにいつ割り込みが発生するかわからないターゲット オペレーティング システム（プリエンプティブ スケジューリング）では、正しくありません。ポインタを使用すると、データはまったく保護されないので、エクスプローラが問題を検出しない場合でも、データの不整合が発生する可能性があります。データアクセスに、リモート プロシージャや信号交換などのSDLの構成体を使用すれば、これらの問題は、すべて回避できます。

> 注意！
>
> 上記の理由から、他のプロセスにはポインタを渡さないでください！ 出力信号や、リモートプロシージャ呼び出し、プロセスの生成時、あるいはエクスポートおよび明示された変数などを使って、他のプロセスにポインタを渡さないでください！

この規則をどうしても守れない場合は、同じデータ領域に複数のポインタを保持することを避けるために、ポインタを渡したらすぐに“古い”ポインタを解放します。たとえば、（ポインタpに対して）以下の操作を行います。

```
OUTPUT Sig(p) TO ...;
TASK p := Null;
```

予測不可能なポインタ

デモンストレーションで実行する場合を、除いて常に正常に動作するSDLシステムがある場合、そのシステムではポインタを使用している可能性があります。ポインタによるバグは、検出が非常に困難です。システムが長い期間（偶然に）正しく動作していも、突然奇妙な振る舞いをする可能性があります。そのようなバグを発見するには、非常に長い期間がかかり、時には、まったく発見できないこともあります。

> 注意！
>
> ポインタによるバグは、検出が非常に困難です！

真の分散システムで動作しないポインタ

SDLシステムが“真の”分散システムである場合、つまり、各プロセスがそれぞれのメモリ空間を持っている場合、別のプロセスにポインタを送っても意味がありません。受信側プロセスは、そのポインタを使用しても何もできません。ゆえに、他のプロセスとポインタをやり取りすると、実装の対象となるアーキテクチャに制限が生じることになります。

移植性のないポインタ

Refジェネレータと演算子は、完全にIBM Rational固有のものです。ポインタを使用するSDLシステムが他のSDLツール上で動作することは、きわめてまれです。

SDLでのポインタの使用

ここでは、以上の注意点を踏まえた上で、SDLでポインタを使用する方のために、ポインタの使用方法について説明します。Refジェネレータによって生成されるポインタ型は、常にデフォルトの`Null`リテラルを持っています（C言語の`NULL`に対応します）。`Alloc`リテラルは、新しいメモリを動的に生成する際に使います。Allocリテラルの使用例は、後ほど示します。

> メモ：
>
> 動的に割り当てられたデータ領域の管理と、その領域が不要になった場合の解放処理は、ユーザーが行わなければなりません。

以下に、Refデータ型で利用できる演算子を示します。

```
"*>"  : Ref, Itemsort -> Ref;
"*>"  : Ref -> Itemsort;
"&"   : Itemsort -> Ref;
make! : Itemsort -> Ref;
free  : in/out Ref;
"+"   : Ref, Integer -> Ref;
"-"   : Ref, Integer -> Ref;
vstar2ref : Voidstar -> Ref;
ref2vstar : Ref -> Voidstar;
ref2vstarstar : Ref -> Voidstarstar;
```

さらに、Refデータ型には以下のプロシージャが定義されています。

```
procedure Free; fpar p Voidstarstar; external;
```

これらの演算子は、以下のように定義されています。

- `*>`（後置演算子）：
  ポインタの内容を取得または変更します。これは後置演算子なので、`p*>`と記述した場合、ポインタpの内容を返します。SDLの用語では、ポインタの抽出/更新演算子にあたります。

- `&`（前置演算子）：
  アドレス演算子です。これは前置演算子なので、`&var`と記述した場合、変数`var`へのポインタを返します。

- `make!`または`(. .)`
  メモリ領域を新たに確保し、パラメータの値を代入します。

- `free`
  パラメータとしてポインタ変数を取り、メモリ領域を解放してポインタ変数を`Null`にします。

- `+, -`:
  アドレスのオフセットを加減するために使います。これらの定義は、C言語の演算子と同様です。たとえば、pが構造体へのポインタの場合、`p+1`はバイト`p+1`ではなく、次の構造体を指します。

- `vstar2ref`:
  VoidStarを別のポインタ型に変換します。`void *`を返すC言語の関数の結果を変換する場合にのみ使用します。

- `ref2vstar`:
  ポインタをVoidstarに変換します。この演算子は、`void *`パラメータを持ったC言語の関数を呼び出すときに有用です。

- `ref2vstarstar`:
  ポインタのアドレスを`void **`として返します。この演算子は、`Free`プロシージャを呼び出すときに必要です。

- `Free`プロシージャ:(注:代わりに上述のfree演算子を使ってください)
  このプロシージャは、`alloc`によって割り当てられたメモリを解放します。下位互換性を保つためだけに提供しているものですので、代わりに上述のfree演算子を使うようにしてください。

例36: Ref演算子の使用 ——————————————————————

```
NEWTYPE ptr Ref(Integer)
ENDNEWTYPE;

DCL p ptr,
    i, j Integer;

TASK p := alloc; /* 動的に新しい integer を作成します。
                    p がその integer を指します */
/* ここで、p != Null であることを確認する必要があります */
TASK p*> := 10;  /* p の内容の変更 */
CALL free(p); /* integer 領域を解放 */
TASK p := (. 10 .); /* 領域を確保し、10 に初期化 */
CALL free(p);  /* integer を再び解放 */
TASK p := &i;  /* p は、今度は「i」を指します */
TASK p*> := 5; /* p の内容を変更します。つまり、
                  「i」も変更されます */
TASK j := p*>; /* p の内容の取得 (=5) */
```

ポインタでリンクされた構造の使用

ポインタは、リストやツリーなどのリンクされた構造を定義するときに便利です。ここでは、整数値を持つリンクされたリストの例を示します。図43に、リンクされたリストのためのデータ型が定義されたSDLの一部分と、リンクされたリスト

を実際に構築する遷移の一部分を示します。リストは、`Item`へのポインタによって表現されています。すべての`Item`は、リスト内で次のアイテムを指す`next`ポインタを持っています。リストの最後のアイテムが持つ`next`は`Null`です。

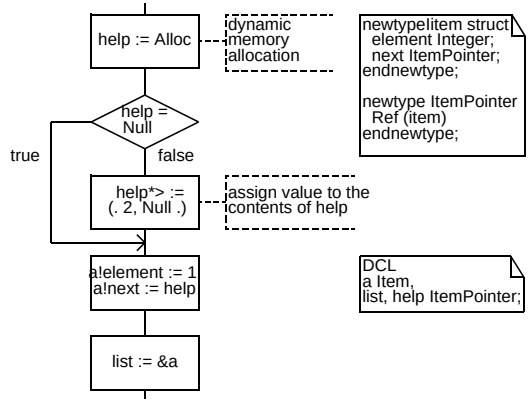


図43:: リンクされたリストの構築

図44に、リスト内のすべての要素の合計を計算するSDLの一部分を示します。リストに前のアイテムを指す要素があると、この計算は無限に続くことに注意してください。この例は、ポインタを使用するとエラーが簡単に発生することを示しています。

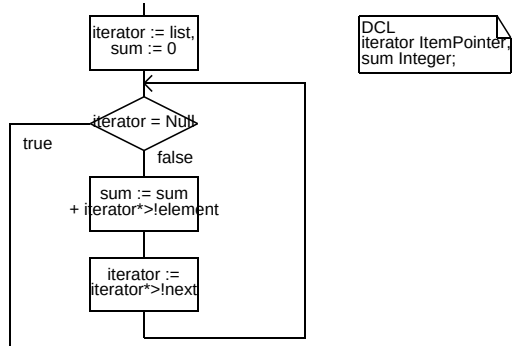


図44: リストの一巡

SDLでのASN.1使用

ASN.1は、データ伝送システムの仕様書やコード内で使用する頻度の高いデータ型を定義するための言語です。ASN.1は、ITU-TのX.680-X.683勧告で規定されている規格です。また、Z.105勧告では、SDLとともにASN.1を使用する方法が定義されています。SDL Suiteには、このZ.105のサブセットが実装されています。

ここでは、ASN.1のデータ型をSDLシステムで使用する方法について説明します。説明する内容は以下のとおりです。

- SDL Suite内でのASN.1モジュールの構成方法
- SDL内でのASN.1データ型の使用方法
- SDLとTTCN間でのASN.1データの共有方法

SDL SuiteでのASN.1モジュール編成

ASN.1モジュールを作成する際は、まず、ASN.1専用のチャプタを作成してください（たとえば、*ASN.1 Modules*など）。多数のASN.1モジュールを使用する場合は、オーガナイザ モジュールを作成して（ASN.1モジュールとは異なります）、グループ化します。『User´s Manual』の第2章「The Organizer」の43ページ、「Module」を参照してください。

図45のオーガナイザ画面の例には、2つのオーガナイザ モジュールを定義したチャプタが表示され、各オーガナイザ モジュールにはASN.1モジュールが定義されています。

図45: オーガナイザ内のASN.1モジュールの例

ここでは、SDLシステムでのASN.1モジュールの使用例を説明します。以下の`MyModule` ASN.1モジュールは、`mymodule.asn`というファイル内に記述されているものとします。

```
MyModule DEFINITIONS ::=
BEGIN

Color ::= ENUMERATED { red(0), yellow(1), blue(2) }

END
```

このモジュールには、`red`、`yellow`、`blue`という3つの値を持つ`Color`というデータ型の定義が含まれています。

まず、オーガナイザで[編集]メニューの[新規追加]をクリックして、*Text ASN.1*タイプの新しいダイアグラムを追加します。このとき[エディタで表示]はオフにします。[編集]メニューから[接続]を選択して「`mymodule.asn`」ファイルにこのダイアグラムを接続します。SDLでASN.1モジュールを使用するために、下の図46に示されているように、作成したシステム ダイアグラムにパッケージ参照句として`use MyModule;`を記述します。

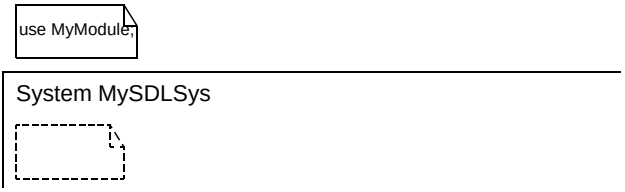


図46: SDL でのASN.1モジュールの使用

図47に、オーガナイザの現在の表示画面を示します。MySDLSysシステム シンボルの下のシンボルは、SDLシステムが外部ASN.1モジュールに依存していることを示す*依存リンク*です。以前のアナライザでは、SDLシステムで使用するためにASN.1モジュールの依存リンクが必要でしたが、現在ではコメントとしての機能しかないため、省略可能することもできます。

図47: ASN.1モジュールを使用するSDL システムのオーガナイザ画面

ASN.1モジュールが他のASN.1モジュールを使用する場合には、ASN.1モジュール間の依存リンクを作成する必要があります。

SDLでのASN.1データ型使用

ここまでの作業を完了すれば、`MyModule`のデータ型をSDLで使用することができます。ここで例として、文字列を対応する色に変換するSDLシステムを作成します。この処理には、以下の2つの信号が必要です。

