ASCET V5.2

リファレンスガイド

著作権について

Document EC010001 R5.2.2 JP

目次

ASCET V5.2

モデリング言語

1 プロジェクト

ASCET において組み込みソフトウェアシステムは、１つの「プロジェクト」という単位で定義されます。プロジェクトには、少なくとも以下のものが含まれます。

- 複数のモジュール
- リアルタイムオペレーティングシステム用のタスクスケジューリング
- プロセス間通信の定義

プロジェクトの中心部にあるのは、オペレーティングシステムのタスクスケジューリングについての記述です。ここでは、システムの動的挙動が定義されています。図 1-1 は、プロジェクトの構造を示します。

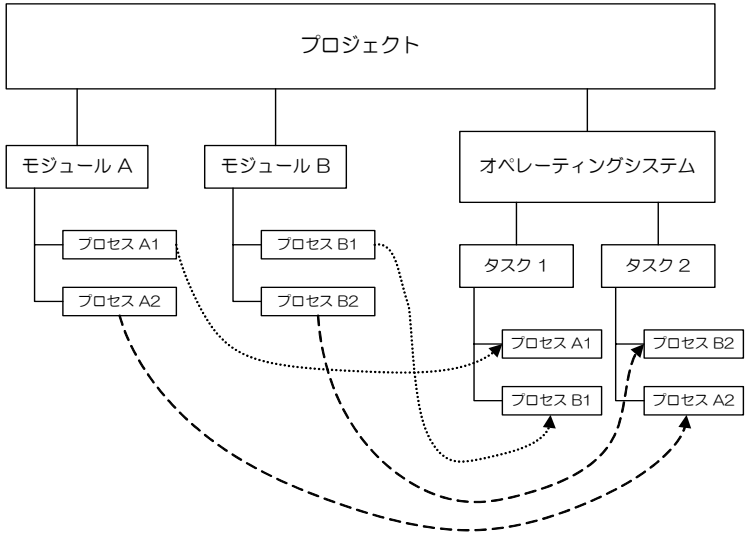


図 1-1 プロジェクトの構造

1.1 オペレーティングシステムのタスクスケジューリング

組み込み制御システムの基本部分は、基礎を成すリアルタイムオペレーティングシステムで、これが各種のアルゴリズムや計算の実行を制御します。ASCET にはこのための専用のエディタがあり、このエディタを使えば、オペレーティングシステムのスケジューリングに関連するすべてのデータを設定することができます。

タスクのスケジューリングは、自動車用リアルタイムオペレーティングシステム ERCOS^{EK} によって行われます。各タスクから並行して発行される数多くのリクエスト（例、カム軸割込みや定周期サンプリングなど）に対応するために、優先度に基づく協調スケジューリングやプリエンプティブスケジューリングが、このオペレーティングシステムの中核になっています。このスケジューリングは、マルチタスク環境でのタスクの実行を制御し、各タスクは、一定の順序で実行されるプロセスのリストとして定義されます。各プロセスは制御アルゴリズムの 1 単位で、一定の周期において、または外部割込み発生時に実行されます。

制御システムにはさまざまなアルゴリズムが必要なため、プロセスの数は非常に多くなります。しかし同時に、これらプロセスの実際の動作の多くは類似しているものがあり、同じ動作を持つプロセスをタスクにまとめれば、オペレーティングシステムのオーバヘッドを抑え、アプリケーションの動作を構造化することができます。その理由により、同じ動作のプロセスは 1 つのタスクにまとめられます。

リアルタイムタスクスケジューリングは、以下の要素で構成されます。

- スケジューリング
- タスク
- プロセス
- アプリケーションモード

1.1.1 スケジューリング

オペレーティングシステムは、モジュール内に定義されているプロセスの実行をスケジューリングします。スケジューリングの定義は、各プロセスのシーケンスをいくつかのグループにまとめ、それらをタスクとしてオペレーティングシステムのタイムスケジュール内に定義することによって行われます。タスクは、指定されたモードでオペレーティングシステムにより起動され、レディ状態となります（例、タイマによる定期的起動、ソフトウェアや外部イベントによる起動など）。

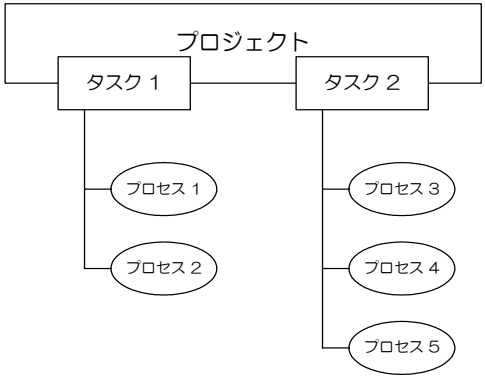


図 1-2　プロセスをタスクにまとめる

図 1-2 は、2 つのタスクに割り当てられているプロセスを示しています。ここで、Task1 は 10ms ごとに起動され、20ms ごとに起動される Task2 よりも優先度が高くなっているとします。また各プロセスの実行時間は、p1 = 2ms、p2 = 1ms、p3 = 2ms、p4 = 1ms、p5 = 1ms であるとした場合、スケジューリングは以下のようになります。

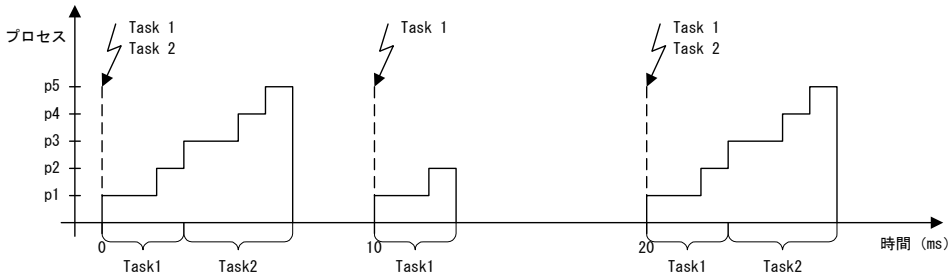


図 1-3　単純なタスクスケジューリング

オペレーティングシステムは 3 種類のスケジューリングをサポートしています。「**協調スケジューリング**」では、現在のプロセスは、それより高い優先度のタスクが起動されてもすぐにはプリエンプト（中断）されません。新しいタスクは、現在のプロセスが終了してから実際に処理を開始します。それまで実行されていたタスクにまだ実行すべきプロセスが残っている場合は、そのタスクはレディ状態のまま待機し、優先度の高いタスクの処理が完了した時点で保留されていたタスクが続行（レジューム）されます。各プロセスの実行時間が p1 = 2ms、p2 = 1ms、p3 = 5ms、p4 = 4ms、p5 = 2ms であるとした場合、このスケジューリングは図 1-4 のようになります。

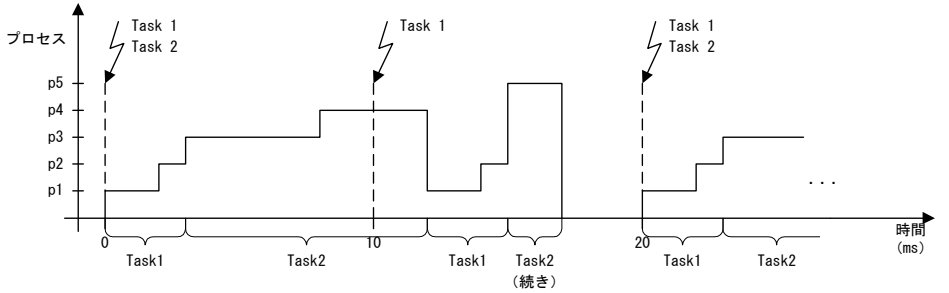


図 1-4　プロセス（p4）終了時にプリエンプトされたタスクのレジューム（協調スケジューリングの場合）

「**プリエンプティブスケジューリング**」では、現在のプロセスは、それより高い優先度を持つタスクが起動されると、すぐに中断（プリエンプト）されます。図 1-8 のように、すべての協調タスクの優先度レベルは他のプリエンプティブタスクやノンプリエンプタブルタスクよりも低いので、協調タスクによってプリエンプティブタスクがプリエンプトされることはありません。中断されたプロセスの実

行は、プリエンプトしたタスクの処理が完了した時点で続行（レジューム）されます。各プロセスの実行時間が前出の例と同じである場合、プリエンプティブスケジューリングを行うと、図 1-5 のようになります。

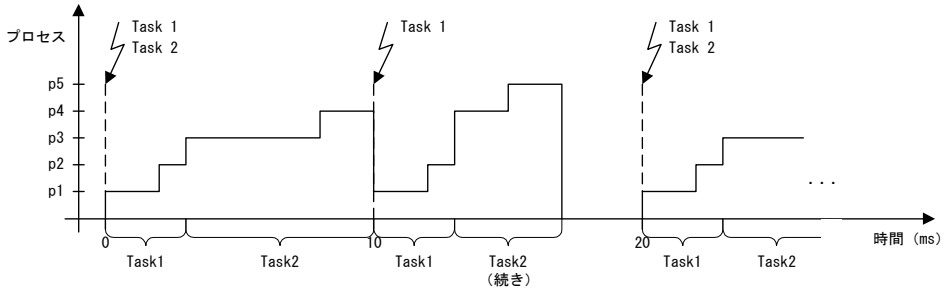


図 1-5　プロセス（p4）実行時にプリエンプトされたタスクのレジューム（プリエンプティブスケジューリングの場合）

「**ノンプリエンプタブルスケジューリング**」（マイクロコントローラターゲットの場合のみ使用可能）では、現在実行中のノンプリエンプタブルタスクより高い優先度のタスクが起動されても、現在実行中のプロセスやタスクはプリエンプトされません。起動されてレディになったタスクは、ノンプリエンプタブルタスクが終了した時に実行されます。図 1-6 は、上の図と同じプロセスを持つ 2 つのタスクがノンプリエンプタブルスケジューリングによって実行される様子を示したものです。

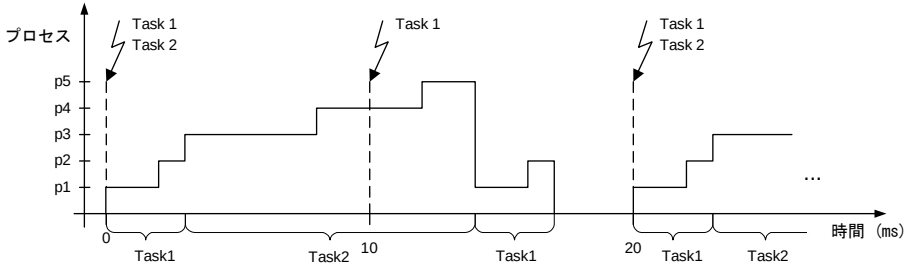


図 1-6　ノンプリエンプタブルタスクのスケジューリング

注記

ノンプリエンプタブルタスクは、OSEK の仕様にあわせて ERCOS^{EK} に導入されたものですが、このタスクの使用はお勧めしません。ほとんどの処理は協調タスクとプリエンプティブタスクのみで実現できる上、ノンプリエンプタブルタスクのコンフィギュレーション設定は、優先度領域の境界の条件に関する点で非常に複雑となります。

1.1.2 タスク

タスクには、そのタスクが起動されることによって実行されるプロセスのリストが含まれています。プロセスの実行順序は固定されています。タスクがオペレーティングシステムのスケジューラによりスケジューリングされる方法は、タスク設定により定義されます。タスクモードの種類は以下のとおりです。

- *Alarm* タスク：定期的に起動されます。周期は秒単位で指定されます。
- *Timetable* タスク（マイクロコントローラターゲットでのみ使用可能）：タイムテーブルに書き込まれるアラームタスクで、実行時間を短縮できます（ただしメモリ消費量は増大します）。
- *Interrupt* タスク：外部イベントにより起動されます。プロセッサに応じて異なる種類のイベントが使用されます。イベントのリストから、適切なイベントを選択することができます。
- *Software* タスク：オペレーティングシステムルーチンを呼び出すことにより起動されます。つまり、ソフトウェアを通じて直接起動されます。
- *Init* タスク：オペレーティングシステムの稼働開始前に一度だけ起動されます。初期タスクの内容は、システムの初期化を行うためのコードです。

各タスクはさらに、協調、プリエンプティブ、ノンプリエンプタブルという 3 つのスケジューリンググループのいずれかに割り当てられ、各グループ内で優先度レベルが割り当てられます。各スケジューリンググループに設定できる優先度レベルの数はユーザーが定義することができ、この数によりスケジューラテーブルの所要メモリが決まるため、最終システムに合わせて調整する必要があります。

実行中のタスクがノンプリエンプタブルタスクではない場合、そのタスクよりも優先度が高いタスクは、実行中のタスクに割り込むことができます。割り込む方のタスクがプリエンプティブなスケジューリンググループに属する場合には実行中のタスクはすぐに中断され、そうでない場合には、現在のプロセスが終了してから中断されます。プリエンプティブおよびノンプリエンプタブルタスクは、協調タスクよりも常に高い優先度を持ちます。優先度機構は図 1-8 に示されているとおりですが、実際に使用できるタスクの数は、ターゲットによって異なります。

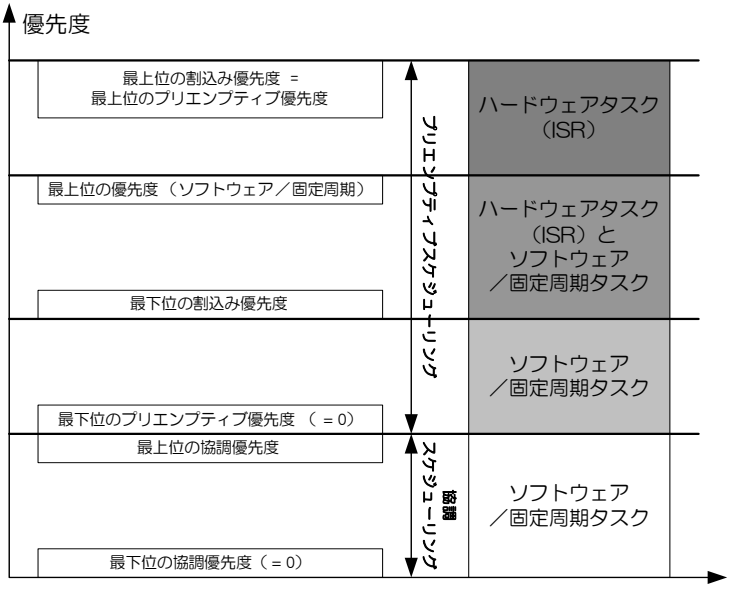


Fig. 1-7 優先度機構

タスクが実行されるたびに、前回の実行開始時からの経過時間がグローバル変数 dT に格納されます。この変数を利用することにより、実行周期に依存しない制御アルゴリズムを記述することができます。

1.1.3 プロセス

タスクは、プロセスのシーケンスとして構築されるものです。プロセスの内容はプログラムの実行コードで、各プロセスのボディが順を追って実行されます。タスクはそれより優先度の高いタスクにより中断される（＝プリエンプトされる）可能性があるので、プロセスもその実行の途中で中断される可能性があります。そこで、プロセスは並行して実行できるように設計する必要があります。

プリエンプティブなシステムを設計する場合、主に問題となるのはデータの整合性です。オペレーティングシステムは、プロセスでの計算結果が入力変数の値だけに依存し、システムでの実行順序には依存しないことを保証する必要があります。

この問題を解決するために、プロセスでは ERCOS^{EK} の「メッセージ」の概念をサポートしています。ERCOS^{EK} オペレーティングシステムにおけるメッセージは、保護されたグローバル変数です。つまり、実際の処理はグローバル変数のコピーを使って行われます。システムはコピーが必要かどうかを分析し、ランタイムのリソースを犠牲にすることなく最適なデータ整合を行います。

1.1.4 アプリケーションモード

アプリケーションモードはオペレーティングシステム ERCOS^{EK} の独自の機能です。プロセッサの実行時の負荷を低く抑えるために、オペレーティングシステムをさまざまなモードで稼働させることができます。一般的には、通常運転モード、EEPROM プログラミングモードといったモードが使用されます。これらのモードは相互に排他的であるため、一度に 1 つのモードだけが有効となり、常に現在のモードに関連するタスクだけが実行されます。

各タスクにはアプリケーションモードが割り当てられます。アプリケーションモードの切替えはソフトウェアにより行われ、モードが切り替わると、そのアプリケーションモードに割り当てられている Init タスクが起動されます。

注記

アプリケーションモードの切替えは、オペレーティングシステムのサービスコールによって行われます。詳しい内容は、ERCOS^{EK} のマニュアルを参照してください。

1.2 モジュールとプロセス

タスクに割り当てられるプロセスは、モジュール内に定義されます。モジュールは、関連性のある複数のプロセス（例、ラムダ制御ファンクションに属するいくつかのプロセス）をカプセル化したものです。1 つの制御アルゴリズムは、各部分をさまざまなタイミングで処理する必要があるため、モジュール内に記述されている機能をいくつかのプロセスに分割します。それにより、たとえばアルゴリズム中の最も重要な部分を最も高い頻度で実行する、といったことが可能になり、制御アルゴリズムの実行時間を大幅に効率化することができます。また同時に、アルゴリズムが整理されるので、開発、保守、および理解がしやすくなります。

複雑な制御タスクの機能は、複数のモジュールに分割して階層的に設計することができます。さらに、サブアルゴリズムやサービスルーチン（例、アキュムレータ、PI 制御など）をクラスやステートマシンを使用して整理することができます。

モジュールは、プロジェクト内の最上位レベルのコンポーネントで、プロジェクトにより排他的に使用されます。通常、モジュールはプロジェクトのユニークな部分（例、ラムダ制御）について記述するために使用されます。そのため、他のコンポーネント（例、アキュムレータ）とは異なり、各モジュールのインスタンスはプロジェクト内に 1 つしか存在しません。

モジュールには、他のすべてのコンポーネントと同様に「インターフェース」があります。モジュールのインターフェースには、プロセスと、データ交換に使用されるメッセージが含まれます。

1.3 プロセス間通信

プロセス間通信はメッセージを通じて行われます。メッセージは ERCOS^{EK} における保護されたグローバル変数で、データの整合性は、グローバル変数のコピーを処理することによって実現されます。

図 1-8 は、プリエンプティブなシステムでデータの不整合が発生する様子を示しています。この不整合を避けるために、メッセージを使ってプロセス間通信を設計します。プロセスは、初めにすべての入力メッセージ（読み取り専用のメッセージ）を受信します。メッセージを受信すると、そのメッセージの一時的なコピーが自動的に作成され、プロセスはそれを使って処理を行います。そしてプロセスの最後に、処理中に書き込みが行われた一時メッセージがすべてオリジナルのメッセージに書き戻されます。このしくみにより、変数の値は、プロセスが直接その値を変更しない限り、そのプロセス内で変更されることはありません。