- `GetColor`信号。この信号はASN.1データ型のIA5Stringパラメータを持ちます。
- GetColor信号がSDLシステムに送られると、SDLシステムは`ReturnColor`信号を返します。この信号は、文字列に対応する色が存在するかどうかを示すBOOLEANのパラメータと`Color`パラメータを持ちます。

図48に、以上の信号定義が記述されたシステム ダイアグラムを示します。

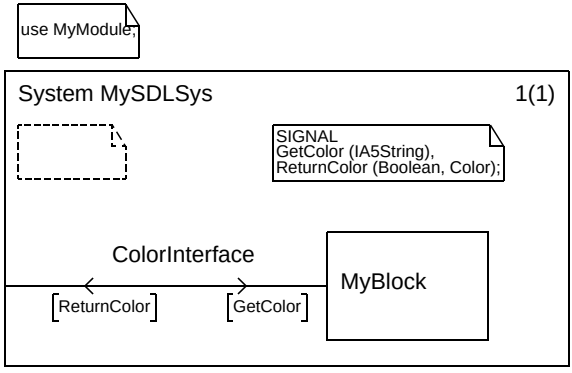


図48: SDL システム ダイアグラム

以下のMSCは、このシステムの使用方法を示しています。

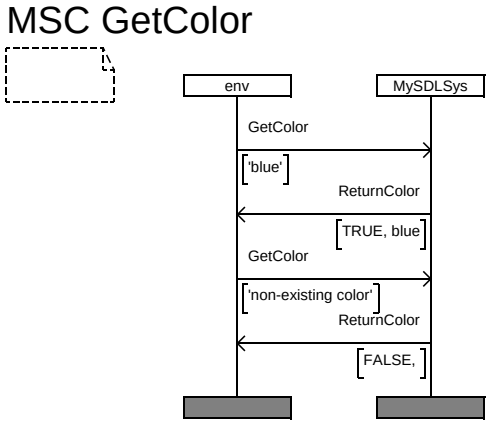


図49: GetColorのMSC

ASN.1データ型で使用できる値と演算子については、『User´s Manual』の第13章「The ASN.1 Utilities」の694ページ、「Translation of ASN.1 to SDL」を参照してください。

たとえば、`Color`は`ENUMERATED`型として定義されます。『User´s Manual』の第13章「The ASN.1 Utilities」の705ページ、「Enumerated Types」の割り当て規則を参照すれば、Colorで使うことができる演算子を確認できます。ここでは、`num`, `<`, `<=`, `>`, `>=`, `pred`, `succ`, `first`, `last`を使うことができ、さらに`=`と`/=`が常時使用可能です。

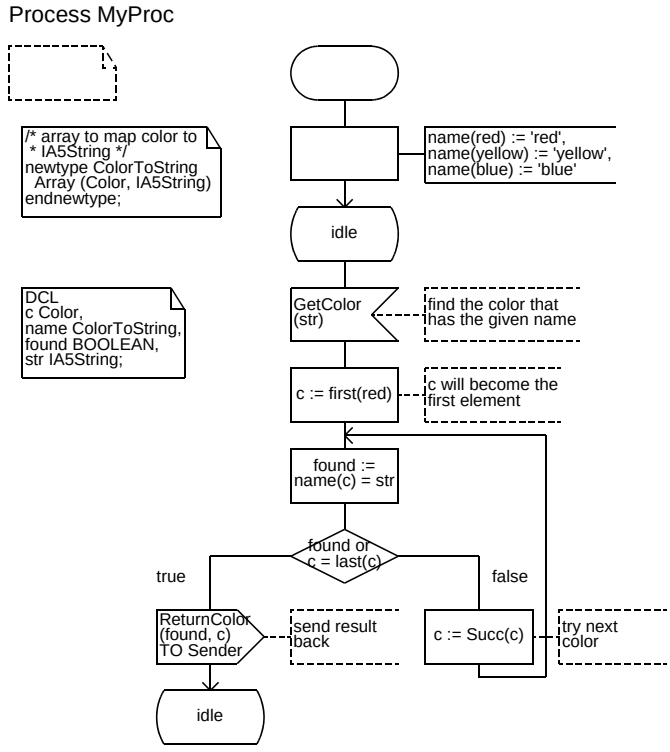


図50: SDL でのColorデータ型使用

図50は、`Color`を使用するSDLプロセスの一部を示しています。このプロセスには、`Color`のすべての値をチェックするループ制御があります。また、`Color`変数の宣言方法や、SDLの新しいソートの定義のもとで`Color`を使用する方法、および`first`、`last`、`succ`演算子の使用例が示されています。このプロセスについては、次の点に注意すべきです。

- 色をIA5Stringに変換する場合は、`ColorToString`データ型を使います。上のプロセスでは、その反対の変換を行っています。他の方法として`StringToColor Array (IA5String, Color)`を使用する方法もあり、この場合はループが不用になります（71ページの「Array」を参照）。ただし、ここで使用する例の目的は、すべての要素に対するループを使った参照方法を示すことにあるため、Arrayは使用しません。

- `c := first(red)`の`first`演算子は、最小の番号に関連付けられた要素を返します。したがって、`Succ`演算子を使用すれば、すべての要素を取得できます。この例では、単に`c := red`と記述することもできます。
- また、ASN.1の定義済みデータ型である`IA5String`が使用されていることを確認できます。`IA5String`は、SDLの`Charstring`定義済みソートのシンタイプです。

SDLでのASN.1定義済みデータ型使用

基本的なASN.1の定義済みデータ型は、直接SDLで使用できます。ほとんどの場合、ASN.1データ型はSDLデータ型と同じ名前を持ちます。たとえばASN.1データ型の`NumericString`はSDLでも`NumericString`です。しかし、ASN.1の定義済みデータ型の中には、`BIT STRING`のようにスペースを含むものがあります。SDLでは、スペースをアンダースコアで置き換える必要があるため、対応するSDLソートは`BIT_STRING`になります。

ASN.1の定義済みデータ型に対して定義されている演算子については、42ページの「SDLデータ型の使用」で説明されています。

SDL SuiteでのASN.1エンコード規則使用

ITU-T勧告X.690およびX.691で定義されているASN.1の構文はサポートされます。これについては『User´s Manual』の第58章「ASN.1 Encoding and De-coding in the SDL Suite」の2805ページ、「ASN.1 Encoding and De-coding in the SDL Suite」を参照してください。

SDLとTTCN間のデータ共有

ASN.1の利点の1つに、TTCNがASN.1をベースにしていることが挙げられます。SDLシステムと外部環境との間で送受信される信号パラメータとしてASN.1データ型を使用することによって、システムのテスト ケース仕様を決定する際に、パラメータの情報をTTCN Suiteで再利用できます。

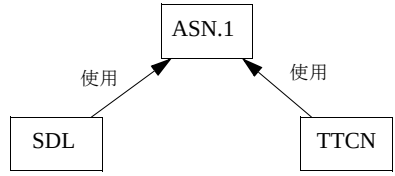


図51: SDLとTTCN間のASN.1の定義の共有

この方法には、SDLの仕様とTTCNのテスト仕様を容易に統一できるという大きな利点があります。

TTCN内での外部ASN.1の使用方法については、TTCN Suiteのマニュアルで詳細に説明されています。この項では、TTCN Linkを使用してSDLとTTCNの間でデータを共有する方法について簡単に説明します。

ここで、`Colors`が定義されたSDLシステムのテスト スイートを記述すると仮定します。オーガナイザに、`ColorTest`などの名前を付けた新しいTTCN Test Suiteダイアグラムを追加します。このテスト スイートでは、`Colors`データ型が含まれるASN.1モジュール"MyModule"の定義を使います。このために、ASN.1モジュールとテスト スイートの間に依存リンクを設定する必要があります。依存リンクの設定は、オーガナイザでリンクを設定するASN.1モジュールを選択し、[生成]、[依存]の順にクリックして、ASN.1モジュールを`ColorTest` TTCNテスト スイートに接続します。

TTCN Linkを使用すれば、SDLシステムの`MySDLSys`から宣言を生成することもできます。これを実現する最も簡単な方法は、SDLシステムをTTCNテスト セットに関連付けることです。オーガナイザで関連付けるSDLシステム ダイアグラムを選択し、[編集]、[関連付け]をクリックしてTTCNテスト セットにSDLシステム ダイアグラムを関連付けます。テスト スイートを示すオーガナイザ画面は、図52のようになります。

図52: ASN.1を使用するテスト スイートのオーガナイザ画面

オーガナイザでSDLシステムを選択し、[生成]、[実装]の順にクリックして、標準のカーネルに「*TTCN Link*」を選択すれば、SDLシステム用のTTCN Link実行ファイルを生成できます。`ColorTest`テスト スイート`ColorTest`をダブルクリックして、TTCN Suiteを起動します。[TTCN Link]、[Generate Declarations]をクリックすると、PCO、ASPデータ型定義、ASN.1データ型定義を自動的に生成できます。結果を参照すると、`Color`が[ASN.1 Type Definition by Reference]に存在することを確認できます。結果を示すテーブルは、つぎのようになります。

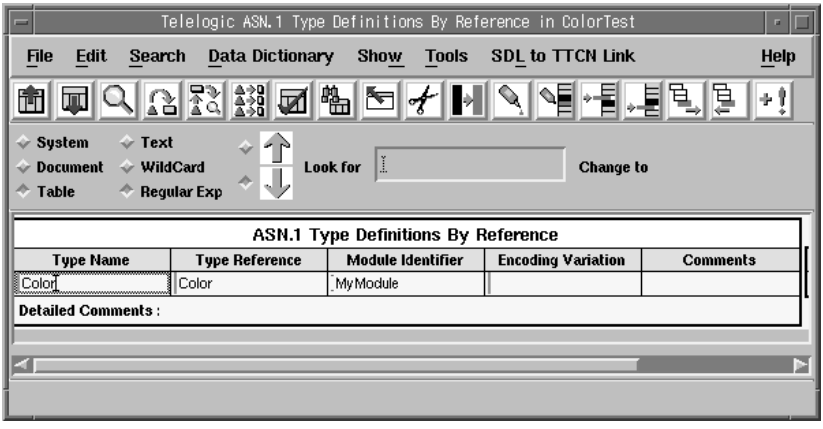


図53: TTCN Suiteの結果テーブル

これにより、このデータ型をテスト ケースの作成や、制約などに使用できます。新しい色を追加する場合など、後で`Color`の定義を変更する際は、テスト スイートを更新するために`Color`に対応するTTCNテーブルを選択します。そして、[アナライザ]ダイアログ ボックスで[強制分析を有効にする]と[Retrieve ASN.1 Definitions]の両方をオンにします。これで、TTCNテスト スイートは新しい`Color`の定義に応じて更新されます。

第 3 章 SDLの拡張表現の使用

この章では、SDLの拡張表現の中で、SDL Suiteで利用可能なIBM Rational独自の表現について説明します。

紹介する拡張表現には、OwnおよびORefジェネレータと、SDLで使用するアルゴリズムの拡張表現があります。SDL Suiteでサポートされている IBM Rational専用のSDLの拡張表現は他にもありますが、これらは主にデータ型、ジェネレータ、演算子に関するものです。これらの拡張表現については、第2章「データ型」を参照してください。

OwnとORefジェネレータ

はじめに

SDLから高速アプリケーションを生成する場合に問題になるのは、データ モデルです。その理由は、データ モデルを使うと多くの場所にデータをコピーしなければならないためです。2つのプロセスの間で信号を交換する場合、信号パラメータは送信元プロセスから信号にコピーされ、次に、信号から受信側プロセスにコピーされます。2つのプロセスが共通のメモリにアクセスできれば、信号を使ってデータへの参照のみを渡すだけで済むので、2回のコピー処理は必要なくなります。

Refというジェネレータを使用すると、参照を使ったデータ アクセスが可能になります（第2章「データ型」111ページの「Refジェネレータ」を参照してください）。ただし、Refジェネレータを使う場合にはいくつかの問題があります。

- 使用しないメモリは、ユーザー自身で解放する必要があります。メモリを解放しない場合は、どのような状況であってもその分のメモリが使用できなくなります。
- 故意または偶発的に、複数のプロセス インスタンスから同じメモリへ容易にアクセスできてしまいます。メモリへの同時アクセスを回避する処理を持たないプログラミング方法はリアルタイム制御では好ましくなく、不正な動作の原因になります。また、この種類のエラーは、通常、見つけるのが非常に困難です。

Ownジェネレータの基本的な性質

Ownジェネレータの目的は、上記の問題を解決することにあります。つまり、Ownジェネレータによって、コピー操作の回数が必要最小限にとどめられ、複数プロセスからメモリへの同時アクセスが制限され、さらに、ユーザーでのメモリ解放の必要性がなくなります。

これらの利点は、一度に1つのOwnポインタのみが同じメモリの参照を許可される、Ownジェネレータの基本的な性質によって実現されます。Ownデータ型で定義される変数はメモリの所有者ということができます。また、メモリの所有権は、代入によって別の変数に渡すことができます。

例37: Own変数 ――

```
newtype Data struct
  a, b integer;
endnewtype;

newtype Own_Data Own(Data)
endnewtype;

dcl
  v1 Own_Data,
  v2 Own_Data;

task v1 := v2;
```

上記の代入操作は、以下のように解釈されます。

- v1がメモリを参照している場合は、そのメモリは解放されます。
- v1にv2の値が代入されます。つまり、v1はv2と同じメモリを参照します。
- v2はNullに設定されます。

――

上記の動作によって、Ownジェネレータの基本的な機能が実現されます。つまり、アクセスされないすべてのメモリは解放され、データへの参照は1つだけが存在します。

より複雑な条件では、上記の操作の実行順序も多少複雑になります。上記の例と同じデータ型と変数を使用し、以下の3つのOwn_Dataパラメータを取るPプロシージャを想定します。

```
task v1 := P(v2, v1, v2);
```

上の式の動作は以下のようになります。

右辺の評価は、左から右へ実行されます。つまり、Pの最初の仮パラメータから評価が開始されます。最初のパラメータにはv2の値が割り当てられ、最初のパラメータはメモリの所有者になります。そして変数v2にはNull値が割り当てられます。同様の処理が変数v1と第2の仮パラメータに対して実行されます。Pの第3の仮パラメータには、このときv2がNullなので、Null値が割り当てられます。

次にPプロシージャが呼び出され、戻り値が取得されます。Pプロシージャの戻り値が左辺の変数v1に代入される前に、現在v1が参照しているメモリが解放されます。上記の式では、Pの第2の仮パラメータが既にメモリの所有者になっているため、v1はNullになります。最後に、変数v1には、Pによって返される値が割り当てられます。

このように、所有権は、代入の際だけでなく参照が他の場所に割り当てられた際にも必ず渡されます。所有権の移動は、代入、入力、出力、設定、リセット、プロシージャ呼び出し、作成などの際に発生します。また、所有権が移動しないのは、以下の場合のみです。

- 以下のように明示的にコピー関数を使用する場合。
  ```
  task v1 := copy(v2);
  ```
- インポート、エクスポート、ビュー操作の実行時。これらの操作には、暗黙的にコピー操作が含められていると見なすことができます。
- 標準の関数である'='と"/="の呼び出し。

copy(v2)は、深いレベルまでコピーする関数です。つまり、コピーされたデータ構造内にあるすべてのOwnポインタに対しても、コピー操作が実施されます。このような深いレベルへのコピーが実行されない場合は、同じメモリへ複数の参照が存在することになります。

Ownジェネレータの定義

Ownジェネレータは、以下のようにSDL内に定義します。

```
GENERATOR Own (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : Own, ItemSort -> Own;
    "*>"  : Own -> ItemSort;
    make! : ItemSort -> Own;
  DEFAULT Null;
ENDGENERATOR Own;
```

Ownジェネレータは、割り当てられたメモリへのアクセスで、ポインタを利用するための手段になります。Null値は、今まで通り空の参照として解釈されます。演算子"*>"はExtract!とModify!演算子を表します。つまり、ポインタを使ってメモリを参照または変更するための手段です。上記のデータ型と変数によって、以下の記述が可能です。

```
task v1*>!a := 1;
task i := v2*>!b;
   /* 整数の割り当てを表します。「i」は整数型の変数です */
task d := v2*>;
   /* 構造体レベルの割り当てを表します。「d」は Data データ型です */
```

"*>"演算子はC言語の'*'と同じ性質を持ちます。つまり、"v1*>"の意味はC言語の"*v1"と同じです。構文を多少簡略化するために、SDL アナライザを設定して

暗黙的に"*>"を必要な式に挿入させることができます。この設定を行うと、上記の記述を下記のように修正できます。

```
task v1!a := 1;
task i := v2!b;
   /* 整数の割り当てを表します。「i」は整数型の変数です */
task d := v2;
   /* 構造体レベルの割り当てを表します。「d」はDataデータ型です */
```

これで、多少読みやすくなりました。この暗黙のデータ型変換の詳細については、134ページの「暗黙のデータ型変換」を参照してください。

Ownポインタを使ってデータを参照し、データ内のコンポーネントを操作するには、まずOwnポインタを正しい値で初期化する必要があります。前の定義に記述されている通り、既定値はNullです。変数を初期化する場合は、Make!演算子を使用する方法が最も適切です。他の場合と同様にMake!の実際の構文は"(. x .)"となります。xの値は、現在のOwnポインタのItemSort値になります。

前述のデータ定義を使用した場合の初期化の例を以下に示します。

```
dcl v1 Own_Data := (. (. 1, 2 .) .);
task v1 := (. (. 5, 5 .) .);
```

内側の"(. .)"は構造体の値であり、外側の"(. .)"はOwnのMake!関数です。ここで、Tデータ型からOwn(T)データ型への暗黙のデータ型変換が存在し、Tデータ型の値の両側に暗黙の"(. .)"が挿入されるため、二重括弧の表現を避けることができます。したがって、上の例は、以下のように記述できます。

```
dcl v1 Own_Data := (. 1, 2 .);
task v1 := (. 5, 5 .);
```

134ページの「暗黙のデータ型変換」を参照してください。"*>"とMake!に加えて、Ownポインタで利用できる操作には代入、等価の評価、およびコピーがあります。代入演算子については既に説明しました。等価の評価（'='と"/="）を使う場合は、2つのOwnポインタが等価になることはないので、ポインタの等価が評価されるわけでありません。その代わり深いレベルでの等価の評価が行われます。すなわち、Ownポインタによって参照される値が比較されます。

Copyという暗黙の演算子は、すべてのデータ型に組み込まれています。この演算子は渡された値のコピーを返します。Ownポインタ以外のデータ型、またはOwnポインタを含まないデータ型では、この演算子は同じ値を返すだけなので特別な機能を持ちません。しかし、OwnポインタまたはOwnポインタを含む構造体の値では、このコピー機能によってOwnポインタが参照する値がコピーされます。

ORefジェネレータ

ORefジェネレータは、Ownジェネレータとともに使用することが想定されており、特定のアルゴリズムの実行時に、所有されているデータを一時的に参照するために使います。この参照時、メモリの所有権には影響を与えません。たとえば、Ownポインタをリストのリンクに使用し、リストの長さを算出するプロシージャを記述する場合、リストを順次参照する一時的なポインタが必要になります。このポインタにOwnポインタを使うと、同じメモリを参照するOwnポインタが1つしか許容されないので、参照時にリストは破壊されてしまいます。

例38: OwnとORef

```
newtype ListElem struct
  Data MyType;
  Next ListOwn;
endnewtype;

newtype ListOwn Own(ListElem)
endnewtype;
newtype ListRef ORef(ListElem)
endnewtype;

procedure ListLength; fpar Head ListRef;
returns integer;
dcl
  Temp ListRef,
  Len  Integer;
start;
  task Len := 0, Temp := Head;
  again :
  decision Temp /= null;
  (true) :
    task Len := Len+1, Temp := Temp!Next;
    join again;
  (false) :
  enddecision;
  return Len;
endprocedure;

dcl
  MyList ListOwn,
  L integer;

task L := call ListLength(MyList);
```

仮パラメータのHeadとローカル変数のTempに対するListRefデータ型の使用に着目します。HeadがListOwnデータ型に定義された場合を仮定すると、MyList変数はListLengthを呼び出した後にNullになり、期待通りの動作をしません。また、Temp変数をListOwnデータ型で定義すると、`Temp := Temp!Next`というステートメントによって、リスト全体のリンクが解除されてしまいます。

ORefのもう1つの典型的な使用例に、リンクされたリスト内で後方ポインタを導入し、リストのダブルリンクを行う方法があります。前方ポインタがOwnポインタの場合、後方ポインタをOwnポインタにすることはできません。これは、同じオブジェクトに2つのOwnポインタが存在することになるためです。

ORefジェネレータは、以下のように定義できます。

```
GENERATOR ORef (TYPE Itemsort)
  LITERALS
     Null;
  OPERATORS
    "*>"  : ORef, ItemSort -> ORef;
    "*>"  : ORef -> ItemSort;
    "&"   : ItemSort -> ORef;
    "="   : ORef, ItemSort -> Boolean;
    "="   : ItemSort, ORef -> Boolean;
    "/="  : ORef, ItemSort -> Boolean;
    "/="  : ItemSort, ORef -> Boolean;
  DEFAULT Null;
ENDGENERATOR;
```

上記の、"*>"は参照しない演算子として使用します。また、'&'はアドレス演算子です。

ランタイム エラー

OwnとOrefに関連するランタイム エラーは、以下のような状況で発生します。

- Nullポインタに対する参照しない演算。
- 解放されたオブジェクトをORefポインタが参照した場合。
- 別のプロセスが所有するオブジェクトをORefポインタが参照した場合。
- Ownポインタの循環参照が形成された場合。この状態になると、メモリを解放できなくなります。