保護されているグローバル変数、つまり「ステートメッセージ」をプロセス間通信に使用することは、組み込み制御システムに適しているといえます。第 1 の理由は、メッセージを送信するプロセスと受信するプロセスの間には依存関係がないために、複雑で実行時間のかかるデータの同期処理が必要ありません。また第 2 の理由としては、メッセージの送信側と受信側に 1 対 1 の関係がなく、1 つのメッセージを複数のプロセスが受信することができます。

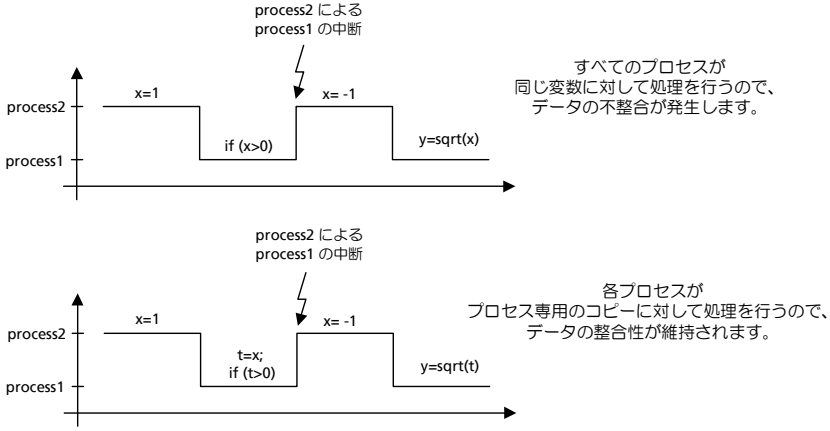


図 1-8　プリエンプティブなシステムにおけるデータの不整合

このメッセージのしくみは、ERCOS^{EK} のメッセージの原理に基づいています。ERCOS^{EK} の開発環境には、オフラインのシステム最適化機能があり、この機能により最適化されたメッセージ実装が実現されます。この機能では、データの整合性が脅かされる場合にのみコピーが作られ、通常はタスクの始めと終わりにしか作られません。

プロセス間通信はプロジェクト内で解決されます。同じ名前のメッセージ同士が結びつけられ、1 つのメッセージとして扱われます。たとえば、2 つのプロセスが `velocity` というメッセージを使う場合、それらのプロセスはこの変数の書き込みと読み取りを通じて通信を行います。名前をベースとして結びつきが決定されるこの方法は、グローバル変数やグローバルパラメータなど、他のグローバルオブジェクトについても適用されます。

2 コンポーネント

プロジェクトは、ASCET における組み込み制御システムを定義する最上位レベルの単位です。ここでは、アプリケーションの枠組みが定義され、その実行が制御されます。プロジェクトは、組み込み制御システムの「ブレーン」となるものです。

これに対し、コンポーネントは「ボディ」を形成する部品です。コンポーネント内で、実際の制御アルゴリズムやシステム内で実行されるその他の各種処理タスクの定義が行われます。

コンポーネントには「インターフェース」が含まれます。インターフェースは、コンポーネント内に記述されているアルゴリズムを、いつ、どのように実行するか、また他のコンポーネントとの間でどのようにデータを交換するか、ということを明確に定義するものです。

コンポーネントには「モジュール」と「クラス」の 2 種類があります。どちらもその設計上、中心となる概念はデータのカプセル化で、これは ASCET はオブジェクト指向の考え方に従っています。1 つのコンポーネントには数多くの「エレメント」が含まれ、そのモジュールまたはクラスで定義されているすべてのプロセスやメソッドがそれらのエレメントを使用することができます。エレメントのスコープを制限して「ローカル」にすることができ、メッセージについても、そのスコープをそのモジュールで定義されているプロセス内のみに制限することができます。

コンポーネントは、以下の要素で構成されます。

- コンポーネントの内容：コンポーネントが使用する変数、パラメータなどの宣言です。
- コンポーネントのインターフェース（プロセスまたはメソッドの形で）：クラスの内部変数やモジュール内で使用されるメッセージに直接アクセスできるように拡張することが可能です。
- アルゴリズム自体：プロセスまたはメソッド内の処理を指定します。

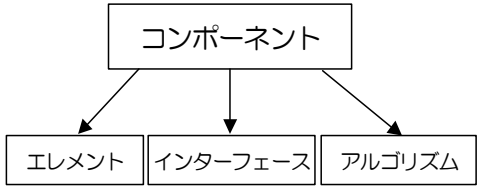


図 2-1　コンポーネントを構成するエレメント

以降では、モジュールとクラスの概要について説明し、続いてコンポーネントのインターフェースの構造について説明します。そして次にはアルゴリズムを記述するさまざまな方法（ブロックダイアグラム、テキスト、C コード）について説明し、本章の最後の部分で「ステートマシン」という特別なクラスについて説明します。この特別なクラス型は、必ずブロックダイアグラムを使って定義します。

2.1 モジュールとクラス

組み込み制御システムを作成する場合、システムのリアルタイム性に関する要件は非常に重要です。そのため ASCET では、リアルタイム処理に対応するインターフェースを備えた特別なコンポーネントである「モジュール」が使用されています。

モジュール内には複数のプロセスが定義され、さらにメソッドを定義することもできます。プロセスの中には、順次実行される一連のコードが含まれます。プロセスはオペレーティングシステムにより起動されるので、パラメータを受け取ることはできませんが、その代わりに、メッセージによるデータ交換、つまりグローバル変数空間への直接アクセスを行うことにより、非常に効率的な通信機構を実現しています。

モジュールは 1 つのプロジェクト内で一度しかインスタンス化できないので、1 つのモジュールはシステム内でただ 1 つしか存在できません。この制約を回避するためには、「クラス」を使用します。クラスはオブジェクト指向の抽象データ型で、データをカプセル化し、適切に定義されたインターフェースによるアクセスを可能にするものです。「インターフェース」は複数のメソッドで構成されます。これらのメソッドはプログラム内のどこからでも呼び出すことができるため、オペレーティングシステムからしか起動できないプロセスと比べ、はるかに柔軟性があります。各メソッドには任意の数の引数と 1 つの戻り値を持たせることができます。

クラスは 2 回以上インスタンス化することができます。たとえば、1 つのプロジェクトに同じアキュムレータクラスのインスタンスが 2 つ以上存在することができます。クラスの各インスタンスはそれぞれ専用のデータ空間（専用のパラメータと変数）を持ちますが、すべてのインスタンスは 1 つの機能記述を共用します。クラスに定義されているグローバル変数は、1 つのクラスのすべてのインスタンスについて同じものが作成されます（これは、オブジェクト指向の観点ではクラス変数と考えることができます）が、それらのグローバル変数には他のコンポーネントからもアクセスすることができます。

ただし、クラスは、メッセージによるリアルタイムのプロセス間通信を使用できません。これには 2 つの理由があります。第 1 に、クラスは複数のインスタンスを持つ可能性があり、ERCOS^{EK} のデータ整合処理ではこれを管理することができないためです。第 2 の理由は、プロセスは 1 つの固定タスクに静的に割り当てられる、という点です。つまり、プロセスが実行されるたびにオペレーションシステムはそのプロセスのすべてのメッセージをコピーを生成しますが、これらのコピーは、各コピーに対応するプロセスのインスタンスからしかアクセスできません。そのため、同じメッセージが複数のプロセスによって使用される場合、各プロセスはそれぞれ自分自身用にコピーされたメッセージを使用することになります。オペレーティングシステムがマルチプロセス処理におけるデータの一貫性を守ることができるようにするためには、この機構が必要です。

一方メソッドの場合は、プログラム内のさまざまなポイント、たとえば異なるタスクの異なるプロセスから自由に呼び出すことができるため、メソッド自身はどのタスクから呼ばれたのかを認識しません。このため、どのメソッド呼び出しにどのメッセージコピーが対応するかを判断することができません。

モジュールとクラスが持つ特性を、下の表 2-1 にまとめます。

プロパティ	モジュール	クラス
プロセス	○	
メソッド	○	○
引数渡し		○
メッセージ	○	
複数インスタンス		○
階層設計	○	○

表 2-1　モジュールとクラスのプロパティ

「ステートマシン」は、ASCET で使用される特殊なクラスです。その実際の機能はクラスと同じですが、記述方法が異なります。たとえば、ステートマシンには、状態遷移の条件を判定するための特別なメソッドがあります。

コンポーネントやモジュール、クラスを記述する場合、以前に記述した他のクラスやモジュールを再利用することになるので、通常その構造は階層構造になります。

2.2　コンポーネントの定義とインスタンス生成

各コンポーネントには 1 つの抽象データ型が定義され、それによって、そのコンポーネントが外部と相互作用するためのインターフェースが利用可能になります。プロジェクト内でコンポーネントを使用するには、「エレメント」を作成する必要があり、この「エレメント」に実際のメモリ領域が割り当てられます。コンポーネントのオブジェクト（実体）を作成する処理は「インスタンス生成」とも呼ばれ、インスタンス生成時には、必要なデータ構造体が構築され、初期化されます。

コンポーネントの各インスタンスは、それぞれ専用のエレメントセットを持ちますが、インターフェースと機能記述部分はコンポーネント自体から継承します。

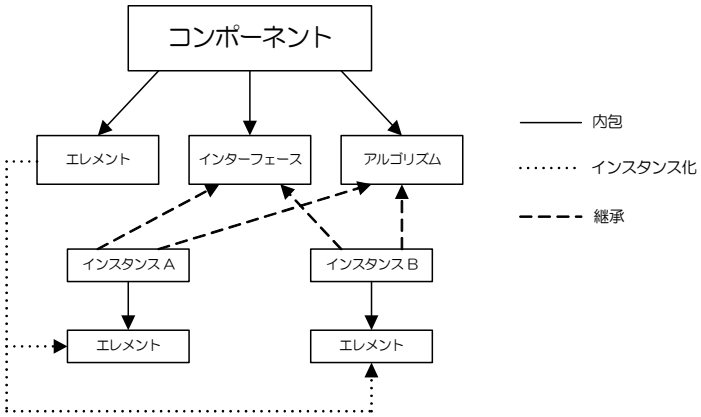


図 2-2　コンポーネントのインスタンス化と継承

したがって、1 つのコンポーネントを定義するということは、インスタンスとして生成されるコンポーネントのテンプレートを定義することを意味します。モジュールはプロジェクト内で 1 つの実体しか持たない（つまり一度しかインスタンス化されない）ので、モジュールのテンプレートとインスタンスが明確に区別されることはなく、両者の間には 1 対 1 の関係があります。

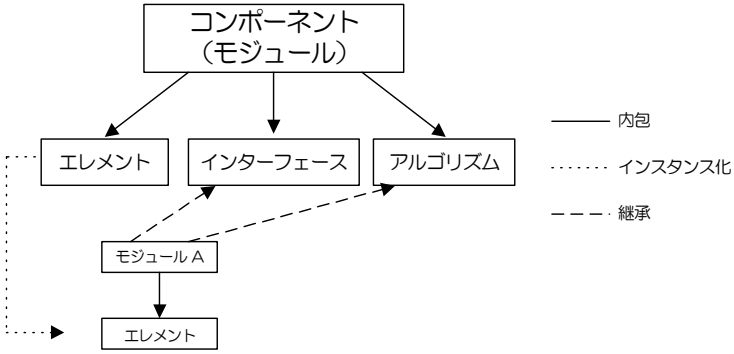


図 2-3　モジュールのインスタンス化と継承

一方、クラスは複数のインスタンスを持つことができます。クラスのテンプレートとインスタンスの間には 1 対 1 の関係がないので、「クラスの定義」を行うということと「インスタンス化」を行うということは明確に区別されます。クラスのテンプレートとインスタンスの関係は 1:n であり、クラスの定義とは、再利用可能なユーザー定義モデル型の定義を意味します。

コンポーネントのインスタンス化は、プロジェクトとして定義された単位の中においてのみ可能です。そのため、コンポーネントについて作業を行う際には、「デフォルトプロジェクト」が自動的に作成され、コンポーネントをインスタンス化するための記述が提供されます。

あるクラスを別のコンポーネントで使用する場合（次の項を参照してください）には、そのクラスはそのコンポーネントがインスタンス化されたときに、その「コンポーネント内」の記述に従ってインスタンス化されます。これとは対照的に、モジュールは必ず「プロジェクト内」でインスタンス化されます。

2.3 コンポーネントのインターフェース

コンポーネントのインターフェースには、メソッド、プロセス、およびグローバル変数へのアクセスが含まれます。たとえば、モジュールはメッセージへアクセスできます。メソッドとプロセスとは同じように構造化され、さまざまな形式で記述することができます。

メソッドやプロセスはそれぞれが 1 つのダイアグラムに割り当てられます。各ダイアグラムはパブリックまたはプライベートのいずれかです。プライベートダイアグラムに割り当てられたメソッドは、そのコンポーネントの中でだけ見ることができ、他のコンポーネントから見ることのできるパブリックインターフェースには属しません。1 つのダイアグラムに割り当てられたすべてのメソッドの記述は、そのダイアグラム内に書き込まれます（ブロックダイアグラムの場合、これらすべてのメソッド用の共通ブロックダイアグラムがあります）。

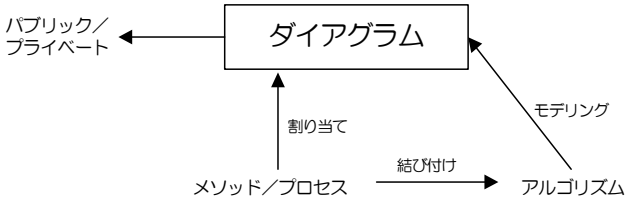


図 2-4 コンポーネントのインターフェース

2.3.1 クラスのインターフェース

クラスのインターフェースは、そのクラスの各ダイアグラムに割り当てられている複数のメソッドで構成されます。各メソッドのインターフェースは、その引数と戻り値です。メソッドはサブルーチンと似ていて、ソフトウェアのどこからでも呼び出すことができます。しかし、クラスのデータカプセル化、つまり、同じように定義されたインスタンス変数やパラメータへのアクセスにより、メソッドとクラスの概念の方がサブルーチンの概念よりもはるかに有力であると言えます。メソッドはそのクラスに定義されているすべてのエレメントにアクセスできます。

メソッドの引数と戻り値は、それらが定義されているメソッドのボディでしか使用できません。また、各メソッドにはそのメソッド固有の変数がいくつかあります。これらの変数は一時的なもので、静的ではなく、引数と同様に、それが定義されているメソッドのボディでしか使用できません。

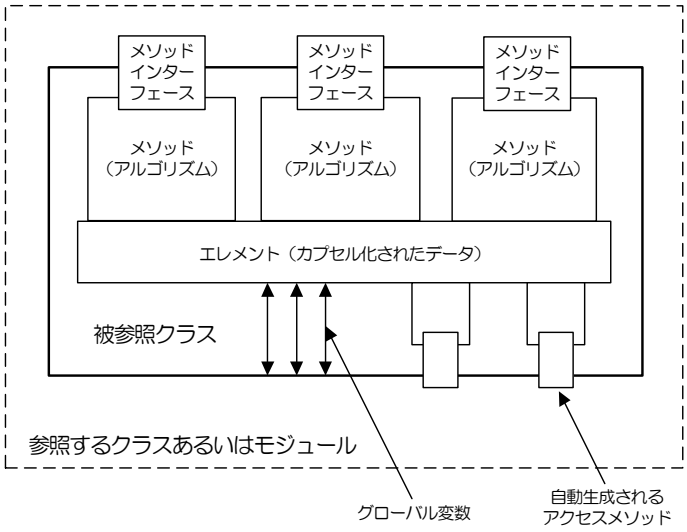


図 2-5　クラスのインターフェース

クラスのインスタンス変数に直接アクセスするために、専用のメソッドを使用することができます。このしくみにより、クラスをデータコンテナ（C のデータレコードと似ています）として使用することができます。

クラスとその外部とのデータ交換は、クラスメソッドの呼び出しにより行われます。メソッドが呼び出されると、そのメソッドのボディにある命令が実行されます。

各クラスメソッドは、パブリックダイアグラムとプライベートダイアグラムのいずれかに割り当てることで、パブリックまたはプライベートに分類されます。パブリックメソッドは、そのクラスを使用できるどのコンポーネントからでも呼び出すことができます。プライベートメソッドは隠蔽されていて、同じクラスのメソッドからしか呼び出すことができません。プライベートメソッドは内部サブルーチンとして使用されます。

2.3.2　モジュールのインターフェース

モジュールのインターフェースは、そのモジュールで使用される複数のプロセス（メソッドも可能）とメッセージで構成されます。「プロセスの起動」と「メッセージによる通信」は区別されるので、モジュール同士はこれら 2 つのレベルで干渉しあうことになります。プロセスの起動は、プロジェクト単位でオペレーティングシステムの管理のもとに行われます。

メッセージによるプロセス間通信は、プロセスの起動とは非同期で行われます。つまり、プロセス内でのメッセージ送信とメッセージ受信は同時には行われません。これは、メソッドの呼び出しと同時に行われるメソッド間のパラメータ渡しとは異なっています。

メソッドと同様に、プロセスも一時的なプロセス固有の変数を持つことができます。

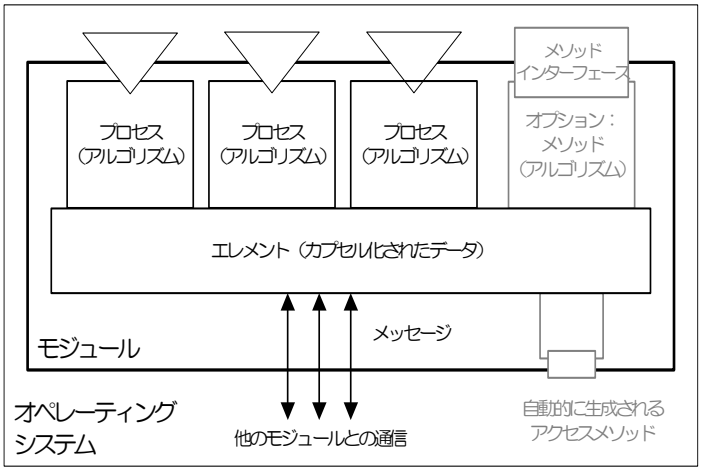


図 2-6　プロセス間通信（グレイの部分はオプション）

2.4　コンポーネントの再利用

コンポーネントを作成する際、以前に作成したクラスやモジュールの中に、再利用できる機能が含まれている場合があります。コンポーネントを再利用すると、コンポーネントは、階層的なツリー構造となります。この構造の葉（ツリーの末端）に相当するのが、他のクラスやモジュールを内包しないクラスまたはモジュールです。