これらの問題は、すべてシミュレーションとバリデーションの段階で検出されます。ただし、ORefポインタの参照するデータ領域が、解放されてから再び割り当てられた領域の場合は、ORefポインタが有効でなくなるため検出されなくなります。

上記のデータ型を使用した場合のランタイム エラーの例を以下に示します。

```
dcl
  L1, L2 ListOwn,
  R1, R2 ListRef,
  I      Integer;

task
  L1 := null,
  R1 := null,
  L1!Data := 1,
    /* エラー : Null ポインタの参照しない演算 */
  I := R1!Data;
    /* エラー : Null ポインタの参照しない演算 */

task
  L1 := (. 1, null .),
  R1 := L1,
  L1 := null,
  I := R1!Data;
    /* エラー : 解放されたメモリへの参照 */

task
  L1 := (. 1, null .),
  R1 := L1;
output S(L1) to sender;
task
  I := R1!Data;
    /* エラー : プロセスが所有していないメモリへの参照 */

task
  L1 := (. 1, null .),
  L1!Next := (. 2, null .),
  L1!Next!Next := L1;
    /* エラー : Own ポインタの循環参照 */
```

暗黙のデータ型変換

メモ :

既定では、暗黙のデータ型変換はSDL アナライザでオフになっています。この設定は、オーガナイザの[分析]ダイアログでオンにできます。しかし、暗黙のデータ型変換は、意味分析の所要時間に影響します。式が複雑になるほど、暗黙のデータ型変換が分析の所要時間に及ぼす影響は大きくなります。この理由は、暗黙のデータ型変換の数が、式の長さによって指数関数的に増加するためです。

暗黙のデータ型変換の目的は、Ownジェネレータの使用を簡単にすることにあります。暗黙のデータ型変換によってデータ構造を操作するコードは、Tデータ構造体を使用する場合と、TへのOwnポインタであるown(T)を使用する場合で、同じになります。これにより、ユーザーがデータを他に渡すときには、所有権を渡すべきか、コピーを実行するべきかを検討するだけで済みます。

暗黙の変換では、代入のデータ型は変更されません。たとえば以下のような代入を行うとします。

```
task R1 := L1;
```

この代入では、R1に暗黙のデータ型変換は適用されません。これは、データ型変換によって代入のデータ型が変更されてしまうからです。したがって、データ型変換は、データ型の整合を得るために右辺のL1に適用される場合もあります。また、以下の代入を行うと、

```
task Q(a) := ...;
```

暗黙のデータ型変換が、インデックス式、つまりaに適用される場合があります。

等価の評価や、以下のような類似した状況の場合、

```
L1 = R1
```

暗黙のデータ型変換は、まず左辺に適用されます。この例では、暗黙のデータ型変換がL1に適用されます。したがって、正しい変換が行われた場合は、適用されたデータ型が採用されます。正しくない場合は、暗黙のデータ型変換が右辺のR1に適用されます。

Tデータ型と、2つのジェネレータのインスタンスである`Own_T = Own(T)`と`ORef_T = ORef(T)`が定義されているものと仮定します。また、以下の変数の定義が仮定されているものとします。

```
dcl
  t1 T,
  v1 Own_T,
  r1 ORef_T;
```

この条件では、以下のような暗黙のデータ型変換が可能です。

```
1. Own_T  -> T,        by v1 -> v1*>
2. T      -> Own_T,    by t1 -> (. t1 .)
3. Own_T  -> ORef_T,   by v1 -> demote(v1)
4. ORef_T -> Own_T,    by r1 -> (. r1*> .)
5. T      -> ORef_T,   by t1 -> &t1
6. ORef_T -> T,        by r1 -> r1*>
```

1と6のデータ型変換によって、コンポーネントの選択を記述する際に、"*>"を省略できます。たとえば、以下のように記述するのではなく、

```
a*>!b*>(10)*>!c
```

次のように記述できます。

```
a!b(10)!c
```

この記述は、通常のRefポインタでも可能です。

2のデータ型変換によって、OwnポインタにOwnポインタ コンポーネントのデータ型の新しい値を割り当てることができます。Aが2つの整数を含む構造体へのOwnポインタである場合、以下のように記述できます。

```
A := (. 1, 2 .);
```

これは、以下と同じ意味です。

```
A := (. (. 1, 2 .) .);
```

ここで、内側の"(. .)"は構造体のMake!関数で、外側の"(. .)"はOwnポインタのMake!関数です。

この記述は、通常のRefポインタでも可能です。

3のデータ型変換によって、ORefポインタをOwnポインタに割り当てることができます。この割り当ては前出の例で既に使用しましたが、ORefとOwnポインタは異なるデータ型であるため、直接的には可能ではありません。Ownポインタを対応するORefポインタへ変換するには、demote演算子実行します（「対応する」の意味は、Ownポインタ データ型と、同じスコープ ユニットに同じコンポーネント データ型を持つ最初のORefが定義されることを示しています)。

4のデータ型変換によって、ORef値から新しいOwn値を生成できます。この変換は2段階で行われます。まず変換6によってORef_TからTに変換され、次に変換2によってTからOwn_Tに変換されます。

5のデータ型変換によって、以下の記述をするとORef_TポインタからDCL変数を参照できます。

```
task r1 := t1;
```

この式は、以下と同じ意味になります。

```
task r1 := &t1;
```

ここで、'&'はアドレス演算子です（C言語の場合と同様)。

SDLのアルゴリズム

SDLの課題の1つには、アルゴリズムを記述するためのサポートが不足していることが挙げられます。通信を含まない純粋な計算では、SDLの図式形式が通常のフローチャートのようになる傾向があり、高度なアルゴリズムの記述には適していません。また、このように規模が大きくなるSDLダイアグラムの部分によって、マシンの状態や通信部分など、SDL記述のより重要な部分が理解しにくくなります。

ここで説明するアルゴリズムの拡張記述は、この問題を解決するために、Taskシンボル内にテキスト形式でアルゴリズムを記述することや、テキスト シンボル内にテキスト形式でプロシージャと演算子を定義することを可能にしています。この方法には、通常のSDLと比較して以下の2つの利点があります。

- アルゴリズムを、通常のプログラミング言語のようにコンパクトな形式で記述できるので、SDLダイアグラムのその他の重要な記述を理解しやすくなります。
- アルゴリズム内で使用される言語には、if-then-elseやループ ステートメントなど、通常のSDLより強力なアルゴリズム構造が用意されています。

さらに、アルゴリズムの拡張記述によって、SDL/GRのテキスト シンボル内でプロシージャと演算子をテキスト形式で定義できるようになりました。

SDLのアルゴリズムの拡張記述は、SDLの次期ITU勧告に反映される公式のMaster List of Changesに取り入れられることがITU Study Group 10によって承認されています。SDL SuiteのSDLアルゴリズムのサポートとITUの定義には、いくつかの相違点があります。これらの相違については、以降の説明の際に注記します。

以下に、拡張された構造を示します。

- 複合ステートメント
- ローカル変数
- Ifステートメント
- Decisionステートメント
- ループ ステートメント
- ラベル ステートメント
- ジャンプ ステートメント
- 空のステートメント

複合ステートメント

*複合ステートメント*は、アルゴリズムの拡張記述の基本概念となります。複合ステートメントは'{'で始まり、一連の変数宣言とステートメントが続き、'}'で終わります。

複合ステートメントは、以下の3つの場所で使用できます。

- TASKの内容
- テキスト シンボル内のプロシージャまたは演算子の定義の本文
- 複合ステートメント内の1ステートメント

> メモ：
>
> ITU言語の定義では、プロシージャや演算子の本文は1ステートメントでもかまいませんが、SDL Suiteでは複合ステートメントが必要です。これは、本文が1つのステートメントで構成される場合も"{ }"で囲む必要があることを意味します。

また、"{ }"は、SDL エディタのTaskシンボルに記述することはできません。"{ }"は、SDLシステムを分析する際、システムのSDL/PRへの変換時に付加されます。

例39 ———————————————————————————————

SDL/GRのTaskシンボルの内容：

```
a := b+1;
if (a>7) b := b+1;
```

SDL/PRの該当するコード：

```
task {
  a := b+1;
  if (a>7) b := b+1;
};
```

例40 ———————————————————————————————

SDL/GRまたはSDL/PRのテキスト シンボル内のプロシージャ：

```
procedure p fpar i integer returns integer
{
  if (i>0)
    i := i+1;
  else
    i := i-1;
  return i;
}
```

———————————————————————————————————

ローカル変数

複合ステートメント内には、ローカル変数を定義することができます。ローカル変数は複合ステートメントに入るときに作成され、複合ステートメントから出るときに破棄されます。複合ステートメントの機能は、その複合ステートメントの位置で定義されて呼び出されるパラメータを持たないプロシージャによく似ています。

複合ステートメント内の変数宣言は、"exported"と"revealed"が使用できないことを除いて通常の変数宣言と同じです。以下に宣言の例を示します。

```
dcl
  a, b integer := 0,
  c    boolean;
```

ステートメント

複合ステートメント内のステートメントは、以下のいずれかです。

- 複合ステートメント
- outputステートメント
- createステートメント
- setステートメント
- resetステートメント
- exportステートメント
- returnステートメント（プロシージャと演算子内のみ）
- プロシージャ コール ステートメント
- 割り当てステートメント
- if ステートメント
- decisionステートメント
- ループ ステートメント
- ラベル ステートメント
- ジャンプ ステートメント
- 空のステートメント

各ステートメントと各変数宣言ステートメントの末尾は、';'で終わります。以下のステートメントは、通常のSDL/PRと同じ構文を使用します。

output, create, set, reset, export, return, call, assignment

例41: 通常のSDL/PRステートメント

```
output s1(7) to sender;
output s2(true, v1) via sr1;
create p2(11);
set(now+5, t);
reset(t);
export(v1);
return a+3;
call prd1(a, 10);
a := a+1;
```

メモ：

ITU言語の定義では、プロシージャ呼び出しのキーワード`call`は省略可能ですが、SDL Suiteでは必要です。

Ifステートメント

ifステートメントの構造：

```
if ( <Boolean式> )
  <ステートメント>
else
  <ステートメント>
```

elseの部分は省略可能です。最初にBoolean式が計算されます。結果がTrueの場合は最初のステートメントが実行され、False場合は、elseステートメントが存在すれば、そのステートメント実行されます。

例42

```
if (a>0)
  a := a+1;

if (a=0) {
  a := 100;
  b := b+1;
} else {
  a := a+1;
  b := 0;
}
```

elseに対応する複数のifステートメントが存在する場合（ぶら下がりelseの問題）、最も内側のifが常に選択されます。

例43

```
if (a>0)
  if (b>0)
    a := a+1;
  else
    a := a-1;
```

これは以下を意味します。

```
if (a>0) {
  if (b>0)
    a := a+1;
  else
    a := a-1;
}
```

Decisionステートメント

decisionステートメントは、SDL内の通常のdecisionとほとんど同じでものです。つまり、多分岐ステートメントです。decisionステートメントと通常のステートメントの主な違いは、decisionステートメント内のすべてのパスがenddecisionで終わることにあります。

例44

```
decision (a) {
(1:10) : {call p(a); a := a-5;}
(<=0)  : a := a+5;
else   : a := a-5;
}
```

上記decisionのテスト式と評価値は、通常のdecisionと同じ構文と意味を持ちます。評価値の後にステートメントが続きます。このステートメントは複合ステートメントでもかまいません。

ループ ステートメント

ループ ステートメントは、ステートメント（通常は複合ステートメント）の実行を繰り返すために使用します。ループはループ変数によって制御されます。ループ変数は、ループ内でローカルに定義することができます。また、ループ外で定義することもできます。

ループの制御部には、以下の3つのフィールドがあります。

- 開始値を割り当てるループ表示変数
- ループの評価式
- 新しいループ変数の値

例45

```
for (a := 1, a<10, a+1)
  sum := sum+a;
```

上記の式は、C言語と同様の構文によって以下のように記述できます。

```
a = 1;
while (a<10) {
  sum = sum+a;
  a = a+1;
}
```

SDLとC言語は、変数の更新する際の処理が異なります。C言語ではステートメントによって変数が更新されますが、SDLではループ変数に割り当てられている式によって更新されます。

ループ表示変数には、新しい変数を定義することもできます。また、以前に定義された変数を使用することも可能です。以下にループ変数の記述例を示します。

```
for (a := 1, ...
for (dcl a integer := 1, ...
```

ループ ステートメントのその他の用法について以下に示します

- ループの制御部には、空のフィールドがあってもかまいません。しかし、ループ表示変数（最初のフィールド）が空の場合は、ループ変数を更新するフィールド（第3のフィールド）も必ず空でなければなりません。以下に空のフィールドの記述例を示します。

  ```
  for (a := 1, , a+1) ..
  for ( , , ) ...
  ```

- ループには、複数のループ制御を定義できます。以下に複数の定義例を示します。

```
for (a := 1, a<10, a+1; b := 1, b<5, b+1)
  sum := sum+a+b;
```

上記の式は、C言語と同様の構文によって以下のように記述できます。

```
a = 1;
b = 1;
while ( (a<10) and (b<5) ) {
  sum = sum+a;
  a = a+1;
  b = b+1;
}
```

- ループから抜けるときは、breakステートメントを使用します。143ページの「ラベル ステートメント」と、144ページの「ジャンプ ステートメント」を参照してください。

- ループ ステートメントは、thenステートメントで終わることもあります。thenステートメントは、ループのテスト式がFalseになったためにループが終了すると実行されます。ループがbreakステートメントによって終了した場合は、thenステートメントは実行されません。以下にbreakステートメントとthenステートメントの記述例を示します。

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

ラベル ステートメント

ラベル ステートメントは、ステートメントの前に付ける単なるラベルです。このラベルは、ラベルに続くステートメントがループ ステートメントの場合にのみ意味があります。ループ ステートメントから抜け出る際にbreakステートメントでこのラベル名を使います。以下に記述例を示します。

```
L:
for (i:=0, i<10, i+1)
  sum := sum+a(i);
```

> メモ：
>
> SDLのアルゴリズムの拡張記述では、joinとgotoステートメントは使用できません。

ジャンプ ステートメント

breakとcontinueのジャンプ ステートメントは、ループ内の実行フローを変更するために使用します。

continueステートメントは、ループ内でのみ使用します。continueステートメントを実行すると、ループ内の残りの部分がスキップされ、ループ変数が次の値に更新されてから、処理が継続されます。

例46

```
for (a:=1, a<10, a+1) {
  if (sum > limit) continue;
  sum := sum+arr(a);
}
```

上記の式は、C言語と同様の構文によって以下のように記述できます。

```
a = 1;
while (a<10) {
  if (sum > limit) goto cont;
  sum = sum + arr[a];
cont :
  a = a+1;
}
```

breakステートメントは、ループの実行を中断し、ループの後のステートメントに直接ジャンプするために使用します。

例47

```
ok := false;
for (a:=1, a<10, a+1) {
  sum := sum+arr(a);
  if (sum > limit) break;
}
then
  ok := true;
```

上記の式は、C言語と同様の構文によって以下のように記述できます。

```
ok = false;
a = 1;
while (a<10) {
  sum = sum + arr[a];
  if (sum > limit) goto brk;
  a = a+1;
}
ok = true;
brk:
```

breakステートメントは、最も内側のループ ステートメントから抜け出します。ラベル付きのループ ステートメントを使用すれば、外側のループから抜け出すこともできます。

例48 ———————————————————————————————

```
L: for (x:=1, x<10, x+1) {
     a := 0;
     for (y:=1, y<10, y+1) {
       a := a+y;
       if (call test(x,y)) break L;
     }
   }
```

内側のループに定義されたbreakステートメントは、外側のループのラベルを指定しているので、2つのループから抜け出します。

———————————————————————————————————

空のステートメント

何も記述しない空のステートメントも使用できます。これは、ループ ステートメントなどで利用します。

```
for (i:=1, Arr(i)/=0 and i<Limit, i+1) ;
  /* このループは、Arr配列内でゼロ値を持つ最初の要素の
     インデックス値を「i」に設定します */
```

アルゴリズムの拡張記述のグラマー

メタ グラマー：

「'dcl', ')', ';'」：終端記号の例です。

「<Stmt>, <Name>」： 非終端記号の例です。

「::=」： 以降にその定義が続くことを意味します。

「$」： 空として使用されていることを意味します。

「*」： 0以上の発生を意味します。

「+」： 1以上の発生を意味します。

「|」： 「or」を意味します。

グラマー開始 ——————————————————————————————

```
<CompoundStmt> ::=
   '{' <VarDefStmt>* <Stmt>* '}'

<VarDefStmt>   ::=
   'dcl'     <Name> (',' <Name>)* <Sort> (':=' <Expr> | $)
      ( ',' <Name> (',' <Name>)* <Sort> (':=' <Expr> | $) )*
   ';'

<Stmt>         ::=
   <CompoundStmt>      |
   <Outputx> ';'       |
   <CreateRequest> ';' |
   <Setx> ';'          |
   <Resetx> ';'        |
   <Export> ';'        |
   <Return> ';'        |
   <ProcedureCall> ';' |
   <IfStmt>            |
   <LabelStmt>         |
   <AssignmentStatement> ';' |
   <DeciStmt>          |
   <LoopStmt>          |
   <JumpStmt> ';'      |
   <EmptyStmt> ';'

<IfStmt>       ::=
   'if' '(' <Expr> ')' <Stmt> ('else' <Stmt> | $)

<DecisionStmt> ::=
   'decision' '(' <Expr> ')' '{'
   ( <Answer> <Stmt> )+
   ( 'else'   <Stmt> | $)
   '}'
```

```
<Answer>       ::= same as answer in ordinary decisions

<LoopStmt>     ::=
   'for' '(' (<LoopClause> (';' <LoopClause>)* | $) ')'
     <Stmt>
   ('then' <Stmt> | $)

<LoopClause>   ::=
   (<LoopVarInd> | $) ',' (<Expr> | $) ',' (<Expr> | $)

<LoopVarInd>   ::=
   'dcl' <Name> <Sort> ':=' <Expr> |
   <Identifier> (':=' <Expr> | $)

<LabelStmt>    ::=
   <Label> <Stmt>

<JumpStmt>     ::=
   <Break> (<Name> / $) |
   <Continue>

<EmptyStmt>    ::=
   $
```

グラマーの終了 ――

シミュレータとエクスプローラのアルゴリズム

下の表に、SDLシミュレータとSDLエクスプローラ内での、新しいアルゴリズムの拡張記述に対するテキスト形式のトレースを示します。

ステートメント	テキスト トレース	コメント
複合		トレースはありません。
If	`IF (true)` `IF (false)`	
Decision	`DECISION  Value: 7`	通常のdecisionと同じトレースです。

ループ	`LOOP variable b := 3`
	`LOOP test TRUE`
	`LOOP test FALSE`
		ループ変数の割り当てに対するトレース。
		ループのテストに対するトレース。

ジャンプ	`CONTINUE`
	`BREAK`
	`BREAK LoopName`	

空		トレースはありません。

変数宣言のない複合ステートメントは、単なるステートメントの羅列と見なすことができますが、変数宣言のある複合ステートメントは、名前とパラメータを持たな

いプロシージャに対するプロシージャ コールと見なすことができます。しかし、この暗黙のプロシージャ コールや返送に対するトレース情報は生成されません。

複合ステートメントの変数は、シミュレータ インターフェイスで、実際のプロシージャと同じように利用できます。つまり、*Up*と*Down*コマンドを使用して、異なるスコープ内の変数を参照できます。なお、ループ変数で定義される変数は、それ自身のスコープが適用されます。

このローカル スコープ内での変数の標準的な処理に対して、複数のプロシージャによって定義された複合ステートメントは例外的な処理が適用されます。

以下のプロシージャを参照してください。

```
procedure p
  fpar in/out a integer
{
  dcl b integer;
  ...
}
```

パラメータaと変数bは、同じプロシージャのスコープが適用されます。最も外側にあるプロシージャスコープの複合ステートメントには、前述の一般規則が適用されます。

アプリケーションの実行パフォーマンス

Cadvanced

アルゴリズムの拡張記述のすべての概念は、ローカル スコープでの変数宣言（ループ変数を含みます）を除いて、C言語に効率的に実装できます。そして、このような複合ステートメントはSDLのプロシージャ コールになります。

Cmicro

アルゴリズムの拡張記述のすべての概念は、C言語に効率的に実装できます。変数を含んだ複合ステートメントはCブロック ステートメントを使って実装できます。

第 4 章 プロジェクトの構成

この章では、複数の人員で構成されたプロジェクトにおいて、オーガナイザで扱うダイアグラムやその他のドキュメントを管理する方法について説明します。ダイアグラム管理に関連するSDL Suiteの機能は、オーガナイザに装備されており、この機能をプロジェクトの開発環境で使用することができますオーガナイザのリファレンス マニュアルは、『User´s Manual』の第2章、「The Organizer」に掲載されています。

はじめに

概要

一般的に、ソフトウェア開発のプロジェクトには、多くのソフトウェア技術者が参加し、各技術者がシステムの各部分を担当します。このようなプロジェクトでは、システムの各部分に対して、慎重に整合を取る必要があり、さらに正しくバージョンを管理する必要があります。一般的に、作成が終了したシステムのバージョンは、すべてのユーザーがアクセス可能な共有領域に保存します。この章では、格納領域をオリジナル エリアと呼ぶことにします。オリジナル エリアは通常、以下のいずれかの構成を取ります。

- 単一のディレクトリにシステムの各バージョンを格納し、RCSやCM SYNERGY、ClearCaseなどのバージョン/リビジョン管理システムによって管理する方法。
- 一連のディレクトリによって構成し、各ディレクトリをシステムのそれぞれのバージョンに対応させる方法。各ディレクトリには、前のバージョンからの変更点を表す情報のみを格納します。たとえば、PBXというシステムがある場合、PBXシステムのバージョン1を、PBX-1ディレクトリに保存します。PBX-1にはバージョン1のすべての要素が格納されます。そして、PBXシステムのバージョン2をPBX-2ディレクトリに保存します。PBX-2は、PBX-1と比較され変更された部分のみがこのディレクトリに格納されます。