コンポーネントは必ず一方向のツリー構造になっていなければならず、逆方向の依存関係は許されません。なぜなら、コンポーネントを使用する、ということはそのコンポーネントを内包することと等しいので、逆方向の依存関係は解決不可能であるためです。

クラスを別のコンポーネント内で使用する場合、そのクラスはそのコンポーネントの初期化時に自動的にインスタンス化されて初期化されます。

ただし、これには 1 つの例外があります。インポートされたクラスを使用する場合、つまり、そのクラスがどこか他の部分で（たとえばプロジェクト内で直接）インスタンス化されている場合、その使用関係は内包関係ではなく、参照関係となります。したがって、この場合には、逆方向の依存関係を持たせても、前出の「解決不可能な内包関係」は生じません。

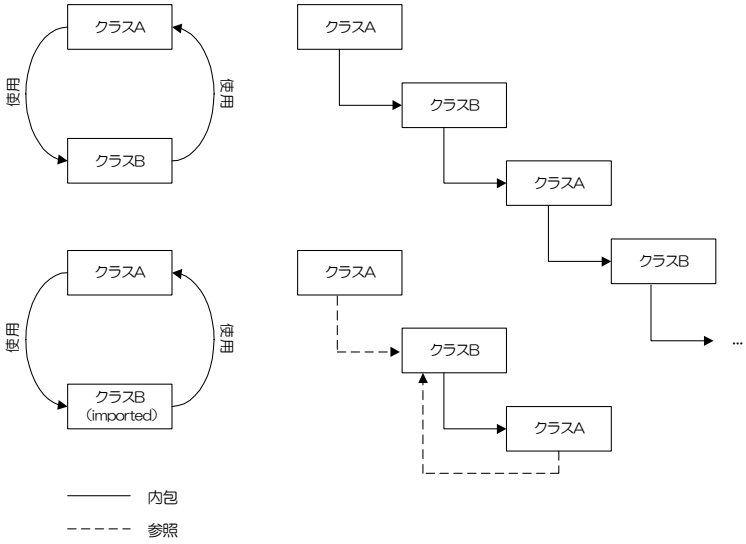


図 2-7　内包関係と参照

モジュールは最上レベルのコンポーネントです。したがって、モジュールをクラスの中に内包させることはできません。しかし、クラスをモジュールや他のクラスの中に内包させることはできます。つまり、以下の関係が成り立ちます。

内包関係	クラスの内包	モジュールの内包
クラス	○	×
モジュール	○	○

表 2-2　各コンポーネントの内包関係

モジュールとクラスとではインターフェースが異なるので、階層モジュール構造と階層クラス構造とでは、やはり意味が異なります。

2.4.1 階層クラス構造

クラスを他のコンポーネントの中で使用する場合、そのクラスのメソッドは、そのコンポーネント内のサブルーチンとして使用することができます。

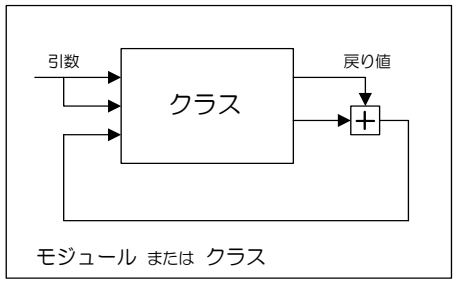


図 2-8 ネストするクラス内でのメソッド呼び出し

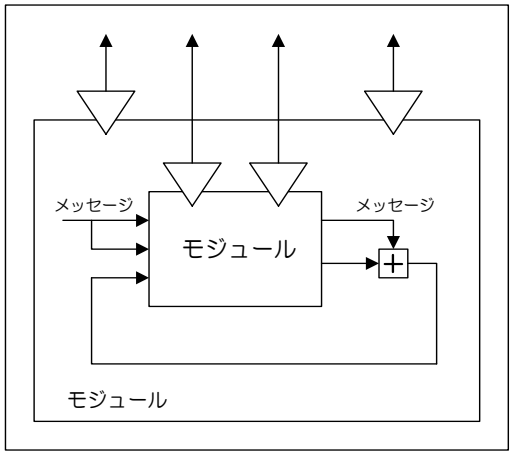
メソッドは、そのコンポーネントのメソッドまたはプロセスの処理の一部として呼び出され、その呼び出しポイントは、そのコンポーネント自体が決定することができます。コンポーネントは、メソッドを呼び出すときに、実際のパラメータを引数としてそのメソッドに供給する必要があります。

2.4.2 階層モジュール構造

前述のように、モジュールは必ずプロジェクト内でインスタンス化されます。つまり、階層モジュール構造において、別のモジュール内で使われているモジュールは、内包する側のモジュール内ではインスタンス化されません。その結果、1 つのプロジェクト内でインスタンス化されるモジュールは、階層構造内での位置とは無関係に、すべて同じレベルとなります。

モジュールを階層構造化することには、主として 2 つの目的があります。階層構造にすると、制御システム全体の機能構成がよくわかります。たとえばエンジン制御の場合、点火制御、噴射制御、ラムダ制御用にそれぞれ独立したモジュールを作成することができます。

しかも、通信構造を階層モードにすると、データフローがブロックダイアグラム上に直接表わされるので、非常にわかりやすくなります。



プロセスは、
オペレーティングシステムにより
起動されます。

図 2-9　階層モジュールの通信構造

また階層モジュール構造には、「保守がしやすい」というもうひとつの利点があります。たとえば、メッセージの名前を変更する場合、通常であればそのメッセージを使用するすべてのモジュールで変更を行わなければなりませんが、階層モジュールを使用している場合には、名前による明示的な結合が行われないため、1つのモジュールを変更するだけで済みます。

2.5　ステートマシン

ステートマシンは特殊なクラスで、演算処理よりも処理フロー制御に重点が置かれる「イベントドリブン」システムです。そのため、ステートマシンの最上位レベルを記述するステートダイアグラムには、「データの流れ」ではなく「状態の遷移」が記述されます。ステートマシンは、有限数の「ステート」（状態）とステート間の「トランジション」（遷移）とで処理フローをモデリングしたものです。ステートマシンを動作させるには、1 つ以上の「トリガ」が必要で、トリガが呼び出されるたびに、ステートマシンの 1 ステップが実行されます。

有限ステートマシンについての詳しい情報は、以下の文献を参考にしてください。

- Harel, David: "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming 8*, 1987, p231-274
- Hatley, Derek J. & Imtiaz A. Pirbhai, *Strategies for Real-Time System Specification*, Dorset House Publishing Co., Inc., NY, 1988.

以下の図は、ステートマシン内の各種コンポーネントを表しています。

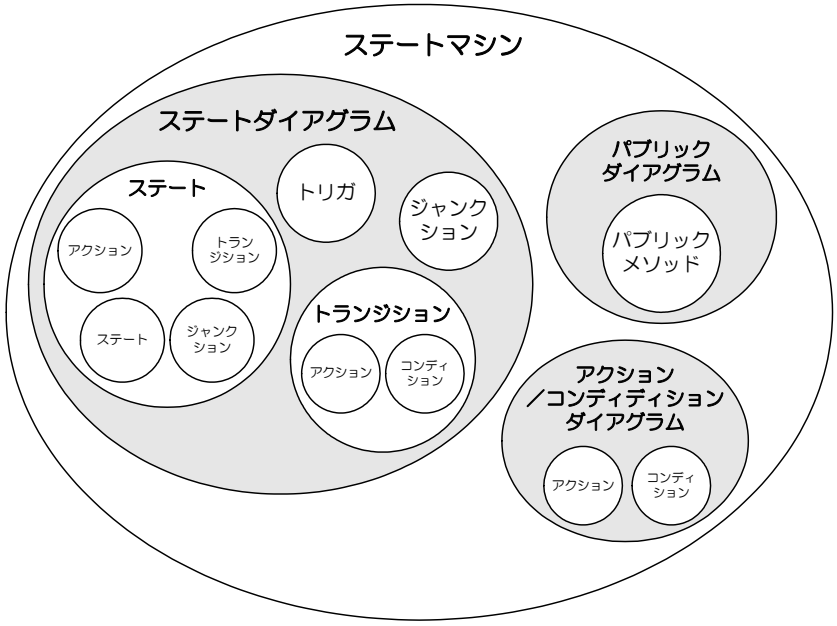


図 2-10 ステートマシンの概念

制御状態の移行は、ステート間のトランジションによりモデリングされます。

ステートマシンは、システムがとりうる状態（「ステート」）と、あるステートから別のステートへの遷移（「トランジション」）が発生するために満たされるべき条件 (「コンディション」)、および遷移が行われる時に実行される処理（「アクション」）により構成されます。

ステートダイアグラムは、ステートマシンを定義するための特別なブロックダイアグラムです。各ステートは、角の丸い四角形で表されます。ステートのうちの 1 つが必ず開始ステートとして定義されていなければならず、ステートマシンはそのステートから開始されます。

トランジションはステート間を結ぶ曲線です。曲線の一方の終端には矢印がついていて、各曲線はその方向へのひとつのトランジションを表します。トランジションの両端は、それぞれひとつづつのステートに接続されています。トランジションの始点側のステートがソースステートで、終点側のステートがデスティネーションステートとなります。双方向トランジションのモデリングには 2 本のトランジションを定義する必要があります。

以下のサンプルダイアグラムには、ステートマシンを構成する各種グラフィックコンポーネントが含まれています。

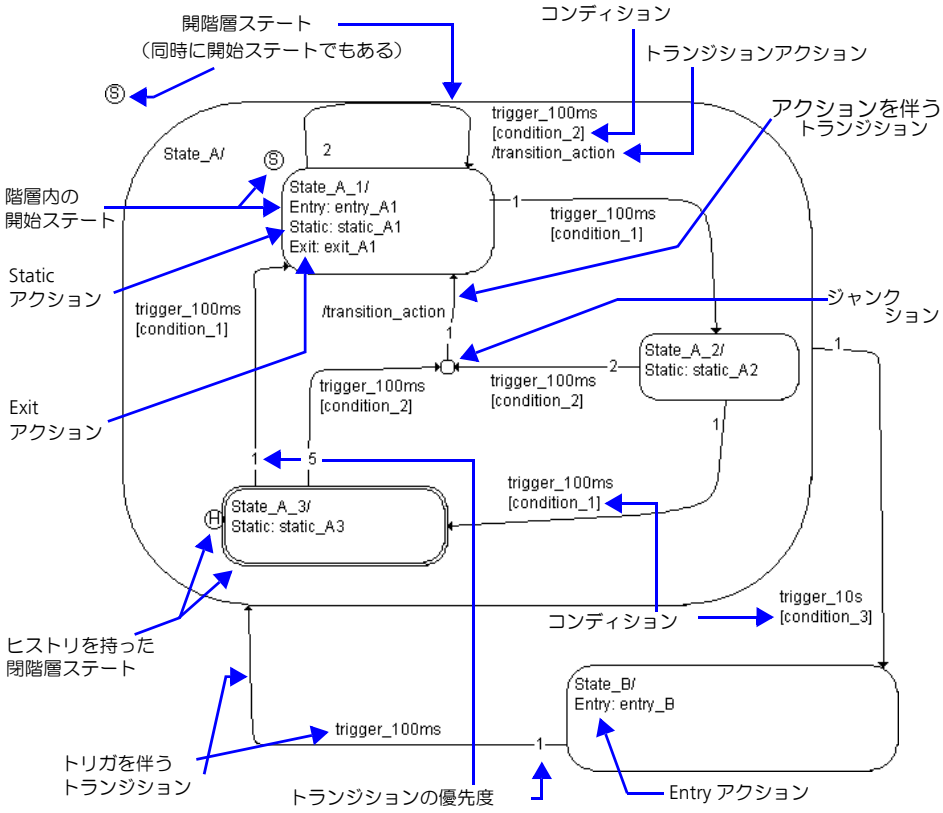


図 2-11 ステートマシンのグラフィックコンポーネント

2.5.1 ステートマシンのコンポーネント

ステート

1 つの「ステート」は、1 つのイベントドリブンシステムを定義するものです。ステートの状態（アクティブ／非アクティブ）は、トリガイベントとコンディションによって、ダイナミックに変化します。

各ステートにはペアレントステートがあります（41 ページの「階層ステート」を参照してください）。最上位のステート（図 2-11 の `State_A` と `State_B`）のペアレントステートは、ステートダイアグラム自体がペアレントとなります。他の上位レベルステート（`State_A` のサブステートである `State_A_1`、`State_A_2`、および `State_A_3`）内にさらにステートを配置することも可

能です。内部にステートを含んでいないステート（図 2-11 の State_A_1、State_A_2、および State_B）を、ベースステートと呼びます。階層ステートには「ヒストリ」（43 ページ参照）を持たせることができます。ヒストリは、過去の動作からの継続した動作を行うために使用されます。

ステートはそれぞれ排他的な関係にあり、常にいずれか 1 つのベースステートのみがアクティブになります。現在アクティブなベースステートが、ある階層ステート内のサブステートであった場合、そのアクティブステートを含む階層ステート全体がアクティブとなります。たとえば図 2-11 のステート State_A_2 がアクティブである場合、階層ステートの State_A もアクティブになり、また State_A_3 内のサブステート（図には示されていません）のうちのいずれかアクティブになれば、State_A_3 および State_A もアクティブとなります。

各ステートにはユニークな名前を付ける必要があり、異なる階層間でも同じ名前のステートを持つことはできません。もしすでに存在する名前を割り当てた場合は、その名前の最後尾に _n というサフィックスが自動的に付加されます。ここでの n は、その名前にまだ割り当てられていない数字のうちの最小の数字で、図 2-11 のステート State_A_1 から State_A_3 のようになります。ステート名には以下の名前は使用できません。

- プロジェクト内のメソッド、プロセス、エレメントなどの名前
- C 言語の予約語（static、define など）

  このようなステート名を使用した場合、エラーメッセージが出力されない場合もありますが、生成されたコードは必ず不正なものとなります。

ステートラベルには、名前以外にもさまざまなアクション（44 ページ参照）が含まれています。これらのアクションは、そのタイプによって適宜に実行されます。タイプには Entry アクション、Static アクション、および Exit アクションがあり、いずれのものも、必要に応じて任意に定義されます。

トランジション

「トランジション」は、2 つのステートを接続するグラフィックオブジェクトです。トランジションの一方の端はソースステートに接合していて、そこからトランジション（遷移）が開始します。もう一端はデスティネーションステートに接合し、そこでトランジションが終了します。このトランジションが「アクティブ」になることにより、その遷移が実行されます。トランジションはジャンクション（38 ページ参照）によって複数のセグメントに分割することが可能です。

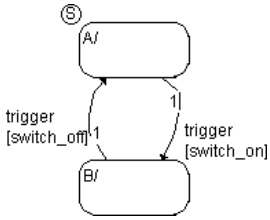
各トランジションには優先度が割り当てられていて、この番号が大きくなるにつれて優先度が上がります。ある 1 つのステートまたはジャンクションを始点とするトランジションが 1 つ以上ある場合、それらのトランジションは優先度が高い順に評価されます。そのため 1 つのステート／ジャンクションを始点とする複数のトランジションは同じ優先度を持つことはでいません。

トランジションのラベルは、あるステートから別のステートに移るための条件を表します。トランジションが評価されるためには、トリガイベントが必要です。トランジションラベルの最初の部分が、トリガの名前です。図 2-11 では、State_A_1 から State_A_2 へのトランジションは trigger_100ms というトリガによって評価が行われます。トランジションには、必要に応じて「コンディション」（44 ページ参照）と「トランジションアクション」（44 ページ参照）

を含めることができます。これらの内容は、ラベルの 2 番目と 3 番目の部分に表示されています。コンディションは [ ] で囲まれて表示され、トランジションアクションは先頭に / が付加されています。ジャンクションを含むトランジションの各セグメントへのトリガ、コンディション、およびアクションの割り当てについては、38 ページの「ジャンクション」を参照してください。

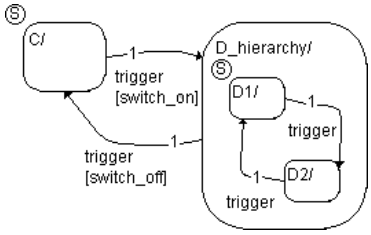
トランジションは、そのソーステートがアクティブであり、かつそのトランジションにコンディションが定義されている場合はその評価結果が true になった時に、アクティブとなります。トランジションには、以下のようなさまざまなケースがあります。

1. ベーステート間のトランジション



ステート A がアクティブである時にトリガイベント trigger が発生し、かつコンディション [switch_on] が true であった場合、ステート A から B へのトランジションがアクティブになります。

2. 階層ステートへのトランジション

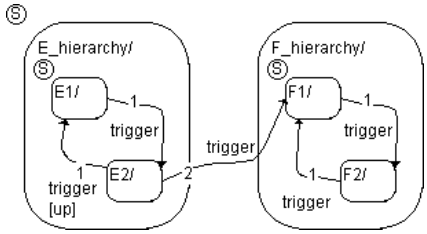


ステート C がアクティブである時にトリガイベント trigger が発生し、かつコンディション [switch_on] が true であった場合、ステート C から階層ステート（41 ページ参照）D_hierarchy へのトランジションがアクティブになります。

階層ステートへの正確な遷移が行われるようにするためには、デスティネーションとなるサブステートが暗黙的に定義されている必要があります。上の例ではサブステート D1 が開始ステートとしてマークされているため、実際のトランジションは C から D1 へと行われます。

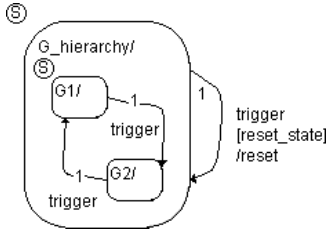
D_hierarchy から C へのトランジションは、いずれのサブステートがアクティブであるかに関わらず、D_hierarchy がアクティブである時にトリガイベント trigger が発生し、かつコンディション [switch_off] が true であった場合に、アクティブになります。