さらに、各プロジェクトのメンバは、各自が現在作業しているシステムの担当部分を保存するために、独自の*ワーク エリア*を持ちます。さらに、プロジェクトでは、リファレンス エリアを使うこともできます。リファレンス エリアには、異なるプロジェクト間で共有する情報を1つまたは複数のパッケージにまとめて格納します。

このような作業形態は、オーガナイザ ツールによってサポートされます。プロジェクトのメンバは、適切に作業が分配され構造化された環境のなかで、独立して作業することができます。オーガナイザのツールがサポートする主要な機能を以下に示します。

- ユーザー環境を個別にカスタマイズできます。
- SDLパッケージ、ダイアグラム、メッセージ シーケンス チャート、物理ファイルなどによって、SDLシステムの内容を簡単に視覚化できます。
- 各ユーザーのワーク エリアから、共有オリジナル エリアに最新バージョンを更新することできます。

- 異なる格納領域間で、SDLダイアグラムとメッセージ シーケンス チャートをコピーできます。

サポートの拡張

より複雑なプロジェクトを管理するために、バージョンとリビジョンを管理する制御システムをSDL Suiteに統合することができます。このようなシステムには、RCSやCM SYNERGY、ClearCaseなどがあります。オーガナイザをカスタマイズして、メニュー コマンドを拡張すれば「チェックイン」、「チェックアウト」、「更新」の処理がサポートされるようになります。メニューの拡張はテキスト ファイルに定義します。定義には、メニューとコマンドの名前、コントロール システムが呼び出す処理、処理の対象などを含めることが可能です。

この章の以降の項の構成を以下に示します。

- 152ページの「ダイアグラムのバインディング」では、ダイアグラムとその他のドキュメントをファイルに関連付ける仕組みについて説明します。
- 155ページの「プロジェクトでのダイアグラム管理」では、プロジェクトの環境でツールを使用してSDLダイアグラムを管理する方法について説明します。

ダイアグラムのバインディング

オーガナイザでは、外部のバージョンやリビジョン コントロール システムを使用しなくても、プロジェクトで開発中のシステムを構成するダイアグラムやその他のドキュメントを管理できます。

オーガナイザのドキュメント管理において重要な点は、各ダイアグラムは単一のファイルに格納され、ダイアグラムとファイルの間の割り当ては*システム ファイル*に保存されることです。システム ファイルは、オーガナイザで[開く（*Open*）]コマンドを実行した際に開くファイルです。システム ファイルのデフォルトの拡張子は`.sdt`です。

オーガナイザでは、以下の方法によってダイアグラムを物理ファイルに接続できます。

- 自動バインディング
- 手動バインディング

自動バインディング

ダイアグラムは、最初に保存するときに自動的に接続されるように設定できます。この場合、物理ファイルの名前は既定で、`<diagram name>.<ext>`となります。*<diagram name>*はダイアグラムの名前を表し、*<ext>*は物理ファイル内に含まれるダイアグラムのタイプを示す3文字の拡張子を表します。既定のファイル名と拡張子については、『User´s Manual』の第1章「User Interface and Basic Operations」の11ページ、「Save」を参照してください。

例49 ———————————————————————————————

DemonBlockという名前を持つブロック ダイアグラムは、既定で`demonblock.sbk`というファイルに接続されます。

———————————————————————————————

手動バインディング

[接続（*Connect*）]コマンドを使用すれば、ダイアグラムとファイルを必要に応じて手動でバインドできます。[接続]コマンドを実行すると、ファイルの選択するダイアログ ボックスが表示され、選択したダイアグラムを既存のファイルに接続できます。また、ディレクトリ内でダイアグラム ファイルを探すことも可能です。

図54: 既存のファイルへのダイアグラムの接続

ダイアグラムを接続する際に、[サブストラクチャの展開]オプションをオンにすれば、サブ構造内のすべてのダイアグラムが自動的に接続されます。この機能を使用して自動的にファイルを割り当てる場合、ダイアグラム名がファイル内に格納されるのでファイル名の管理はさほど重要ではありません。ただし、ファイル名の拡張子の管理は、ファイル内に格納されているダイアグラムのタイプに対応しなければならないため重要です。

ソースとターゲット ディレクトリ

オーガナイザには、ダイアグラムの格納場所に関する以下の2つの設定があります。

- *ソース ディレクトリ*
  新しいダイアグラムが保存されるディレクトリです。
- *ターゲット ディレクトリ*
  生成されるファイルが保存されるディレクトリです。生成されるファイルにはソース ファイル、すなわちオーガナイザのダイアグラム構造の中にあるファイルは含まれません。

オーガナイザのソース ディレクトリとターゲット ディレクトリは、[ディレクトリの設定（*Set Directories*）]コマンドを使用して変更できます。
[ディレクトリの設定]ダイアログ ボックスには、ファイルの絶対パスまたはソース ディレクトリからの相対パスのいずれを表示するか指示するオプションもあります。このコマンドに関する詳細は、『User´s Manual』の第2章「The Organizer」の70ページ、「Set Directories」を参照してください。

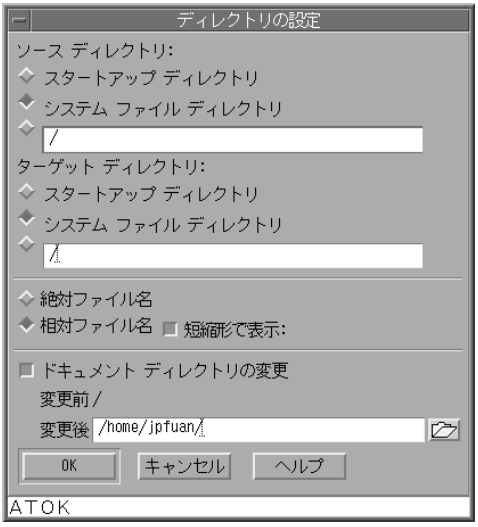


図55: ソースとターゲット ディレクトリの指定

以下の設定によって、システム ファイルへのファイルのパスの保存方法を指定できます。

- [絶対ファイル名]オプションをオンにすると、絶対パスがシステム ファイル内に保存されます。
- [相対ファイル名]オプションをオンにすると、相対パスのみがシステム ファイル内に保存されます。ソース ディレクトリを移動可能にする場合には、このオプションをオンにすることは重要です。

プロジェクトでのダイアグラム管理

プロジェクトでダイアグラムを管理する方法の1つに、RCS（Revision Control System）などの、外部の構造制御システムやバージョン コントロール システムを使用して、オリジナル エリアを管理する手法があります。RCSは、ほとんどのコンピュータ システムで利用できます。この管理方法では、プロジェクト メンバ全員が各自のワーク エリアを確保することができます。各ワーク エリアには単一のシステム ファイルを格納します。また、複数のシステム ファイルにシステムの異なる部分を定義する場合は、複数のシステム ファイルを格納することもあります。システム ファイル内のすべてのダイアグラムを、各ユーザーのワーク エリア内のファイルに接続します。ユーザーのワーク エリアのファイルとオリジナル エリアのファイルの関係は、外部の構造制御システムやバージョン コントロール システムによって管理されます。

この項の以降の部分では、バージョン コントロール システムとしてRCSを使用する方法について説明します。SDL Suite/RCSの統合に関する情報は、`<inst dir>¥examples¥cm¥win32¥rcs¥`あるいは`<inst dir>/examples/cm/unix/rcs/`にあるReadmeファイルをご覧ください。CM SYNERGYを使用する方法については、162ページの「SDLシステムでのCM SYNERGYの使用」を参照してください。RCSの代わりにClearCaseを使用する方法については、169ページの「SDLシステムでのClearCaseの使用」を参照してください。

バージョン管理の処理は以下の機能によって成り立っています。

- チェックインとチェックアウト コマンドを使ったRCSでのロック機能。
- コントロール ユニット ファイル（*Control Unit File*）を使ったオーガナイザ内での複数ユーザーのサポート。
- オーガナイザをRCSに接続するために使用する、SDL Suiteへの専用メニュー コマンドの追加。

> メモ：
>
> 以降の説明と手順は、UNIXシステムでRCSのUNIXバージョンを使用する場合にのみ適用されます。

SDLシステムでのRCS使用

以下の手順で、RCSが管理するファイルを既存のSDLシステムに導入します。

1. システムに関連する、すべてのファイルを、1つのディレクトリ階層に格納します。以降、このディレクトリ階層を *ワーク エリア*と呼びます。
2. オーガナイザのソース ディレクトリ（*Source Directory*）をワーク エリア ディレクトリに設定します。
3. 次に、システムの*オリジナル エリア*を作成し、RCSファイル データベースとして設定します。ワーク エリアとオリジナル エリアは、管理するダイアグラム システムに対して同じディレクトリ構造を持たなければなりません。オリジナル エリアには、リビジョン ファイルを保持するRCSディレクトリが作成されますが、現時点では双方のディレクトリの内容は空です。（ワーク エリアには、RCSディレクトリは作成されません。）

例50: DemonGameシステムのワーク エリアとオリジナル エリア ———

ワーク エリアには、以下の3つのディレクトリが作成されます。

- システムに関連するファイルのディレクトリ
- GameBlockとDemonBlockブロック用の2つのディレクトリ

```
demonblock      gameblock       system

demonblock:
demon.spr       demonblock.sbk

gameblock:
game.spr        gameblock.sbk   main.spr

system:
demongame.msc   demongame.ssy   systemlevel.msc
```

ファイル システム内の別の場所に作成されたDemon Gameシステムのオリジナル エリア内の構造を以下に示します。

```
RCS     demonblock      gameblock       system

demongame/RCS:

demongame/demonblock:
RCS

demongame/demonblock/RCS:
```

```
demongame/gameblock:
RCS

demongame/gameblock/RCS:

demongame/system:
RCS

demongame/system/RCS:
```

4. オリジナル エリアのルート ディレクトリを定義し（`setenv rcsroot <ディレクトリ名>`）、オーガナイザにRCSのメニュー セット（`$telelogic/sdt/examples/RCS/sdtrcs.mnu`を実行）をロードします。これにより、オーガナイザのメニュー バーに[RCS]という新しいメニューが表示されます。

   – オーガナイザの[RCS]メニューを設定する方法は、ダイアグラムやダイアグラム階層に対してチェックインとチェックアウトを実行するための1つの例です。この例で紹介するコマンドの設定は、それぞれのニーズに合わせて簡単に変更できます。

5. オーガナイザの[RCS]メニューから[Recursive Check In]または[Check In]コマンドを選択して、各ダイアグラム ファイルの最初のバージョンに対してチェックインの処理を実行します。

   – コマンドの結果を表示するには、オーガナイザでファイル アクセス権を有効にします（[表示]メニューの[表示オプション（*View Options*）]をクリックし、[ファイル アクセス権（*File access permissions*）]をオンにします）。

6. チェックインの処理が正常終了したら、オリジナル エリアにRCSファイルが保存されます。

例51: オリジナル エリアに格納されるファイル

```
RCS     demonblock      gameblock       system

RCS:

demonblock:
RCS

demonblock/RCS:
demon.spr,v             demonblock.sbk,v

gameblock:
RCS

gameblock/RCS:
game.spr,v      gameblock.sbk,v main.spr,v
```

```
system:
RCS

system/RCS:
demongame.msc,v          demongame.ssy,v
systemlevel.msc,v
```

マルチユーザー環境でのSDL SuiteとRCS使用

システムをグループで開発する場合、システムを分割し、システムの各部分にコントロール ユニット ファイル（*Control Unit File*）を割り当てると便利です。ここでDemonGameシステムの開発において、ひとりの開発者がDemonBlockを開発し、別の開発者がGameBlockを開発するものとします。この場合は、以下の要素を作成する必要があります。

- 最上位のコントロール ユニット ファイル
- DemonGameシステムのコントロール ユニット ファイル
- DemonBlockのコントロール ユニット ファイル
- GameBlockのコントロール ユニット ファイル

1. オーガナイザ オブジェクトにコントロール ユニット ファイルを割り当てるために、オブジェクトを選択して[Edit]メニューの[CMグループ（*Configuration > Group File*）]をクリックします。
2. 上記の4つのコントロール ユニット ファイルを割り当てたら、オーガナイザでシステムを保存します。これによって、コントロール ユニットも対応する`.scu`ファイルにそれぞれ保存されます。
3. 次に、生成された`.scu`ファイルに対してチェックインの処理を実行します。コントロール ユニット ファイルは、対応するオブジェクト ファイルと同じディレクトリに保存してください。

例52: コントロール ユニット ファイルが含まれるオリジナル エリア ——

コントロール ユニット ファイルに対してチェックインの処理を実行したDemonGameのオリジナル エリアは以下のようになります。

```
RCS              demonblock       gameblock
system

alfa/RCS:
demongame.scu,v

alfa/demonblock:
RCS

alfa/demonblock/RCS:
DemonBlock.scu,v         demon.spr,v
demonblock.sbk,v
```

```
alfa/gameblock:
RCS

alfa/gameblock/RCS:
GameBlock.scu,v game.spr,v      gameblock.sbk,v
main.spr,v

alfa/system:
RCS

alfa/system/RCS:
DemonGame.scu,v         demongame.ssy,v
demongame.msc,v         systemlevel.msc,v
```

RCSによるローカル変更のグローバル化

チェックアウトが終了しロックされたファイルに対して、ユーザーが変更を加え、さらに変更内容をプロジェクト全体に対してグローバルに反映する場合は、以下の操作を行います。

- 変更したダイアグラムに対して、オーガナイザの[RCS]メニューの[Check In]または[Recursive Check In]を実行します。[Recursive Check In]を実行すると、コントロール ユニット ファイルも自動的に処理されます。

例53: コントロール ユニットにダイアグラムを追加して保存するには — 新しいプロセス ダイアグラムをGameBlockに追加する場合に必要となる操作を以下に示します。

1. ユーザーは`gameblock.sbk`と`GameBlock.scu`ファイルに対してチェックアウトの処理を実行する必要があります。
2. SDL エディタを使って、GameBlockダイアグラムに新しいプロセス参照シンボルを追加し、更新します。
3. オーガナイザで保存すると、新しいプロセス ダイアグラムのファイル名で`GameBlock.scu`が自動的に更新されます。すべての変更は、GameBlockのダイアグラム システムに対してローカルに反映されます。そして、このダイアグラムに加えられた変更を、システム ファイルへのグローバルな更新として反映する必要はありません。

RCSによるグローバル変更のローカル化

システムに変更があったことを知らせる通知をユーザーが受信した場合は、以下の手順でローカル システムを更新する必要があります。

- [Check Out]または[Recursive Check Out]を使用してダイアグラムを更新します。
  - 例53の変更によってGameBlockが変更されている場合、GameBlockオブジェクトに対して[Recursive Check Out]を実行すると、最新のチェックイン バージョンを取得できます。

RCSベースのオリジナル エリアからのワーク エリア作成と配置

あるオリジナル エリアに、チェックインされたダイアグラムとコントロール ユニット ファイルを内容に持つRCSディレクトリが存在するとします。このとき、新たに参加する開発者が、自分のワーク エリアにシステムの完全な最新版を作成するための手順は、以下のようになります。

1. 開発者は、まず自分のワーク エリアのディレクトリ ツリーを作成または更新します。そして`rcsroot`環境変数を定義して、その環境変数にオリジナル エリアの位置を指定します。

2. 何も開いていないオーガナイザで、[RCS]メニューをロードし（たとえば、`$telelogic/sdt/examples/RCS/sdtrcs.mnu`コマンドを実行します）、ソース ディレクトリ（*Source Directory*）をワーク エリアに設定して、最上位のコントロール ユニット ファイルをSDTシンボルに関連付けます（オーガナイザの[CMグループ（*Configuration > Group File*）]コマンドを使用します）。

   – 最上位のコントロール ユニット ファイルには、チェックインされた正しいファイル名を割り当てる必要があります。DemonGameの場合、最上位のコントロール ユニット ファイルの名前は、ワーク エリア内の最上位ディレクトリにある`DemonGame.scu`になります。

3. ワーク エリアへの配置と更新を行うために、オーガナイザでSDTシンボルを選択し、[Recursive Check Out]をクリックします。

4. システム ファイルを保存します。これによりダイアグラム システムの個人用ビューを利用できるようになります。

   – システム ファイルを保存する際、オーガナイザは、最上位のコントロール ユニット ファイルの保存を試行します。このファイルは読み出し専用なのでオーガナイザはエラー メッセージを表示しますが、この表示は無視してかまいません。

RCSによるエンドポイント処理

ここでは、システムのリンク情報を保持するグローバル リンク ファイル（*マスター リンク ファイル*）が1つある場合を考えます。開発者グループは、RCSのチェックアウト機能とロック機能を利用して、常時ひとりの開発者だけにファイルへの更新を許可するように設定することができ、このファイルに変更の整合性を取ることができます。*ローカル リンク ファイル*は、ユーザーがエンドポイントやリンク情報に加えた変更を一時的に保存するための領域です。これらの変更は、グローバルな変更として更新するまでの間、ローカル リンク ファイルに保存されます。リンク ファイルの最初のバージョンをオリジナル エリアにチェックインした後で、リンク ファイルを変更する場合の手順を、以下に示します。

1. [RCS]メニューの[Add Local Link File]をクリックします。
2. エンドポイントとリンク情報へのすべての変更がローカル リンク ファイルに保存されます。また、マスター リンク ファイルは未変更のまま残ります。
3. グローバル リンク情報を更新する際に、[Check Out And Lock Link File]をクリックしてマスター リンク ファイルに対するチェック アウト処理を実行します。
4. [Merge Local Link File]をクリックして、ローカル リンク ファイルの情報でマスター リンク ファイルを更新します。
5. [Check In Link File]をクリックして、マスター リンク ファイルに対するチェックイン処理を実行します。

SDLダイアグラムの同時編集

オーガナイザの[ツール]メニューにある[分割（*Split*）]や[結合（*Join*）]コマンドを使うと、複数のユーザーが同じSDLダイアグラムに対して同時に作業できるように設定することができます。

[分割]コマンドを使うと、複数のページで構成されているダイアグラムを、ページ ごとに分割して複数のファイルに保存することができます。これによって、複数のユーザーがそれらのファイルを個別に編集できます。また、[結合]を使用すれば、同じ種類の2つのSDLダイアグラムを結合することができます。

SDLシステムでのCM SYNERGYの使用

ここでは、統合の手段としてSDL SuiteとCM SYNERGYを使用する代表的な方法について紹介します。SDL SuiteとCM SYNERGYの詳細については、Readmeファイルを参照してください。Readmeファイルは、
<SDLスイートのインストール先ディレクトリ>¥examples¥cm¥win32¥cmsynergy¥
または
<SDL スイートのインストール先ディレクトリ>/examples/cm/unix/cmsynergy/にあります。

設定またはバージョン コントロール システムの全般的な使用法の説明については、155ページの「プロジェクトでのダイアグラム管理」を参照してください。

SDL SuiteへのCM SYNERGYの導入 - 移行

CM SYNERGYでファイルを操作する場合は、SDl Suiteで開発中のシステムに適用します。この場合、次の方法を使用できます。

1. システム関連のすべてのファイルがCM SYNERGY外の1つのディレクトリに設定されていることを確認します。

2. CM SYNERGYのメニューをオーガナイザにインストールします。次に、cmsynergy.iniをorg-menus.iniファイルに追加します。
   SDL Suiteは、まずSDL Suiteを開始したディレクトリで、org-menus.iniを検索します。次に、HOME環境変数によって示されたディレクトリを検索し、最後に、SDL Suiteがインストールされたディレクトリを検索します。org-menus.iniファイルがない場合は、cmsynergy.iniファイルを使用できます。この場合、cmsynergy.iniファイルをHOMEディレクトリまたはSDL Suiteがインストールされているディレクトリにコピーし、ファイル名を「org-menus.ini」に変更します。動的なメニューの詳細については、第1章「User Interface and Basic Operations」18ページの「Defining Menus in the SDL Suite」を参照してください。SDL Suiteに含まれるCM SYNERGYのメニューは一例であり、ユーザーがカスタマイズすることができます。

3. 次の情報を含む、パスの環境変数を設定します。

   – <CM SYNERGYのインストール先ディレクトリ>¥binまたは<CM SYNERGYのインストール先ディレクトリ>/bin、そして
   – <SDL スイートのインストール先ディレクトリ>¥bin¥wini386または<SDL スイートのインストール先ディレクトリ>/bin。

   それぞれのディレクトリからツールを開始できるようにするために、この処理が必要です。

4. CM SYNERGYクライアントを開始します。

5. ccm_admin へのロールを変更します。

6. SDLファイルのタイプを定義するために、［Admin］から［Type Definition］ダイアログを表示します。次の値を入力します。

   Type Name: SDL
   Description: Binary file
   Super Type: binary
   Initial Status: working
   Require Task at:<none>

   Verify Comment Existence on Promote / Check In: Off
   Allow Update during Model Install: Off
   Associate with a File in the File System: On
   Can be a Product File: Off

   Icon Color / Fil: binary.bmp
   File Name Extension: .ssy

7. ［Type］から［Modify File Operations］を選択します。表示されたダイアログで次の値を入力します。

   (必要であれば以下のsdtをSDL スイートを開始するコマンドに置き換えます。sdtスクリプトがパスにない場合は、完全なパスを指定する必要があります。)

   Type Name: SDL
   Description: Binary file
   Command Templates:
   Graphical User Interface:

   Edit: sdt %ファイル1

View: sdt %ファイル1
Compare: sdt -fg -grdiff %ファイル1 %ファイル2
Merge: sdt -fg -grdiff %ファイル1 %ファイル2 -mergeto %出力ファイル