3. 異なる階層内のサブステート間のトランジション



ステート `E2` がアクティブである時にトリガイベント `trigger` が発生すると、階層ステート `E_hierarhcy` 内のサブステート `E2` から階層ステート `F_hierarcy` 内のサブステート `F1` へのトランジションがアクティブになります。トランジションによって、サブステート `E2` からの明示的な離脱と、階層ステート `E_hierarhcy` からの暗黙的な離脱が行われます。また `F_hierarcy` への暗黙的なエントリと `F1` への明示的なエントリが行われます。

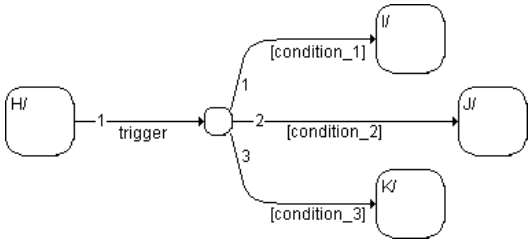
4. ループ



ループは、あるステートからそれ自身へのトランジションで表されます。上の図の場合、`G_hierarchy` のいずれかのサブステートがアクティブである時にトリガイベント `trigger` が発生し、かつコンディション `[reset_state]` が `true` であった場合にアクティブとなります。この時、システムは現在のサブステートのアクティブ状態を保ったまま `G_hierarchy` から抜け出してトランジションアクションを実行し、再び `G_hierarchy` に戻って最終的にサブステート `G1` がアクティブになります。

5. ジャンクションを含むトランジション

   トランジションには複数のジャンクション（次の項を参照してください）を含めることができます。以下に一例を挙げて説明します。



   ステート `H` がアクティブである時にトリガイベント `trigger` が発生すると、システムはステート `H` から離脱し、ジャンクションにおいて、その先のセグメント（`[condition_1]`、`[condition_2]`、`[condition_3]`）に分岐するための各コンディションが優先度の順に従って評価されます。たとえばここでコンディション `[condition_2]` の条件が満たされていれば、ステート `J` への遷移が発生します。いずれのコンディションの条件も満たされていない場合は遷移は発生せず、ステート `H` がアクティブのままとなります。

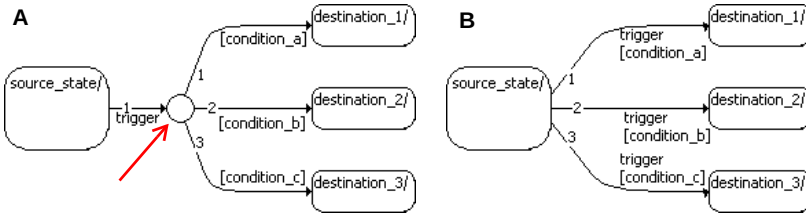
ジャンクション

「ジャンクション」は、ステートダイアグラムをよりわかりやすく、さらにはコード生成をより効率的にするためのグラフィカルオブジェクトです。またこれによってシステムの挙動がより正確なものとなります。

ジャンクションはステートとは異なり、ステートダイアグラム内の分岐点を示すものです。ジャンクションによってトランジション（35 ページ参照）が複数のセグメントに分割されます。つまり、1 つのセグメントでソースとジャンクションを接続し、そのジャンクションから複数のセグメントをそれぞれ異なるターゲットステートに接続します。必要に応じて 1 つのトランジション内に複数のジャンクションを含め、分岐をネストさせることもできます。このようにして、何通りものトランジションを表現することが可能となります。またトランジションセグメントの再利用も可能です。

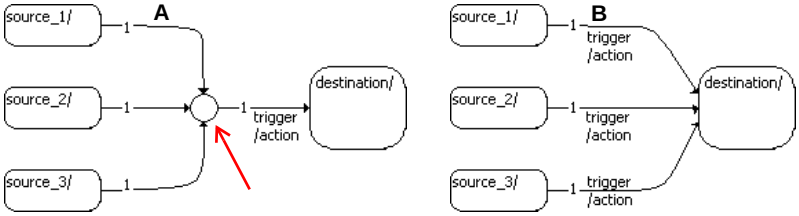
ジャンクションの使用例を以下に示します。

- 1 つのソースステートからいくつかの異なるステートへのトランジションが、明確に表現されます。



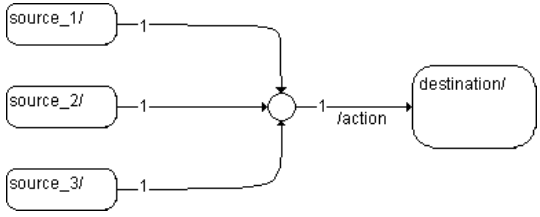
上図の A のようなジャンクションを使用してモデリングされる動作は、B に示されるように、ソースステート `Source_state` から各ターゲットステートへの直接的なトランジションによっても表現することができます。しかし A のようにジャンクションを使用すれば、ソースステートとジャンクション間のセグメントが最初に評価され、それが無効であれば、ジャンクション以降のセグメントの評価は一切行われずにトランジションは無効となるため、実行時間の短縮に役立ちます。

- 複数のソースステートから 1 つのターゲットステートへのトランジションも、ジャンクションによって明確に表現できます。



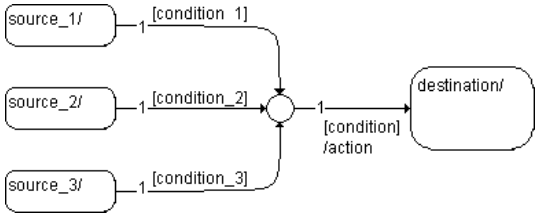
この場合も 2 通りの表現が可能ですが、ジャンクションを使用して、ジャンクションの先のセグメントに 3 つのトランジションで共有されるアクションを接続できます。

- ジャンクションから先のすべてのトランジションセグメントが無効な場合はトランジションは実行されず、システムは現在のステートに留まります。
- ジャンクションからターゲットステートへのトランジションセグメントには、アクションを割り当てることができます。



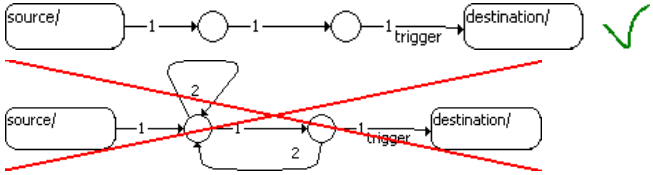
一方、ジャンクションの手前のトランジションセグメントにはアクションを割り当てることはできません。これは、トランジションアクションが複雑になり、それによって非効率的なコードが生成されてしまうのを避けるためです。

- コンディションは、トランジション内の各セグメントに割り当てることができます。

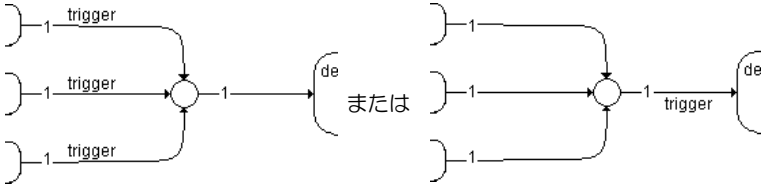


一方、ジャンクションの手前のトランジションセグメントにはアクションを割り当てることはできません。これは、トランジションアクションが複雑になり、それによって非効率的なコードが生成されてしまうのを避けるためです。

- ジャンクションからジャンクションへの遷移（「カスケードジャンクション」）は可能ですが、ここではどのようなループも使用できません。



- トリガは、トランジション内のいずれか 1 つのセグメントにのみ割り当てることができます。通常、トリガは、最初のジャンクションに向かうセグメント、または最後のジャンクションからターゲットステートに向かうセグメントのいずれかに割り当てます。



注記

1 つのトランジション内の複数のセグメントにトリガを割り当てる操作は可能ですが、そのような操作を行うとエラーメッセージが出力されます。トリガの割り当てには充分な注意が必要です。

- 複数のターゲットステートに接続される各セグメントが 1 つも有効でない場合、遷移は発生しません。この場合、ステートマシンはソースステートに留まります。

トリガ

「トリガ」によって、ステートマシンが起動されます。つまり、トリガ呼び出しが行われるたびにステートマシンの 1 ステップが実行されます。トリガはステートマシンのパブリックメソッドで、ステートダイアグラムごとにトリガを定義する必要があります。トリガには、他の ASCET コンポーネントとの通信を行うための引数を持たせることができます（81 ページの「ステートマシンクラス」および『ASCET ユーザーズガイド』の「ステートマシンエディタ」の項を参照してください）。

1 つのステートマシンには、1 つまたは複数のトリガを持たせることができます。各トランジションをいずれかのトリガに割り当てることにより、1 つのステート内に複数のサブステートマシンを定義することが可能となります。各トリガはそれぞれ異なるタイミングで発生させることができます。

ステートマシンは、所定のトリガが発生するたびに起動され、このとき、現在のステートを始点とするすべてのトランジションが優先度に従って評価され、有効なトランジションがあれば、それがアクティブになり、状態遷移が行われます。

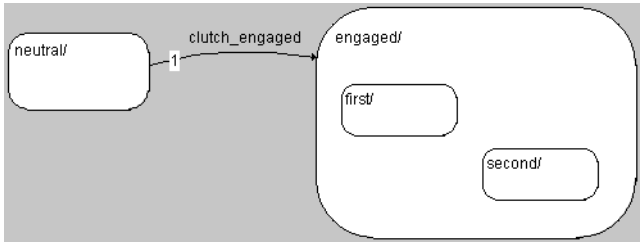
階層ステート

ステートマシンには非常に多くのステートが含まれる場合がありますが、「階層ステート」を利用すれば、複雑なシステムを親子関係のオブジェクト構造によってわかりやすく整理することが可能です。階層構造にすることにより、トランジションの数が削減され、構造的で理解しやすいダイアグラムとなります（『ASCET ユーザーズガイド』の「階層ステート」の項を参照してください）。

ASCET では、開いた形の階層ステートと閉じた形の階層ステート（図 2-11 の `State_A` と `State_A_3`）の両方がサポートされています。両者の違いは、グラフィック表記上のものだけです。

内部に他のステートを含んでいるステートを階層ステートと呼び、他のステートを含まないステートをベースステートと呼びます。階層ステートに含まれるステートを、その階層ステートのサブステートと呼びます。システムの状態は常にいずれか 1 つのベースステート内にあり、同時に、そのベースステートが含まれている階層ステート内にあることにもなります。

以下のステートダイアグラムには 2 つのサブステートが含まれています。（単純化のため、いくつかのトランジションは省略されています。）



階層ステート engaged には 2 つのサブステート、first と second が含まれています。ここでは engaged が first と second のペアレントステートとなっています。トリガイベントである clutch_engaged が発生すると、システムは、neutral ステートから階層ステート engaged に遷移します。

さらに複雑な階層構造も可能です。以下の例（図 2-12）のステートマシンには 2 つの階層ステートがあり、そのうちの 1 つにはさらにもう 1 つの階層ステートが含まれています。なおステート間の線は内包関係を示すもので、トランジションではありませんので、注意してください。

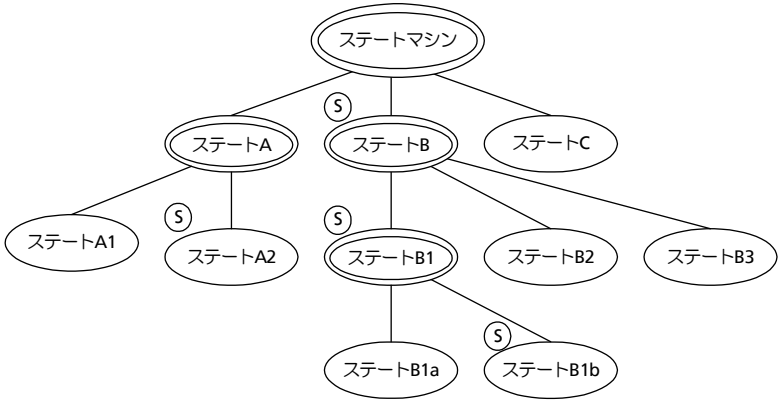


図 2-12 複雑なステートマシン内の関係

階層ステート内では、いくつかのサブステートが独自のステートマシンを形成しています。たとえば、ステート A1 とステート A2 は独自のステートマシンを形成しています。階層ステート内のステートは、同じ階層ステート内に存在しない他のステートへのトランジションを持つことができます。ステート同士はトランジションで接続され、各階層ステート内ではいずれか 1 つのステートがその階層ステート内の開始ステートとして定義されます。初期状態では、階層ステートマシンは開始ステートの状態となり、この開始ステートが階層ステートの場合には、その階層ステート内の開始ステートが実際の開始ステートになります（以下同様）。

この例では、ステートマシンの開始ステートはステート B1b です。その理由は、ステート B1b はステート B1 の開始ステートであり、ステート B1 はステート B の開始ステートであり、さらにステート B は最上レベルの開始ステートであるためです。

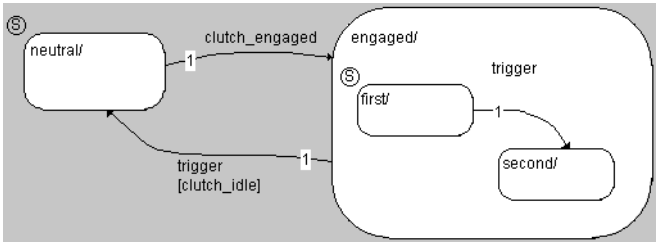
階層ステートからの遷移が行われる時には、アクティブなサブステートからの離脱も自動的に行われます。またサブステートからのトランジションは、階層ステートの枠を超えて、別のサブステートに向けて行うことができます。あるサブステートがアクティブになると、その上位の階層ステートもアクティブになります。

開始ステート

「開始ステート」は、1 つの階層レベル内に遷移可能なステートがいくつかある場合に、実際にどのステートをアクティブにするかを決定するためのものです。これによって、ステートマシン全体、あるいは 1 つの階層レベルの開始ステートが決まります。

ステートマシンを作成する際に間違いやすい点として、開始ステートの定義を忘れてしまうことがあります。この場合、デフォルト状態においてどのステートがアクティブになるか、という定義が存在しないことになるため、ASCET はコード生成時にエラーメッセージを出力します。

以下のステートダイアグラム全体の開始ステートは、`neutral` ステートです。

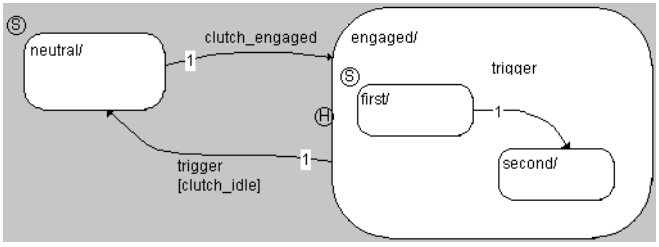


ここでは、ステートマシンが最初に起動された時、`neutral` ステートがアクティブとなります。もし開始ステートが定義されていないと、`neutral` と `engaged` のどちらがアクティブになるかが未定義となってしまいます。`neutral` から `engaged` への遷移が発生すると、階層ステート内ではサブステート `first` がアクティブになります。

ヒストリ

「ヒストリ」オプションは、階層ステートへのトランジションにおいて、過去の動作に基づいて遷移先のサブステートを決定するために使用されます。階層ステートがヒストリを持っている場合、トランジションのデスティネーションは、最後にアクティブであったサブステートとなります。

ヒストリは、そのオプションが設定されている階層に属します。ヒストリの内容は、階層内の開始ステートよりも優先されます。

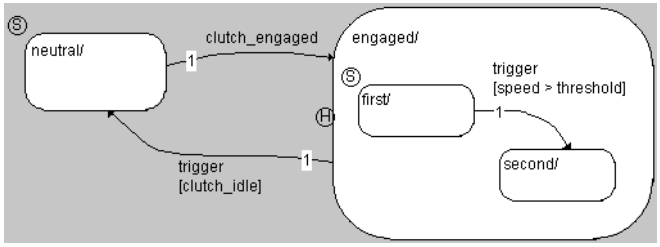


上の図に表示されている H のマークは、階層ステート engaged がヒストリを持っていることを表しています。neutral から engaged へのトランジションにおいてサブステートの first と second のいずれがアクティブになるかは、過去にどちらが最後にアクティブになっていたかにより決定されます。

コード生成時には、ヒストリ用の「ヒストリ変数」が生成されます。

コンディション

「コンディション」は論理式です。その式の結果が true になるとトランジションがアクティブになり、実行されます。図 2-11 では、[Condition_3] というコンディションに定義された論理式が true になると、State_A から State_B への遷移が実行されます。



このシステムでは、first から second へのトランジションは、論理式 [speed > threshold] が true になった時に行われます。

コンディションは、ブロックダイアグラムまたは ESDL で記述します。ブロックダイアグラムの場合は独立したダイアグラムとして記述し、ESDL の場合は、独立したダイアグラムとして記述するか、またはトランジション内に直接記述します。詳しくは『ASCET ユーザーズガイド』の「コンディションとアクションの定義」の項を参照してください。

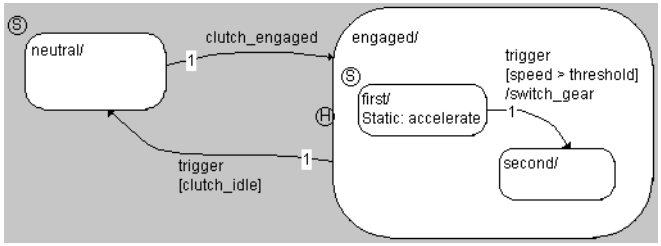
コンディションには、他の ASCET コンポーネントとの通信を行うための引数を持たせることができます（81 ページの「ステートマシンクラス」および『ASCET ユーザーズガイド』の「他のコンポーネントとの通信」の項を参照してください）。

アクション

「アクション」は、ステートマシンの処理の一部として実行されます。アクションは、ステート間のトランジションの一部として（例、図 2-11 の /transition_action）、あるいはステートの状態に応じて（例、図 2-11 の static_A2 あるいは exit_A1）実行されます。

トランジション、およびジャンクションから先のトランジションセグメントは「トランジションアクション」を持つことができ、またステートは、「Entty」、「Static」、「Exit」アクション を持つことができます。これらのアクションは、必要に応じて任意に定義します。34 ページの図 2-11 では、ステート State_A_1 は 3 つのすべてタイプのアクションを持っていますが、State_A_2 は Entry

および Exit アクションは持っておらず、Static アクションのみが割り当てられています。また `State_A_1` から `State_A_2` へのトランジションはアクションを持っていません。



上の図では、`first` ステートがアクティブであって、かつ状態遷移が発生しない時は、`accelerate` という Static アクションが実行されます。そして `first` から `second` へのトランジションにおいて、`switch_gear` というトランジションアクションが実行されます。

どのタイミングでどのアクションが実行されるかは、「基本的なステートマシン」、「ジャンクションを使用したステートマシン」、および「階層ステートマシン」の項に詳しく説明されています。アクションは、ブロックダイアグラムまたは ESDL で記述します。ブロックダイアグラムの場合は独立したダイアグラムとして記述し、ESDL の場合は、独立したダイアグラムとして記述するか、またはトランジション内に直接記述します。詳しくは、『ASCET ユーザーズガイド』の「コンディションとアクションの定義」を参照してください。

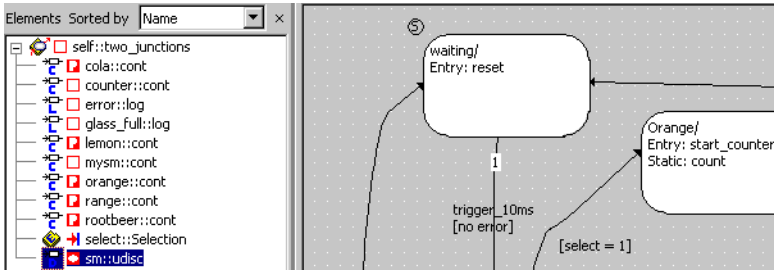
アクションには、他の ASCET コンポーネントとの通信を行うための引数を持たせることができます（81 ページの「ステートマシンクラス」および『ASCET ユーザーズガイド』の「他のコンポーネントとの通信」の項を参照してください）。

データ

「データオブジェクト」は、ステートダイアグラム内の数値を保存し処理するために使用されるもので、以下のタイプがあります。

- 変数、パラメータ、定数（92 ページ参照）
- 列挙型データ（91 ページ参照）
- 配列、マトリックス（87 ページ参照）
- リテラル（92 ページ参照）
- テンポラリ変数（93 ページ参照）
- 特性カーブ／マップ（88 ページ参照）
- 他の ASCET コンポーネントからの入力
- 他の ASCET コンポーネントへの出力
- 他のクラス（タイマ、カウンタ、コンパレータ等）

以下の図に示された `unsigned descrete` 型のステート変数 `sm` は、この「データ」のひとつです。この変数は、すべてのステートマシンにおいて自動的に作成されます。



この変数には現在アクティブなステートの番号が格納されます。これをステートマシンエディタで編集することはできませんが、実験において測定することが可能です。ステートマシンを含むプロジェクト用の ASAM-MCD-2MC ファイルが作成されると、`sm` パラメータもそのファイルに格納されます。

2.5.2 ステートマシンの動作

ひとつのステートマシンは、有限数のステートから成り立っています。各ステートは、システムがとり得るさまざまな状態を表します。たとえば、ドアがロックされているか、開いているか、あるいは閉じているか、といった状態です。これらのステートの遷移は、ステート間のトランジションによってモデリングされます。この遷移が実行されるためには、トランジションに定義された条件が満たされている必要があります。

ステートマシンは、外部イベント（「トリガイベント」）によって起動されます。トリガは、ステートマシンのパブリックメソッドです。ひとつのステートマシンは、常にそのステートマシン内のいずれかのステートになっています。始めは、開始ステートという特別のステートになります。トリガイベントが発生すると、システムはそれに応じたアクション（シグナルの生成、変数の更新、他のステートへの遷移等）を実行します。

ステートの Entry アクションは、そのステートへの遷移が発生した時に行われます。実際にはそのステートがアクティブになった後に、Entry アクションが開始されます。

注記

ステートマシンが始めて呼び出される時は、開始ステートの Entry アクションは実行されません。

ステートの Static アクションは、ステートがアクティブで、かつ発生したトリガイベントによってそのステートからのトランジションがアクティブにならなかった時に実行されます。同じ階層ステート内でサブステート間の遷移が発生した場

合は、階層ステート自体からのトランジションは発生しないため、ソースステートからの離脱後、トランジションアクションが実行される前にその階層ステートの Static アクションが実行されます。

ステートの Exit アクションは、そのステートからの遷移が発生した時に実行されます。この Exit アクションが完了した後、ステートは非アクティブ状態となります。

トランジションのトランジションアクション は、ソースステートからの離脱後、デスティネーションステートがアクティブになる前に実行されます。

以下に続く項には、ステートダイアグラムが実際にどのように実行され、どのような順番で各アクションが実行されるかについて詳しく記述されています。適切なステートマシンを作成して効率のよいコードが生成されるようにするためには、ステートダイアグラムの意味を正確に理解することが必要です。誤った記述を行うと、シミュレーションの挙動が不正確なものになってしまいます。

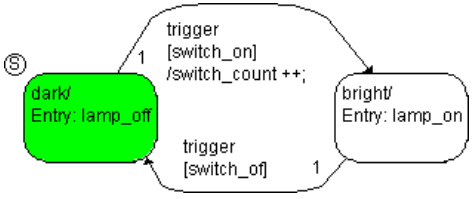
説明には、以下の点についてのルールが含まれます。

- 各ステート内での処理
- 有効な（つまり、条件の満たされた）トランジションの検索
- トランジションに定義された状態遷移の実行

いくつかの例をあげて、さまざまなステートマシンの動作について詳しく説明します。

2.5.3 基本的なステートマシン

例 1: 2 つのステート間のトランジション



これは、ライトスイッチの簡単なステートマシンモデルです。最初はランプは OFF で、`dark` ステートがアクティブとなります。トリガイベント `trigger` が発生するとステートマシンの処理が開始されます。ライトスイッチが押されたために `switch_on` というコンディションが true になり、その後は以下のような順で処理が行われます。

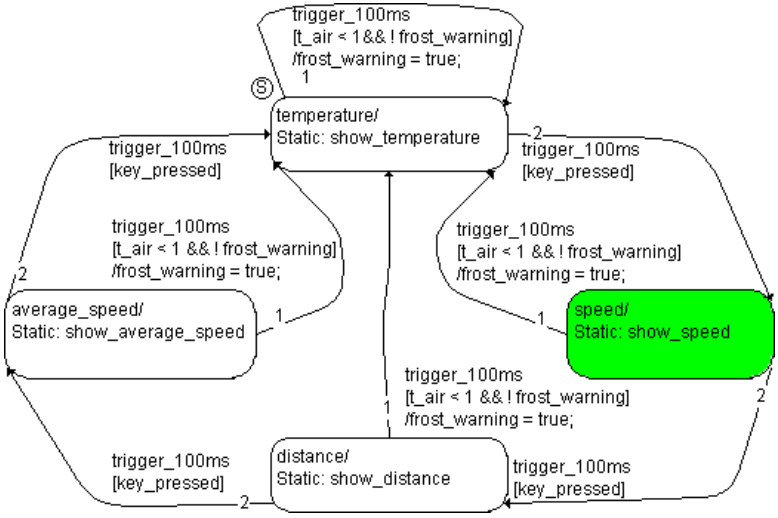
1. システムは、有効なトランジションがあるかどうかをチェックします。
2. `dark` ステートがアクティブなので、`dark` から `bright` へのトランジションのみが評価されます。`[switch_on]` という条件が満たされているので、トランジションは有効となります。
3. `dark` ステートには Exit アクションがないので、Exit アクションなしにステートは非アクティブ状態となります。

4. トランジションアクションが実行され、カウンタ `switch_count` に 1 が加算されます。
5. `bright` ステートがアクティブになります。
6. `lamp_on` という Entry アクションが実行されます。完了すると、ランプが ON になります。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します。

ステートは、1 つあるいは複数の他のステートへのトランジションを持っていますが、ステートマシンの動作を明確にするためには、各トランジションに優先度を割り当てる必要があります。この優先度によって、トランジションの条件を評価する順番が決定されます。コンディションの評価結果が true になると、そのトランジションによる遷移が実行され、それよりも低い優先度を持つトランジションのコンディションは評価されません。true となるコンディションがひとつもない場合は、ステートの状態は変化せず、Static アクションが実行されます。

例 2: 1 つのステートからいずれか 1 つの遷移が発生する場合

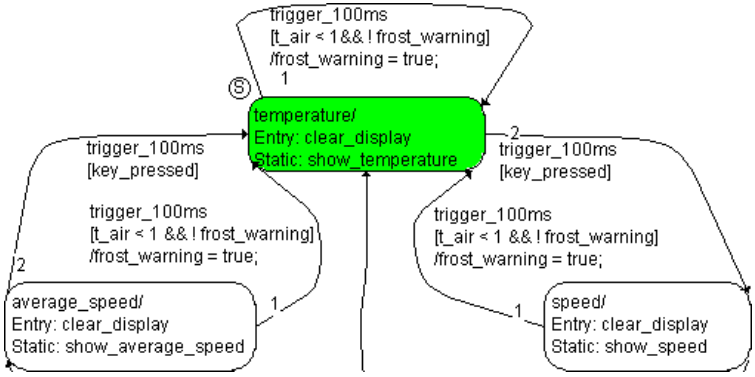


これは、ディスプレイパネルのステートマシンモデルです。要求に応じて、外気温度、車速、平均車速、および走行距離を表示します。また、表示内容を切り替えるためのキースイッチもあります。外気温 1 ℃を下回ると、温度表示への切替えが発生し、フロストワーニング（氷結に関する警告メッセージ）を表示します。

ステートマシンの現在のステートは `speed` です。外気温が 1.5 ℃から 0.5 ℃に低下し、トリガイベント `trigger_100ms` が発生します。表示切替えスイッチは押されていません。ここでは以下のような順で処理が行われます。

1. システムは、speed からの有効なトランジションがあるかどうかをチェックします。
2. speed から distance へのトランジションは最上位の優先度を持っているので、最初に評価されます。しかし [key_pressed] という条件は満たされていないため、トランジションは無効です。
3. speed から temperature へのトランジションには [t_air < 1 && !frost_warning] というコンディションがあります。今までは温度はスレッシュホールドの 1 ℃よりも高く、ワーニングは必要ありませんでしたが、現在の温度は 0.5 ℃に低下しています。そのためコンディション内の両方の条件が true となり、トランジションが有効となります。
4. speed ステートには Exit アクションがないので、Exit アクションなしにステートは非アクティブ状態となります。
5. /frost_warning = true というトランジションアクションが実行され、ワーニングメッセージが表示されます。
6. temperature ステートがアクティブになります。
7. このステートには Entry アクションがないので、トリガイベントによって開始されたステートマシンの処理はここで終了します。

例 3: ループ



これは例 2 のステートマシン内のステートに clear_display という Entry アクションを加えたものです。ステートマシンの現在のステートは、temperature です。トリガイベント trigger_100ms が発生し、このとき表示切替えスイッチは押されていません。ここでは以下のような処理が行われます。

1. システムは、temperature からの有効なトランジションがあるかどうかをチェックします。
2. temperature から speed へのトランジションは最上位の優先度を持っていますが、条件が満たされていないため、トランジションは無効です。
3. temperature から temperature 自身へのトランジションには [t_air < 1 && !frost_warning] というコンディションがあります。現在この条件が満たされているので、トランジションは有効となります。

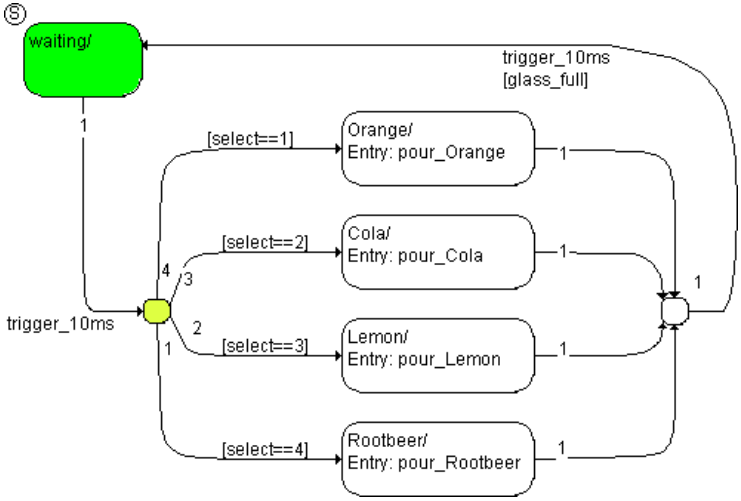
4. `temperature` ステートには Exit アクションがありません。ステートはアクションなしに非アクティブ状態となります。
5. `/frost_warning = true` というトランジションアクションが実行され、ワーニングメッセージが表示されます。
6. `temperature` ステートがアクティブになります。
7. `temperature` ステートの Entry アクション `clear_display` が実行され、完了します。

   トリガイベント `trigger_100ms` によって開始されたステートマシンの処理は、ここで終了します

2.5.4 ジャンクションを使用したステートマシン

ジャンクション（38 ページ参照）は、ステートダイアグラムをより分かりやすくするためのものです。以下のすべての例は、ジャンクションを使用せずに、それぞれのステート間を直接結ぶトランジションで記述することもできます。

例 4: If ... Then .. Else の構築



このステートマシンは、4 種類の飲み物を提供する簡単なドリンクマシンをモデリングしたものです。ステートマシンの現在のステートは、`waiting` です。コーラを飲みたい人がいて、トリガイベント `trigger_10ms` が発生しました。選択ボタンによって `select` は 2 に設定されているため、以下のような処理が行われます。

1. システムは、`waiting` からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。

   `waiting` からダイアグラム左側のジャンクションまでのトランジションセグメントが有効です。

2. ジャンクションから先のトランジションセグメントが優先度の順に評価されます。つまり orange ステートへのセグメントが最初に評価されます。

   [select==1] という条件は満たされていないので、このセグメントは無効です。

3. 次にジャンクションから Cola ステートへのセグメントが評価されます。

   [select==2] という条件が満たされているので、waiting から Cola へのトランジションの全区間が有効となります。

4. ここで初めてトランジションで定義された状態遷移が実行されます。waiting ステートには Exit アクションがないため、アクションなしに非アクティブ状態となります。

5. Cola ステートがアクティブになります。

6. pour_Cola という Entry アクションが実行され、完了します。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します

例 5: 状態遷移が発生しない場合

例 4 のステートマシンを使って説明します。ステートマシンの現在のステートは waiting です。トリガイベント trigger_10ms が発生しました。操作ミスのために select は 5 に設定されています。ここでは以下のような処理が行われます。

1. システムは、waiting からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。

   waiting からダイアグラム左側のジャンクションまでのトランジションセグメントが有効です。

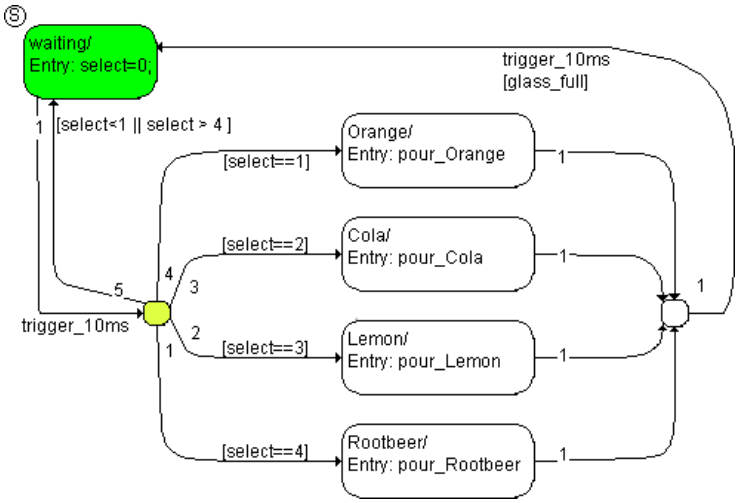
2. ジャンクションから先のトランジションセグメントが優先度の順に評価されます。

   select は 5 に設定されているためにどの条件も満たされず、いずれのセグメントも無効です。

3. waiting からの有功なトランジションは存在しないため、waiting ステートに留まります。このステートには Static アクションがないため、なにもアクションは実行されません。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します

例 6: ループの構築



例 4 のステートマシンに、ジャンクションから `waiting` に戻るトランジションセグメントとを加え、さらに `waiting` に Entry アクションを加えたものです。

ステートマシンの現在のステートは、`waiting` で、トリガイベント `trigger_10ms` が発生しました。`select` は操作ミスのために 5 に設定されています。ここでは以下のような処理が行われます。

1. システムは、`waiting` からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。

   `waiting` からダイアグラム左側のジャンクションまでのトランジションセグメントが有効です。

2. ジャンクションから先のトランジションセグメントが優先度の順に評価されます。この例の場合では、ジャンクションから `waiting` ステートに戻るセグメントが最初に評価されます。

   `[select < 1 || select > 4]` という条件が満たされているので、このセグメントは有効です。このため、`waiting` から `waiting` へのトランジションの全区間が有効となります。

3. `waiting` ステートには Exit アクションがないため、アクションなしに非アクティブ状態となります。

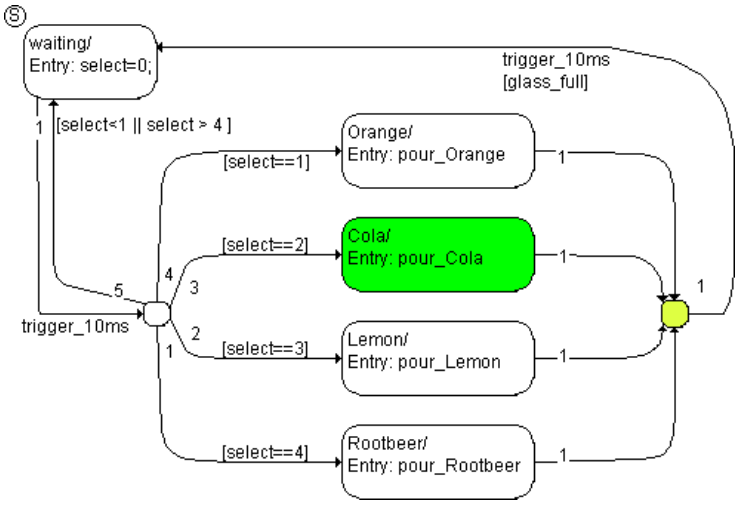
4. `waiting` から `waiting` へのトランジションにはトランジションアクションがないため、`waiting` ステートはすぐに再びアクティブとなります。

5. `waiting` の Entry アクションである `Select=0;` が実行され、完了します。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します。