Command line interface:
Edit: sdt %ファイル1
View: sdt %ファイル1
Compare: sdt -fg -grdiff %ファイル1 %ファイル2
Merge: sdt -fg -grdiff %ファイル1 %ファイル2 -mergeto %出力ファイル

Print:
Compare Attribute: source

8. ［OK］をクリックします。
9. ［Update Type］をクリックします。
10. 次の値を入力し、プロジェクト ファイルのタイプ（*.sdt）を定義します。

Type Name: SDT
Description: SDT project file
Super Type: ascii
Initial Status: working
Require Task at:<none>

Verify Comment Existence on Promote / Check In: Off
Allow Update during Model Install: Off
Associate with a File in the File System: On
Can be a Product File: Off

Icon Color / Fil: ascii.bmp
File Name Extension: .sdt

11. ［Type］から［Modify File Operations］を選択します。表示されたダイアログを使用して次の値を入力します。

（必要であれば以下のsdtをSDL Suiteを開始するコマンドに置き換えます。sdtスクリプトがパスに無い場合は、完全なパスを指定する必要があります。）

Type Name: SDT
Description: SDT project file

Command Templates:
Graphical User Interface:

Edit: sdt %ファイル1
View: sdt %ファイル1
Compare: %FAIL
Merge: %FAIL

Command line interface:
Edit: sdt %ファイル1
View: sdt %ファイル1
Compare: %FAIL
Merge: %FAIL

Print:
Compare Attribute: source

12. ［OK］をクリックします。
13. ［Update Type］をクリックします。
14. ダイアログを閉じます。(［File］メニューから［Close］をクリックします。)

    これで、SDL システム ダイアログのCM SYNERGYとSDTプロジェクト ファイルのセットアップが完了しました。

15. ほかのSDLダイアグラムのタイプを管理するには、メニューから［ツール］、［移行］、［オプション］、［設定］、［編集］の順にクリックし、テキスト エディタで移行規則ファイル（migrate.rul）を開きます。
16. 新しいタイプのエントリのあとに次のような行があることを確認します。(プラット フォームにより違いがでる可能性があります)。

```
MAP_FILE_TO_TYPE .* [Ss][Ss][Yy]$ SDL # Created
automatically ...
MAP_FILE_TO_TYPE .* [Ss][Dd][Tt]$ SDT # Created
automatically ...
```

以下の行を追加し、このファイルを保存します。

```
MAP_FILE_TO_TYPE .* [Ss][Bb][Kk]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Pp][Rr]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Pp][Dd]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Uu][Nn]$ SDL
```

```
MAP_FILE_TO_TYPE .* [Ss][Oo][Pp]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Ss][Tt]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Bb][Tt]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Ss][Uu]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Pp][Tt]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Vv][Tt]$ SDL
MAP_FILE_TO_TYPE .* [Ss][Ss][Vv]$ SDL
MAP_FILE_TO_TYPE .* [Mm][Ss][Cc]$ SDL
```

このファイルでは構文エラーが起こりやすいので、構文には注意してください。開いているダイアログを閉じてください。

17. ［ツール］メニューから［移行］を選択し、"Migrate "ファシリティを使用してファイルをCM SYNERGYにロードします。

    – ［From Directory］フィールドにあるProjectディレクトリへのパスを入力するか、または［参照］ボタンをクリックしてディレクトリを探します。

    – ［Project］フィールドでProjectName-Versionを構文として使用し、バージョン名に意味のある値が設定されていることを確認します。

    – 次に、メニューから［オプション］、［設定］、［Set Object State To］、［released］の順にクリックし、［OK］をクリックします。

    – Windowsを使用している場合、ファイルを変更するには、［Work Area Properties］にある［Make Copies Modifiable］オプションを設定して［OK］をクリックします。

    – 正しいファイルが移行されているかどうかを確認するために、［Preview］をクリックすることをお勧めします。ファイルに誤りがなければ先に進み、［Load］をクリックします。

    これで、リリース状態をベースラインとするプロジェクトが作成され、作業を開始できるようになります。

18. 各開発者は、SDL Suite and TTCN SuiteでCM SYNERGYを使用する前に、自分の作業環境をセットアップする必要があります。

SDL SuiteへのCM SYNERGYの導入 - 作業環境のセットアップ

ビルド マネージャによってSDLシステムをCM SYNERGYに移行した後は、開発者は自分専用のCM SYNERGY作業領域をセットアップする必要があります。これを行うには、CM SYNERGYクライアントを使用し、選択されているオプションがすべて正しいものであることを確認します。

1. 1. SDLシステムのプロジェクトの正しいベースラインを選択します。ベースラインについては、ビルド マネージャの指示に従ってください。
2. ビルド マネージャから指示された正しいオプションを使用し、自分の個人用作業バージョンのプロジェクト階層をチェック アウトします。次に、システム ファイルをチェック アウトします。このファイルは個人用バージョンとなります。

> メモ：
>
> ソフトウェアの主要なリリース時にしなければならないことは、通常、この方法で作業環境をセットアップするだけです。

3. CM SYNERGYメニューをオーガナイザにロードします<org-menus.ini>。

SDL SuiteへのCM SYNERGYの導入 - CM SYNERGYでの日常作業

以下の手順は、推奨のCM SYNERGYタスク ベースの CMメソドロジを使用することを前提としています。これらはオーガナイザから実行できます。

1. SDLシステムを選択し、オーガナイザから CM SYNERGYを開始します。
2. デフォルトのタスク（現在のタスクであることがほとんど）を設定します。必要に応じてタスクを作成します。
3. 作業ファイルをチェック アウトします。
4. 必要に応じて編集およびテストを行います。
5. （デフォルトの）タスクをチェック インします。

> メモ：
>
> チームの一員として作業している場合は、トップ レベルのディレクトリを選択し、Updateコマンドを使えば、別メンバの最新の作業内容を確認することができます。（場合によっては、CM SYNERGYクライアントを使用して、完全な更新処理を行う必要があることがあります。これについては、ビルドマネージャに問い合わせてください。）

SDLシステムでのClearCaseの使用

ここでは、SDL SuiteとClearCaseを統合的に使用するための方法について説明します。SDL Suite/ClearCaseの統合に関する情報は、
`<inst dir>¥examples¥cm¥win32¥clearcase¥`あるいは
`<inst dir>/examples/cm/unix/clearcase/`にあるReadmeファイルをご覧ください。構造制御システムまたはバージョン コントロール システムの一般的な使用方法については、SDL SuiteへのClearCase導入–ファイルのチェックイン155ページの「プロジェクトでのダイアグラム管理」を参照してください。

SDL SuiteへのClearCaseの導入 - ファイルのチェックイン

SDL Suiteで開発しているシステムにClearCaseを使ったファイル管理の導入手順について、以下に示します。

1. 管理対象となるシステムに関連するすべてのファイルが、ClearCaseの外部で、単一のディレクトリ階層によって構成されていることを確認します。156ページの例50のワーク エリアに関する記述を参照してください。

   – ClearCaseで作業をするときは、オリジナル エリアとワーク エリアは同じダイアグラム システムの最上位ディレクトリを指します。

2. ClearCaseメニューをオーガナイザにインストールします。次に、clearcase.iniをorg-menus.iniファイルに追加します。SDL Suiteは、まずSDL Suiteを開始したディレクトリで、org-menus.iniを検索します。次に、HOME環境変数によって示されたディレクトリを検索し、最後に、SDL Suiteがインストールされたディレクトリを検索します。org-menus.iniファイルがない場合は、clearcase.ini ファイルを使用できます。この場合、clearcase.ini ファイルをHOMEディレクトリとSDL Suiteがインストールされているディレクトリにコピーし、ファイル名を「org-menus.ini」に変更します。動的なメニューの詳細については、第1章「User Interface and Basic Operations」18ページの「Defining Menus in the SDL Suite」を参照してください。SDL Suiteに含まれるClearCaseのメニューは1例であり、ユーザーがカスタマイズすることができます。

3. ClearCaseビューを適切な設定に変更します。システム ファイルをClearCaseファイル システムの最上位ディレクトリにコピーします。必要に応じて、システム ファイルを編集し`SourceDirectory`の定義のある行を削除します。

4. [Open]をクリックしてシステム ファイルを開き、システムの構造ビューを表示します。オーガナイザは、ダイアグラムが無効であることを表示しますが、この時点ではこれで正常です。

5. [MkDir for Object]をクリックして、ClearCase VOBにディレクトリ構造を作成します。オーガナイザでオブジェクトを選択し、[MkDir for Object]をクリックすると、選択したオブジェクトに対応するディレクトリがClearCase VOB内に作成されます。

6. 次に、手順1.のディレクトリからダイアグラム ファイルをコピーし、ClearCaseのディレクトリ構造に保存します。

7. オーガナイザのシステム ファイルを再び開きます。すべてのダイアグラムがそれぞれのファイルに接続されているはずです。

8. [システム ファイル（*System File*）]アイコンをクリックし、[Recursive MkElem]をクリックして、ClearCase VOB内にすべてのオブジェクトを作成します。

   – システム ファイルに対して、ClearCase要素は作成されません。

9. [システム ファイル（*System File*）]アイコンをクリックし、[Recursive Check In]をクリックして、すべてのオブジェクトをClearCase VOBにチェックインします。

10. 各ディレクトリを個別にチェックインします。この処理には、[Check In Directory]を使用します。

ダイアグラム システムのシステム ファイルをチェックインすることもできますが、オーガナイザはリビジョンの変更に関係なくシステム ファイルを更新するので、バージョン コントロールには適していません。システム ファイルは、開発中のシステムに対する開発者の個人用ビューとして扱います。一方、最上位のコントロール ユニット ファイルはチェックイン処理が必要であり、リビジョン コントロールの対象としなければなりません。

SDL SuiteへのClearCaseの導入 - システムの起動

ユーザーが何もない状態から作業を始める場合、最上位のコントロール ユニット ファイルを使ってシステムをオーガナイザにロードすることができます。ClearCase VOB内にチェックインされたダイアグラム システムがある場合、ユーザーは、ダイアグラム システムに対する作業を以下の手順で始める必要があります。

1. ダイアグラム システムが保存されているClearCase VOBに移動します。
2. ClearCaseビューを適切に設定し、新しいシステムに対してSDL Suiteを起動します。
3. `$telelogic/sdt/examples/ClearCase/sdtcc.mnu`を使用して、オーガナイザに[ClearCase]メニューをロードします。
4. ソース ディレクトリ（*Source Directory*）を、対象のダイアグラム システムの最上位ディレクトリに設定します。
5. [接続（*Connect*）]をクリックして、[システム ファイル（*System File*）]アイコンと最上位のコントロール ユニット ファイルを接続します。次に、[Recursive Update ClearCase]をクリックします。これにより、オーガナイザに対象のシステムがロードされます。
6. システム ファイルを*保存*します（オーガナイザは、最上位のコントロール ユニット ファイルが読み出し専用であることを警告しますが、これは無視できます）。