ループの構築は、例 3 のように、あるステートからそのステート自身への直接のトランジションを使用します。

例 7: 複数のソースステートから 1 つのターゲットステートへの、1 つのトリガによるトランジション



例 6 と同じステートマシンです。現在 `Cola` ステートがアクティブで、グラスにコーラが満たされ、論理変数 `glass_full` が `true` に設定されました。トリガイベント `trigger_10ms` が発生すると、以下のような処理が行われます。

1. システムは、`Cola` からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。

   `Cola` からダイアグラム右側のジャンクションまでのトランジションセグメントが有効です。

2. ジャンクションから `waiting` ステートへのトランジションセグメントには

   `[glass_full]` というコンディションがあります。`glass_full` は true に設定されているため、このセグメントも有効で、トランジションに定義された状態遷移が実行されます。

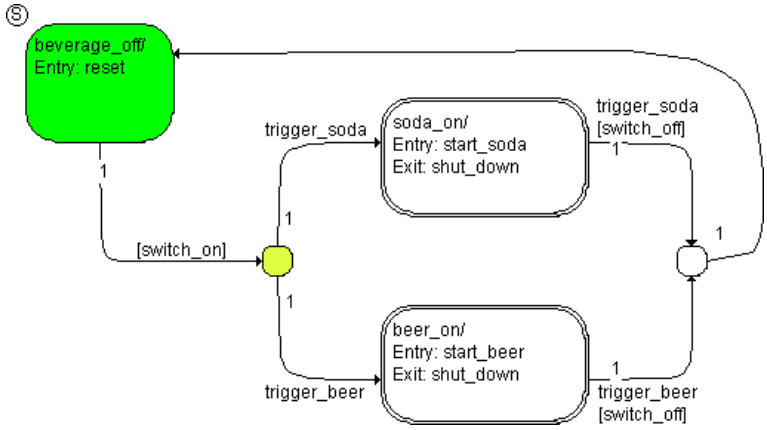
3. `Cola` ステートには Exit アクションがないため、アクションなしに非アクティブ状態となります。

4. トランジションにはトランジションアクションがないため、`waiting` ステートはすぐにアクティブとなります。

5. `waiting` の Entry アクションである `Select=0;` が実行され、完了します。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します

例 8: 1 つのソースステートから複数のターゲットステートへの、複数のトリガによるトランジション



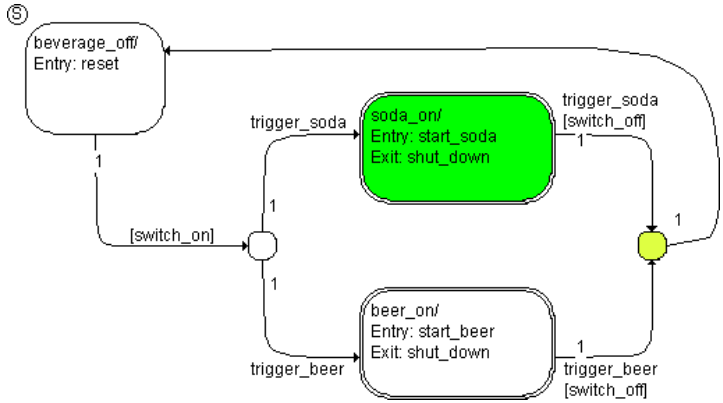
このステートマシンは、それぞれ何種類かのソーダとビールを提供するドリンクマシンを表しています。実際の種類の判定は、階層ステート `soda_on` と `beer_on` で行われます（階層ステートについては「階層ステート」の項で説明します）。

ステートマシンの現在のステートは `beverage_off` です。トリガイベント `trigger_soda` が発生し、この時ドリンクマシンのスイッチは ON になっているため `switch_on` は `true` に設定されています。ここでは以下のような処理が行われます。

1. システムは、`beverage_off` からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。
2. コンディション `[switch_on]` の条件が満たされているため、`beverage_off` からジャンクションまでのトランジションセグメントが有効です。さらにトリガイベント `trigger_soda` が発生しているためにジャンクションから `soda_on` ステートへのセグメントも有効であるため、トランジションに定義された遷移が実行されます。
3. `beverage_off` ステートには Exit アクションがないため、アクションなしに非アクティブ状態となります。
4. `beverage_off` から `soda_on` へのトランジションにはトランジションアクションがないため、`soda_on` ステートはすぐにアクティブとなります。
5. `soda_on` の Entry アクションである `start_soda` が実行され、完了します。
6. 階層ステート内の処理が実行されます。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します

例 9: 複数のソースステートから 1 つのターゲットステートへの、複数のトリガによるトランジション



例 8 と同じステートマシンです。現在のステートは soda_on（なおかつ階層ステート内のいずれかのサブステート）です。トリガイベント trigger_soda が発生し、この時ドリンクマシンのスイッチは OFF になっていて、switch_off が true に設定されています。ここでは以下のような処理が行われます。

1. システムは、soda_on からの有効なトランジションあるいはトランジションセグメントがあるかどうかをチェックします。
2. [switch_off] という条件が満たされているため、soda_on からジャンクションまでのトランジションセグメントが有効です。トリガイベント trigger_soda が発生しているためにジャンクションから beverage_off ステートへのセグメントも有効であるため、トランジションに定義された遷移が実行されます。
3. 階層ステート内での処理が実行されます。
4. soda_on ステートの Exit アクションである shut_down が実行されます。
5. soda_on から beverage_off へのトランジションにはトランジションアクションがないため、soda_on ステートはすぐにアクティブとなります。
6. beverage_off の Entry アクションである beverage_off が実行され、完了します。

   トリガイベントによって開始されたステートマシンの処理は、ここで終了します

2.5.5 階層ステートマシン

ステートマシンが起動されると、トランジションのための条件（コンディション）が評価されます。優先度は階層内のレベルによって決まり、最上位レベルの優先度が最も高くなります。つまり、最上位レベルのコンディションが最初に評価されます。ある階層ステートから離脱した時は、現在のサブステートからも離脱することになります。最初に最も内側のサブステートからの離脱が行われ、最も外

側の階層ステートからの離脱は最後に行われます。ある階層ステートに入る、つまりエントリする時には、最も外側の階層ステートから最も内側のステート（ベースステート）への順番で、Entry アクションが実行されます。つまり、最初に最も外側のステートにエントリし、最後に最も内側のステートにエントリすることになります。遷移が行われなかった場合は、内側から外側への順で Static アクションが行われます。つまり、最初に最も内側のサブステートの Static アクションが実行され、最も外側の階層ステートの Static アクションが最後に実行されることになります。

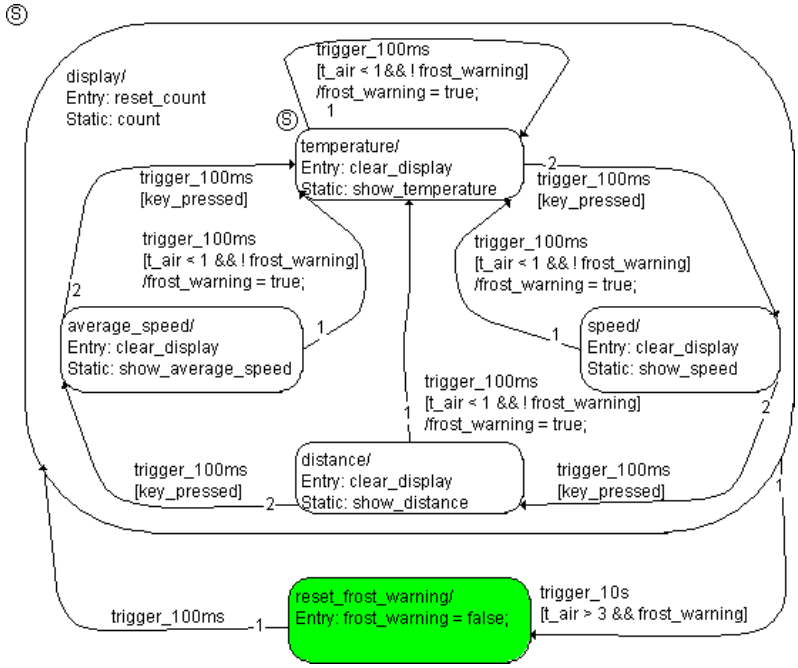
注記

以下に説明されている例は、階層ステートの Static アクションに関する最適化が行われないことを前提としています。この最適化が有効になっていると、ステートマシンの動作が変わります。詳しくは「コードサイズの最適化」（74 ページ）を参照してください。

例 10: ヒストリを持たない階層ステートへのトランジション

システムが階層ステートにエントリする際には、2 通りのケースがあります。1 つはこの例のように、階層ステートが、前回そこから離脱した際にどのサブステートがアクティブであったかを覚えていないケースです。そしてもう 1 つのケースは、例 11 のように、階層ステートがヒストリを持っていて前回アクティブであったサブステートにエントリするケースです。

ヒストリを持つかどうかは、階層ステートごとに設定することができます。ヒストリを持つ階層ステートに初めてエントリする際には、その階層ステートの開始ステートにエントリします。



階層ステートである display には例 3 のディスプレイ機能が含まれていてます。温度が 3 ℃を超えると、直ちにフロストワーニングがリセットされます。この処理は、最上位レベルにあるもうひとつのステートである reset_frost_warning によって行われます。display から reset_frost_warning へのトランジションは 10 秒毎に行われ、そこでワーニングはクリアされます。

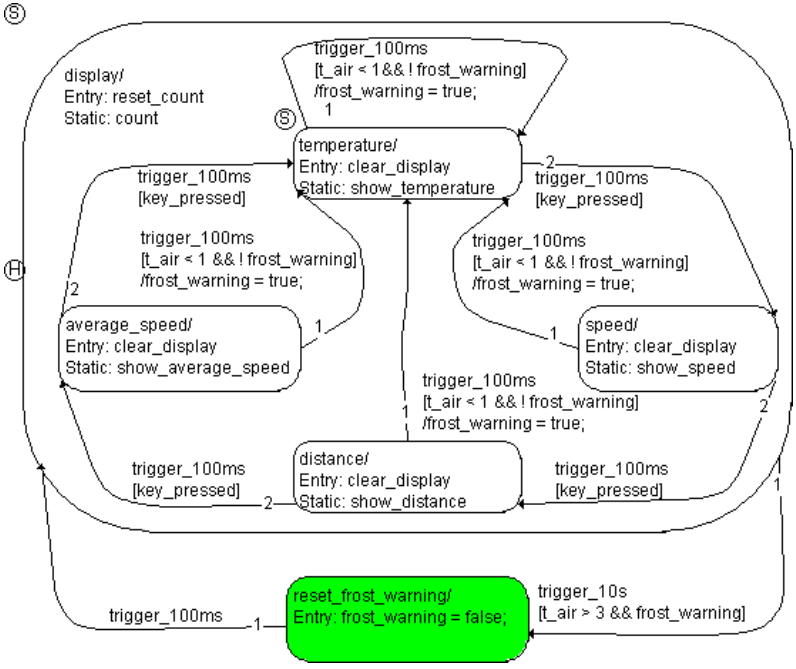
フロストワーニングが表示された（frost_warning = true）後、速度表示が選択され、システムは speed ステートとなりました。温度が 5 ℃まで上昇し、トリガイベント trigger_10s の発生によって display から reset_frost_warning への遷移が行われました。現在システムは reset_frost_warning ステートとなっています。ここでトリガイベント trigger_100ms が発生すると、以下のような処理が行われます。

1. システムは、reset_frost_warning からの有効なトランジションがあるかどうかをチェックします。
2. reset_frost_warning から display へのトランジションにはコンディションが定義されていないため、このトランジションは、トリガイベント trigger_100ms が発生するだけで有効となります。
3. reset_frost_warning には Exit アクションがないので、直ちに非アクティブ状態となります。

4. `reset_frost_warning` から `display` へのトランジションにはトランジションアクションが定義されていないため、階層ステートの `display` は直ちにアクティブとなります。
5. 階層ステート `display` の Entry アクションである `reset_count` が実行され、完了します。
6. 階層ステート内のサブステート `temperature` が開始ステートであるため、これがアクティブになります。
7. サブステート `temperature` の Entry アクションである clear_display が実行され、完了します。

   トリガイベント `trigger_100ms` によって開始されたステートマシンの処理は、ここで終了します。

例 11: ヒストリを持つ階層ステートへのトランジション



例 10 のステートマシンに似ていますが、このステートマシンはヒストリを持っています。フロストワーニングが発生してから `reset_frost_warning` ステートに移行するまでの状況や、開始ステートの割り当ては、例 10 と同じです。

現在システムは `reset_frost_warning` ステートとなっています。ここでトリガイベント `trigger_100ms` が発生すると、以下のような処理が行われます。

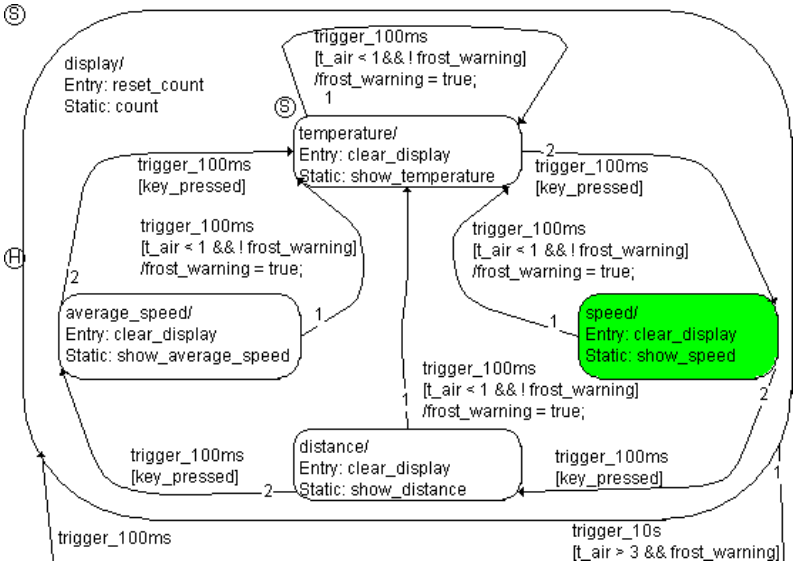
1. システムは、reset_frost_warning からの有効なトランジションがあるかどうかをチェックします。
2. reset_frost_warning から display へのトランジションには条件が定義されていないため、このトランジションは、トリガイベント trigger_100ms が発生するだけで有効となります。
3. reset_frost_warning には Exit アクションがないので、これは直ちに非アクティブ状態となります。
4. reset_frost_warning から display へのトランジションにはトランジションアクションが定義されていないため、階層ステートの display は直ちにアクティブとなります。
5. 階層ステート display の Entry アクションである reset_count が実行され、完了します。
6. display はヒストリを持っているため（図の 'H' マークがヒストリを示します）、サブステートの speed がアクティブとなります。このステートは、前回階層ステート display から離脱した時にアクティブであったステートです。
7. サブステート speed の Entry アクションである clear_display が実行され、完了します。

   トリガイベント trigger_100ms によって開始されたステートマシンの処理は、ここで終了します。

例 12: 階層ステート内のトランジション

トランジションが 1 つの階層ステート内で行われた場合、ステートマシンはその階層ステート内に留まります。そのため、階層ステートの Static アクションが実行され、さらにこのステートを含むすべての階層ステートの Static アクションも

実行されます。これらのアクションは、すべての Exit アクションが実行された後、トランジションアクションが実行される前に実行され、また最も内側の階層から外側へ向かう順番で実行されます。



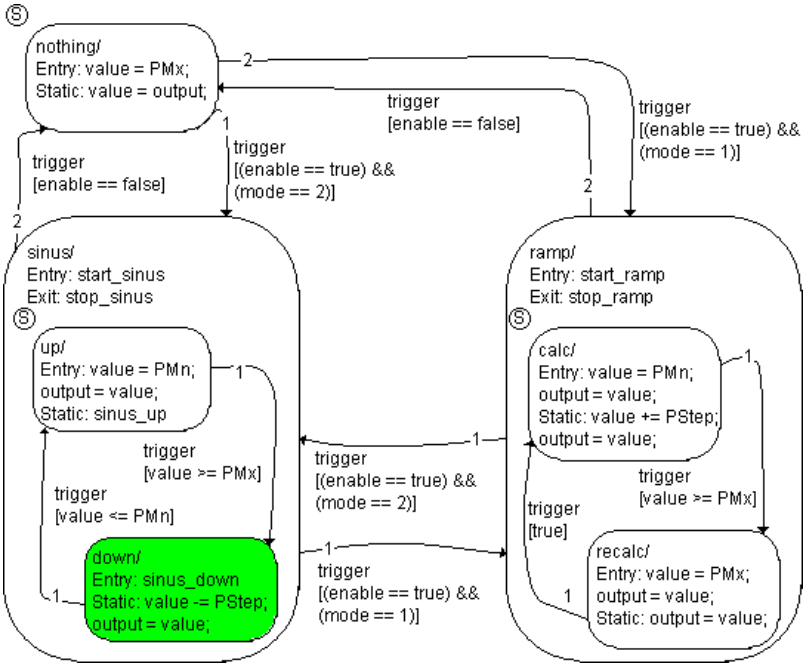
このステートマシンは例 11 と同じもので、現在 speed ステートがアクティブになっています。温度はまだ 5 ℃を保っていて、フロストワーニングは OFF（frost_warning が false）となっています。ここでトリガイベント trigger_100ms が発生し、このとき表示切替えスイッチは押された状態（key_pressed が true）となっています。ここでは以下のような処理が実行されます。

1. システムは、有効なトランジションがあるかどうかをチェックします。
2. もうひとつのトリガ trigger_10s によって階層ステート display から reset_frost_warning へのトランジションが評価されますが、これはここでの一連の処理に影響ありません。
3. サブステート speed から distance へのトランジションが評価されます。[key_pressed] という条件が満たされているので、トランジションが有効となります。
4. speed ステートには Exit アクションが定義されていないため、このステートは直ちに非アクティブとなります。
5. 階層ステート display はアクティブのままです。そのため、Static アクションである count が実行され、完了します。
6. speed から distance へのトランジションにはトランジションアクションが定義されていないため、サブステートの distance は直ちにアクティブとなります。

7. サブステート `distance` の Entry アクションである `clear_display` が実行され、完了します。

   トリガイベント `trigger_100ms` によって開始されたステートマシンの処理は、ここで終了します。

例 13: 階層ステート間のトランジション



このステートマシンは、データジェネレータとして機能します。`enable` が true にセットされると、のこぎり波（`ramp` ステート、`mode = 1` の場合）あるいは正弦波（`sinus` ステート、`mode = 2` の場合）を生成します。

現在は、階層ステート `sinus` のサブステート down がアクティブです。シグナルモード `mode` が 1 にセットされ、`enable` は `true` のままです。トリガイベントが発生すると、以下の処理が行われます。

1. システムは、有効なトランジションがあるかどうかをチェックします。階層ステート `sinus` からのトランジションの優先度は down からのトランジションよりも高いので、`sinus` からのトランジションが先に評価されます。
2. `sinus` から `nothing` へのトランジションが最上位の優先度を持っていますが、`[enable == false]` という条件が満たされていないので、このトランジションは有効にはなりません。

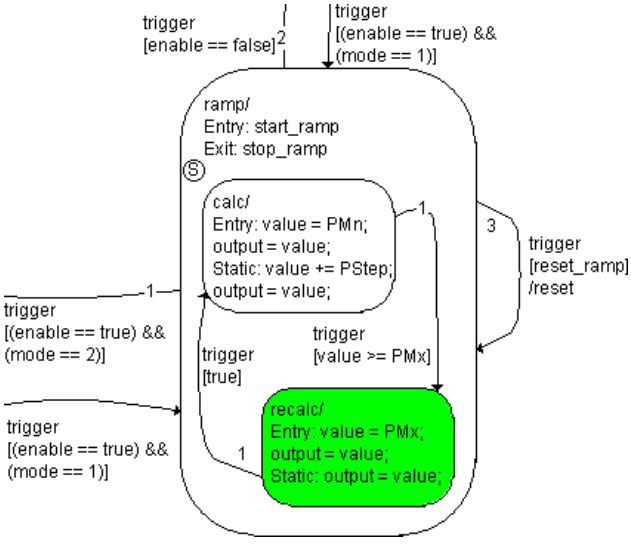
3. 次に sinus から ramp へのトランジションが評価されます。[(enable ==true) && (mode == 1)] という条件は true であるため、遷移が行われます。

   サブステート down から up へのトランジションの優先度は最下位であるため、評価されません。
4. サブステート down には Exit アクションがないので、Exit アクションなしにステートは非アクティブ状態となります。
5. 階層ステート sinus の Exit アクション stop_sinus が実行され、完了します。
6. 階層ステート sinus が非アクティブ状態となります。
7. sinus から ramp へのトランジションにはトランジションアクションがないため、続けて階層ステート ramp がアクティブになります。
8. ramp の Entry アクション start_ramp が実行され、完了します。
9. 階層内の開始ステートであるサブステート calc がアクティブになります。
10. calc の Entry アクション（value = PMn; output = value;）が実行され、完了します。

    トリガイベントによって開始されたステートマシンの処理は、ここで終了します。

例 14:　ループ

トランジションの始点と終点を同じステートとすることができます。このようなトランジションはループと呼ばれ、階層ステートのリセット機能として、よく使用されます。



例 13 のステートマシンの階層ステート `ramp` は、ループの形で表されたリセット機能を持っています。ループは、`ramp` からそれ自身へ向けたトランジションです。`ramp` 以外の部分は、ここでは省略されています。

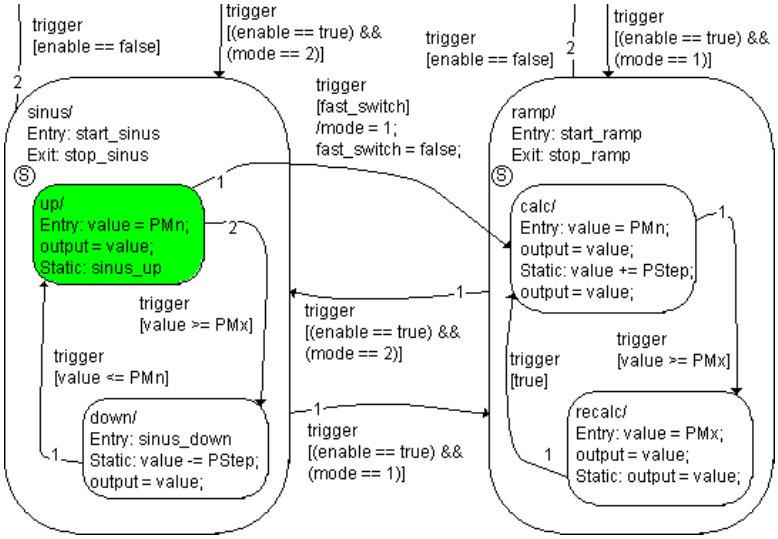
現在は、階層ステート `ramp` 内のサブステート `recalc` がアクティブです。トリガイベントが発生し、この時リセットボタンが押された状態（`reset_ramp = true`）となっていて、`enable` と `mode` の状態は、変わっていません。ここでは、以下のような処理が行われます。

1. システムは、有効なトランジションがあるかどうかをチェックします。
2. ループのトランジションが最上位の優先度を持っています。`[reset_ramp]` という条件が満たされているので、このトランジションが有効になります。

   他のトランジションについては評価されません。
3. サブステート `recalc` には Exit アクションがないので、直ちに非アクティブ状態となります。
4. 階層ステート `ramp` の Exit アクション `stop_ramp` が実行され、完了します。
5. 階層ステート `ramp` が非アクティブ状態となります。
6. ループのトランジションアクション `/reset` が実行され、完了します。
7. 階層ステート `ramp` が再びアクティブになります。
8. `ramp` の Entry アクション `start_ramp` が実行され、完了します。
9. 階層内の開始ステートであるサブステート `calc` がアクティブになります。
10. `calc` の Entry アクションが実行され、完了します。

    トリガイベントによって開始されたステートマシンの処理は、ここで終了します

例 15: 異なる階層ステート内にあるサブステート間のトランジション



このステートマシンは、例 13 のステートマシンに、`sinus` 内のサブステート `up` から `ramp` 内のサブステート `calc` へのトランジションを追加したものです。

現在は、階層ステート `sinus` 内のサブステート `up` がアクティブで、値 `value` は最大値 `PMx` 未満となっています。トリガイベントが発生し、`mode` は 2 のままで `enable` も `true` のままですが、固定スイッチが押された状態（`fast_switch = true`）となっています。ここでは以下のような処理が行われます。

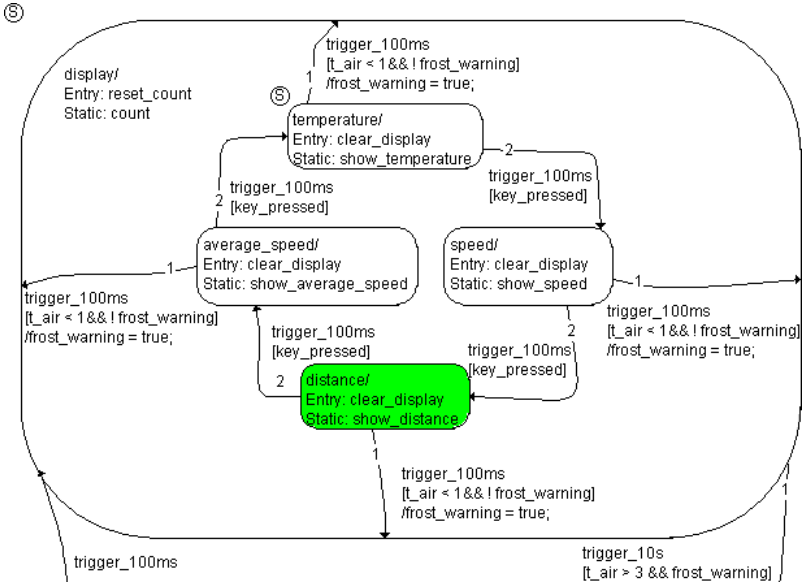
1. システムは、有効なトランジションがあるかどうかをチェックします。
2. 最初に、`sinus` から `nothing` へのトランジションと `sinus` から `ramp` へのトランジションが順に評価されます。これらの条件は共に満たされていないため、両方のトランジションは無効です。
3. 次にサブステート `up` からサブステート `down` へのトランジションが評価されます。条件 `[value >= PMx]` は満たされていないため、このトランジションも無効です。
4. サブステート `up` からサブステート `calc` へのトランジションは最低の優先度を持っているため、最後に評価されます。この条件 `[fast_switch]` は満たされているため、トランジションに定義された状態遷移が実行されます。
5. サブステート `up` には Exit アクションがないので、直ちに非アクティブ状態となります。
6. 階層ステート `sinus` の Exit アクション `stop_sinus` が実行され、完了します。
7. 階層ステート `sinus` が非アクティブ状態となります。

8. トランジションアクション（`/mode = 1; fast_switch = false;`）が実行され、完了します。
9. 階層ステート `ramp` がアクティブになります。
10. `ramp` の Entry アクションが実行され、完了します。
11. サブステート `calc` がアクティブになります。
12. `calc` の Entry アクションが実行され、完了します。

    トリガイベントによって開始されたステートマシンの処理は、ここで終了します。

例 16:　サブステートから上位の階層ステートへのトランジション

サブステートからのトランジションのターゲットが、別のサブステートではなく上位の階層ステートであった場合も、処理内容は前の例とほとんど同じです。サブステートを離脱し、階層ステートも離脱しますが、すぐに同じ階層ステートに再エントリします。そして階層ステートがヒストリを持っているかどうかにより、階層ステート内で最後にアクティブであったサブステート、あるいは階層ステート内の開始ステートにエントリします。この方法によって、前の例と異なる方法で表現されたフロストワーニングを例にとって説明します。



これは例 10 のステートマシンによく似ていて、フロストワーニング処理が階層ステート内のトランジションを利用して実現されています。現在のステートは `distance` で、`frost_warning` は `false` になっています。トリガイベント `trigger_100ms` が発生し、このとき温度は 0.5 ℃まで下がっています。表示切替えスイッチは押されていません。ここでは以下のような処理が行われます。

1. システムは、distance からの有効なトランジションがあるかどうかをチェックします。
2. もうひとつのトリガ trigger_10s によって階層ステート display から reset_frost_warning へのトランジションが評価されますが、これはここでの一連の処理に影響ありません。
3. distance から avarage_speed へのトランジションが評価されます。[key_pressed] という条件は満たされていないため、トランジションは無効です。
4. distance から display へのトランジションには [t_air < 1 && !frost_warning] というコンディションがあります。この中の 2 つの条件がすべて満たされているので、トランジションは有効となります。
5. distance ステートには Exit アクションがないので、これは直ちに非アクティブ状態となります。
6. display 階層ステートには Exit アクションがないので、これも直ちに非アクティブになります。
7. display 階層ステートが再度アクティブになります。
8. display 階層ステートの Entry アクションである reset_count が実行され、完了します。
9. /frost_warning = true というトランジションアクションが実行され、フロストワーニングのメッセージが表示されます。
10. 階層ステート display はヒストリを持っていないため、開始ステートである temperature がアクティブとなります。
11. サブステート temperature の Entry アクションである clear_display が実行され、完了します。

    トリガイベント trigger_100ms によって開始されたステートマシンの処理は、ここで終了します。

例 17: 状態遷移が発生しない場合

例 16 のステートマシンにおいて、現在、temperature ステートがアクティブになっています。温度は変化していません。トリガイベント trigger_100ms が発生し、このとき表示切替えスイッチは押されていません（key_pressed が false）。ここでは以下のような処理が実行されます。

1. システムは、有効なトランジションがあるかどうかをチェックします。
2. もうひとつのトリガ trigger_10s によって階層ステート display から reset_frost_warning への状態遷移が発生しますが、これはここでの一連の処理に影響ありません。
3. temperature から speed へのトランジションは、スイッチが押されていないため、無効です。
4. 現在 frost_warning = true であるために temperature から display へのトランジションの条件は false となり、このトランジションも無効です。
5. 他に評価すべきトランジションはありません。

6. サブステート `temperature` の Static アクション `show_temperature` が実行され、完了します。
7. 階層ステート `display` の Static アクションである `count` が実行され、完了します。

   トリガイベント `trigger_100ms` によって開始されたステートマシンの処理は、ここで終了します。

2.5.6 ステートマシンの動作についてのまとめ

ステートダイアグラムの初期化： システム全体の開始ステートがアクティブとなります。この開始ステートが階層ステートであった場合、この階層内の開始ステートも同様にアクティブとなります。このとき Entry アクションは実行されません。

ステートへのエントリ：

1. 該当ステートの上位レベルの階層ステートが非アクティブであった場合、この 1 ～ 4 のステップはその階層ステートについても実行されます。
2. ステートがアクティブになります。
3. Entry アクションが実行されます。
4. 必要に応じてサブステートへの暗黙的なエントリが行われます。
   4.1 ステートが下位のダイアグラムを含み、かつヒストリを持っていて、初期化後にそのステートが 1 回でもアクティブになったことがある場合は、最後にアクティブであったサブステートがアクティブになり、その Entry アクションが実行されます。
   4.2 ステートが下位のダイアグラムを含んでいてもヒストリを持っていない場合、あるいは持っていても初期化後にそのステートが 1 回もアクティブになっていない場合は、そのステート内の開始ステートであるサブステートがアクティブになり、その Entry アクションが実行されます。

ステート（ベースステート）の実行：

1. 現在のステートからのトランジションと、その上位レベルのステートからのトランジションが、優先度に従った順番で評価されます。
2. 有効なトランジションが見つかると、それが実行されます。これによってそのステートの処理は終了します。
3. 有効なトランジションが 1 つも存在しない場合は、ステートの Static アクションが実行されます。
4. そのステートが上位レベルのステートに含まれるサブステートであった場合、上位レベルのステートの Static アクションも実行されます。

ステートからの離脱：

1. ステートの中にアクティブなサブステートが含まれている場合、そのステートとサブステートの両方の Exit アクションが実行されます。Exit アクションは、最も内側（下位）のベースステートから順に実行されます。
2. ステートが非アクティブになります。

トランジションに定義された遷移条件の評価と遷移の実行：

カレントステートに複数のトランジションが存在する場合、各トランジションは割り当てられた優先度の高い順に評価されます。階層ステートからのトランジションは、その階層ステート内のサブステートからのトランジションよりも高い優先度も持ちます。

1. トランジションあるいはトランジションセグメントが評価されます。
2. 評価したトランジションあるいはセグメントが無効であった場合、次に優先度が高いトランジションあるいはセグメントが評価されます。
3. 有効なトランジションあるいはセグメントが見つかると、次に行われるステップは、トランジションあるいはセグメントの接続先によって異なります。

   ステートに接続される場合：

   3.1 ここまでに評価されたもの以外に評価されるトランジションやセグメントはありません。

   3.2 ソースステート内のサブステートからの離脱が行われます（「ステートからの離脱」を参照してください）。

   3.3 ソースステートからの離脱が行われます。

   3.4 トランジションアクションが実行されます。

   3.5 トランジションのターゲットステートが、システムステートとなります（「ステートへのエントリ」を参照してください）。

   ジャンクションに接続される場合：

   3.1 ジャンクションからステートに接続されているトランジションセグメントが、1 ～ 3 の手順で評価されます。

   3.2 ジャンクションから先のトランジションセグメントがすべて無効であった場合、システムは、ジャンクション手前のソースステートに戻り、次に優先度の高いトランジションあるいはセグメントについて、1 ～ 4 のステップが実行されます。
4. ステートからのすべてのトランジションあるいはトランジションセグメントが無効であった場合、ステート間の遷移は行われず、システムは同じステート内に留まります。

上記のシーケンスは、図 2-13 のように表されます。

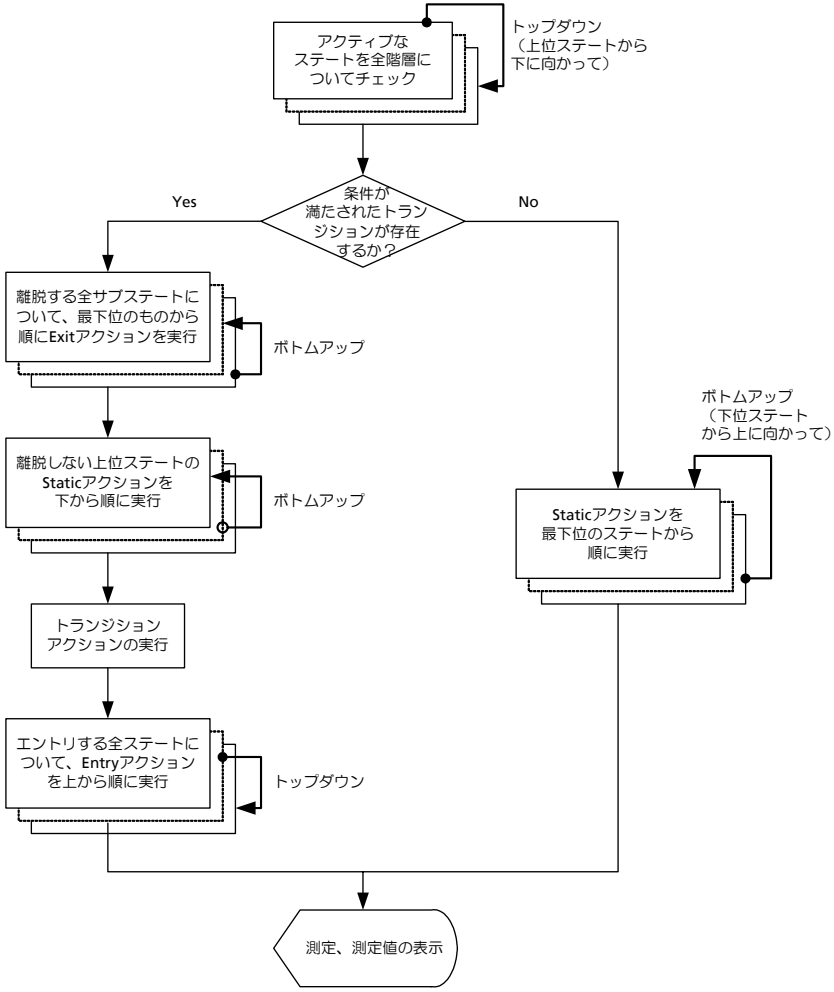
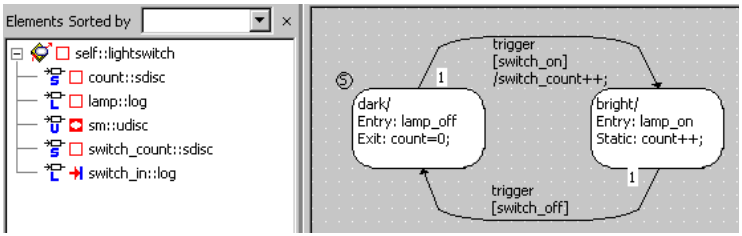


図 2-13 ステートマシンの評価手順

2.5.7 生成されるコードの例



このシンプルなステートマシンから生成された C コードの一部を、以下に示します。

```
/*
 *----------------------------------------------------------------------------
 *      Defines
 *----------------------------------------------------------------------------
 */

#define bright 1
#define dark 0
/*

/**************************************************************************
 * BEGIN: Function definitions - Algorithms
 *-------------------------------------------------------------------------*/
void LIGHTSWITCH_IMPL_trigger(struct LIGHTSWITCH_IMPL_Obj *self)
{
   switch (self->sm->val)
   {
      default:
      case dark :
      {
         if (LIGHTSWITCH_IMPL_switch_on (self))
         {
            self->count->val = (sint32)0;
            self->switch_count->val++;
            self->lamp->val = (uint8)true;
            self->sm->val = bright;
            return;
         }
         return;
      }
      case bright :
      {
         if (LIGHTSWITCH_IMPL_switch_off (self))
         {
            self->lamp->val = (uint8)false;
            self->sm->val = dark;
            return;
         }
         self->count->val++;
         return;
      }
   }
}
uint8 LIGHTSWITCH_IMPL_getlamp(struct LIGHTSWITCH_IMPL_Obj *self)
{
   return (self->lamp->val);
}
uint8 LIGHTSWITCH_IMPL_setswitch_in(struct LIGHTSWITCH_IMPL_Obj *self,
   uint8 parm)
{
   return ((uint8)(self->switch_in->val = parm));
}
```

各ステートを定義する #define 文

dark の Exit アクション

dark から bright へのトランジション

トランジションアクション

bright の Entry アクション

dark の Entry アクション

bright から dark へのトランジション

bright の Static アクション

出力 lamp のパブリックメソッド

入力 switch_in のパブリックメソッド

2.5.8 ステートマシンの最適化

一般に、同じファンクションを定義するためには何通りかの方法がありますが、コード生成やビルド処理のオプションを設定する場合も同様です。

ステートマシンのコードが生成される際、ステートやトランジションに定義されたアクションやコンディションが、直接各所に挿入される場合もあり（インライニング）、またある条件においては、独立したメソッドとして生成される場合もあります（アウトライニング）。アクションまたはコンディションのアウトライン化が行われる条件は、以下のとおりです。

1. ステートマシンが含まれるプロジェクトの“Project Properties”ウィンドウの“Statemachine”ノードにおいて、ステートマシンの最適化オプションである **Outline Generated Methods (may be changed locally)** がオンになっていること

   このオプション設定は、そのプロジェクトに含まれるすべてのステートマシンおよびすべてのタイプの実験（物理、量子化、実装）に適用されます。

2. ステートマシンのインプリメンテーションエディタにおいて、**Outline automatically generated methods for State Machines** オプションがオンになっていること

注記

1. の条件が満たされていない場合、プロジェクト内のすべてのステートマシンについてアウトライニングは行われません。
1. の条件が満たされていても、2. の条件が満たされていないステートマシンについては、アウトライニングは行われません。

上記の 2 つの条件が満たされていると、コード生成時において、アウトライニングを行う場合と行わない場合のコードサイズが比較され、コードサイズが小さくなる場合に限りアウトライニングが行われます。

アクションとコンディション（またはそれらの一部分）が独立したダイアグラムで記述されている場合、そのコードは、独立したプライベートメソッドとして生成されるか（アウトライニング）、または必要な箇所に挿入されます（自動インライニング）。

自動イントライニングが行われる条件は、以下のとおりです。

1. ステートマシンが含まれるプロジェクトの“Project Properties”ウィンドウの“Statemachine”ノードにおいて、ステートマシンの最適化オプションである **Auto-inline private methods (Smaller code-size - may be changed locally)** がオンになっていること

   このオプション設定は、そのプロジェクトに含まれるすべてのステートマシンおよびすべてのタイプの実験（物理、量子化、実装）に適用されます。

2. ステートマシンのインプリメンテーションエディタにおいて、**Auto-inline private methods (Smaller code-size)** オプションがオンになっていること

注記

1. の条件が満たされていない場合、プロジェクト内のすべてのステートマシンについ自動インライニングは行われません。
1. の条件が満たされていていても、2. の条件が満たされていないステートマシンについては、自動インライニングは行われません。

上記の 2 つの条件が満たされていると、コード生成時において、自動インライニングを行う場合と行わない場合のコードサイズが比較され、コードサイズが小さくなる場合に限り自動インライニングが行われます。通常、自動インライニングが行われるのは、小さなプライベート関数や呼び出し回数の少ない関数に限ります。関数ごとに条件がチェックされ、条件に該当する関数のみがインライニングされます。

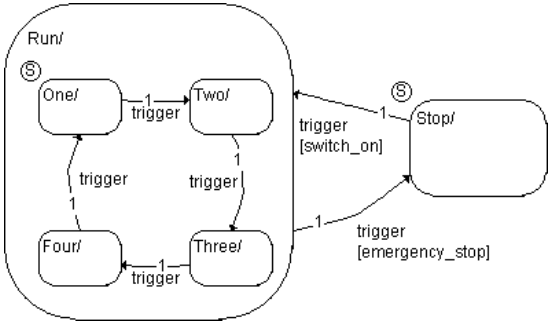
ステートマシンの最適化は、要件に応じて以下の 3 つの観点から行うことができます。

- 応答時間
- 実行速度
- コードサイズ

応答時間の最適化

応答時間が最も重要視される場合は、階層構造、およびトランジションの優先度を利用します。高速で処理すべきアクションを最上位の階層レベルに配置すると、効率的なコードを生成され、応答時間を最短化することができます。

以下に例をあげます。



緊急停止ボタンが押された時（`emergency_stop = true`）は、システムはできる限り速やかに停止、つまり `Stop` ステートに移行しなければなりません。このような場合、階層ステート `Run` から `Stop` ステートへのトランジションを定義すれば、このトランジションは階層内で最も高い優先度を持つことなるので、最優先で評価されます。

いずれのサブステートがアクティブであっても、緊急ボタンが押されれば Run から Stop へのトランジションは必ず評価され、実行されます。

これを、各サブステートから Stop への直接のトランジションとして定義すると、実行速度は速くなりますが、1 つで済むトランジションが 4 つ必要なため、メンテナンス効率は低下します。

緊急度の高いイベントを別のトリガとして定義する（例：`emergency_stop = true`）ことによる応答時間の最適化も可能ですが、この場合、追加されたトリガ用のコードによってコードサイズが増大します。

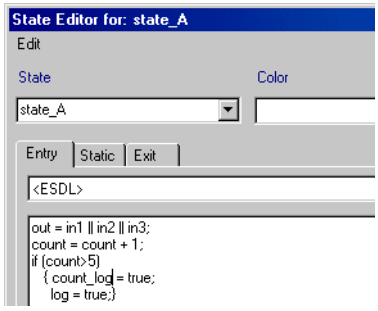
実行速度の最適化

実行速度が最も重要視される場合は、いくつかの最適化オプションを、単独で、または組み合わせて使用することにより、効率のよいコードを生成することができます。

アクション／コンディション：

複数のアクションまたはコンディション内に、一部または全体が同じである複数の機能パーツが存在する場合、この部分を、実行速度、あるいはコードサイズについて最適化することができます。「実行時間の最適化」は、各アクション／コンディションのコードを、コード生成時に各所に挿入することで実現されます。これによって関数呼び出しが省略されます。ただしこの場合、同じコードが複数回生成されるため、メモリ使用量が増大する、というデメリットがあります。

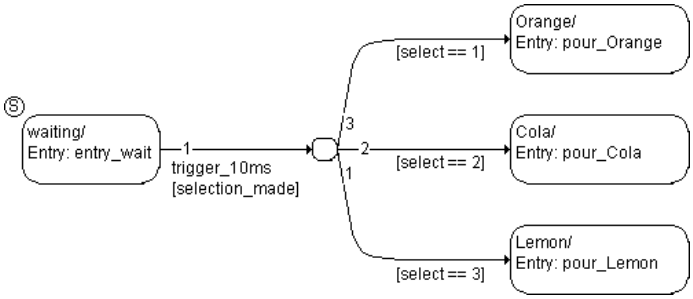
この最適化を行うには、下の例のように、各アクション／コンディションのコードをステートまたはコンディション内に明示的に記述し、さらに **Outline Generated Methods (may be changed locally)** および **Outline automatically generated methods for State Machines** オプション）をオフにします（71 ページ参照）。



また、自動インライニング機能（71 ページ参照）をオンにすることにより、さらに実行速度を最適化することができます。このオプションがオンになっていると、独立したダイアグラムで記述されたアクション／コンディションも、適切な場合はインラインコードとして展開されます。

このようなアクション／コンディションについて、インプリメンテーションエディタで **Inline** オプションをオンにすることにより、強制的にインラインニングを行うことができます。

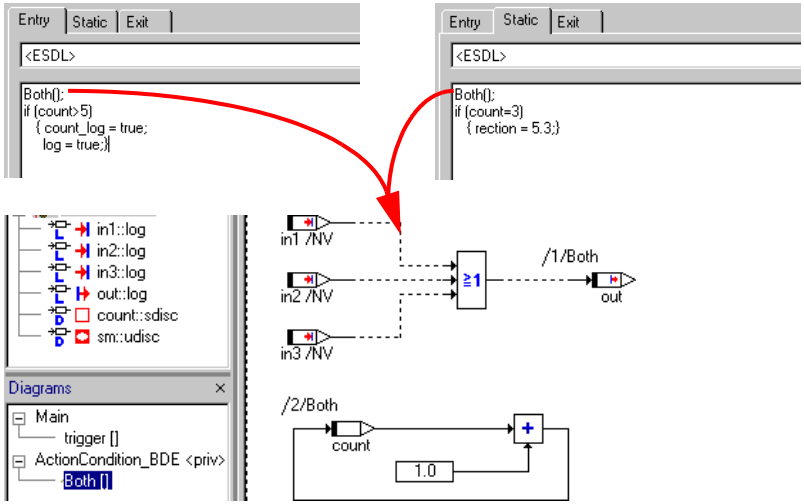
ジャンクション： 1 つのステートを始点とする複数のトランジションについて、それぞれのコンディションに共通な部分がある場合、ジャンクションを使用することによって実行時間を短縮できます。共通な条件は、ソースステートから 1 番目のジャンクションまでのトランジションセグメントのコンディションとして割り当てます。この条件が満たされていなければ、その先のセグメントは一切評価されません。



コードサイズの最適化

アクション／コンディション： コードサイズの最適化は、各アクション／コンディション内の共通部分を、独立したプライベート関数としてコード生成し、それを必要に応じて呼び出すことによって実現されます。

これを行うには、次の図のように、繰り返し使用される部分を 1 つの独立したダイアグラム内にメソッドとして記述し、各メソッドをアクションから呼び出すようにします。



別の方法として、これらのコードをステートやトランジション内に直接記述し、アウトライニング機能を利用することも可能です。

これらの方法により、実際のコードは 1 箇所にしか生成されなくなります。ただしこの場合、関数呼び出しが増えます。

それに対し、小規模のプライベート関数や呼び出し箇所の少ない関数などについては、インラインコードの挿入によってコードサイズを縮小できるというメリットがあります。この「自動インライニング」機能（71 ページ参照）は、**Auto-inline private methods (Smaller code-size - may be changed locally)** および **Auto-inline private methods (Smaller code-size)** オプションをオンにすることによって有効になります。この機能により、アクションとコンディションのコードサイズを効率よく最適化できます。

階層ステートの Static アクション： 階層ステートの Static アクションについては、更なる最適化オプションが用意されています。

デフォルトでは、階層ステートの Static アクションは、その階層を離脱しないすべてのトランジションごとに生成され、これは階層の各サブステートに関しても同様です。そのため、大きな階層の場合はコードサイズが非常に大きくなってしまいます。

このような場合、プロジェクトのコード最適化オプションのひとつである **Optimize Static Actions (Restricted Modeling)** オプションをオンにすることによって、階層ステートの Static アクションが各サブステートごとに 1 つだけ生成されるようにすることができ、コードサイズを縮小することが可能です。

ただしこの最適化は、モデルの構造によっては適用できない場合があります。たとえば、1 つのステートマシン内に、階層ステートから直接離脱するサブステートが含まれる場合、このトランジションは、そのサブステートからの他のすべてのトランジションよりも優先度が高くなければなりません。そうでない場合、以下のメッセージが出力されてコード生成は中断されます。

```
ERROR(YSm72): higher priority transitions do not exit
hierarchy state "HState", but this transition does.
```

コード生成時にこの最適化が行われると、ステートマシンの動作は以下のように変更されます。

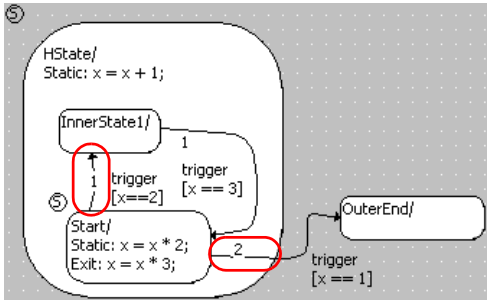
注記

これらの最適化によってステートマシンの動作が変更されるため、既存のステートマシンに関してこの最適化オプションを有効にする際は、十分な注意が必要です。

- 階層ステートの Static アクションは、サブステートからの各トランジションに定義されたコンディションが評価される前に実行されます。
- いずれの遷移も発生しなかった場合、階層ステートの Static アクションは、サブステートの Static アクションの前に実行されます。
- 遷移が発生した場合、階層ステートの Static アクションは、サブステートの Exit アクションの前に実行されます。

以下に、2 つのステートマシンを例に挙げてこの最適化の効果を説明します。両例とも、使用されているステートマシンには、2 つのサブステート（`Start` および `InnerState1`）を含む階層ステート（`HState`）と、ベースステート（`OuterEnd`）が含まれています。`Start` には 2 つのトランジションが定義されていて、そのうちの 1 つ（`Start` → `OuterEnd`）は、階層ステート `HState` から離脱するためのものです。

1 番目の例では、`Start` から `OuterEnd` へのトランジションの優先順位は、`Start` から `InnerState1` へのトランジションよりも高くなっています。この場合、**Optimize Static Actions (Restricted Modeling)** オプションがオン／オフのどちらであっても、コード生成は正常に行われます。



以下に、このオプションがオンである場合とオフである場合にそれぞれ生成されるコードを示します。太字で示されている部分が、HState ステートの Static アクションです。

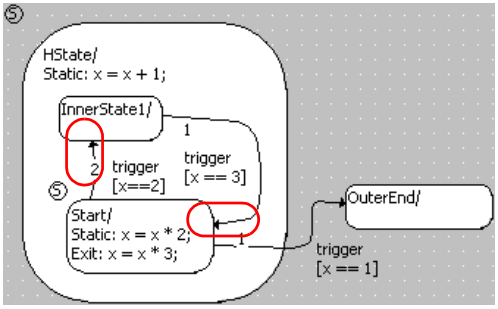
オプションがオフの場合

```
case Start :
{
  if (x == 1.0)
  {
    x = x * 3.0;
    sm = OuterEnd;
    return;
  }
  if (x == 2.0)
  {
    x = x * 3.0;
    x = x + 1.0;
    sm = InnerState1;
    return;
  }
  x = x * 2.0;
  x = x + 1.0;
  return;
}
case InnerState1 :
{
  if (x == 3.0)
  {
    x = x + 1.0;
    sm = Start;
    return;
  }
  x = x + 1.0;
  return;
}
```

オプションがオンの場合

```
case Start :
{
  if (x == 1.0)
  {
    x = x * 3.0;
    sm = OuterEnd;
    return;
  }
  x = x + 1.0;
  if (x == 2.0)
  {
    x = x * 3.0;
    sm = InnerState1;
    return;
  }
  x = x * 2.0;
  return;
}
case InnerState1 :
{
  x = x + 1.0;
  if (x == 3.0)
  {
    sm = Start;
    return;
  }
  return;
}
```

それに対して 2 番目の例では、Start から OuterEnd へのトランジションの優先順位は Start から InnerState1 へのトランジションよりも低くなっています。この場合は、**Optimize Static Actions (Restricted Modeling)** オプションがオンになっていると、コード生成は行えません。



オプションがオフの場合

```
case Start :
{
  if (x == 2.0)
  {
    x = x * 3.0;
    x = x + 1.0;
    sm = InnerState1;
    return;
  }
  if (x == 1.0)
  {
    x = x * 3.0;
    sm = OuterEnd;
    return;
  }
  x = x * 2.0;
  x = x + 1.0;
  return;
}
case InnerState1 :
{
  if (x == 3.0)
  {
    x = x + 1.0;
    sm = Start;
    return;
  }
  x = x + 1.0;
  return;
}
```

オプションがオンの場合

```
ERROR(YSm72): higher priority
transitions do not exit
hierarchy state "HState", but
this transition does.
```

階層コード生成： 階層ステートマシンのコード生成には 2 とおりの方法があります。

「フラットコード生成」を行うと、各階層がフラットに展開されます。つまり、すべてのベースステートおよびトランジション用に、1 つの `switch` 文が生成されます。

それに対して「階層コード生成」では、階層構造に従い、ネストされた複数の `switch` 文が生成されます。この機能を使用するには、以下のオプションをオンにする必要があります。

- プロジェクトオプションの“Statemachne" ノード設定：**Hierarchical Code Generation (may be changed locally)**
- ステートマシンのインプリメンテーションエディタ：**Hierarchical code generation for State Machines**

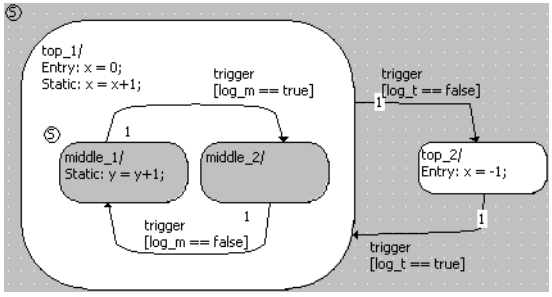
上記の 1 番目のオプションがオフになっていると、階層コード生成は一切行われません。1 番目のオプションがオンになっている場合、各ステートマシンの 2 番目のオプションのオン／オフによって、個々のステートマシンの階層コード生成を行うかどうかが決まります。

階層ステートからのトランジションのコードは、フラットコード生成の場合は、内包されるベースステートごとに 1 箇所ずつ生成されますが、階層コード生成においては、1 箇所にのみ生成されるので、コードサイズが大きく削減されます。この削減率は 30% 程度にまで及びます。実験においては、ステートマシンの動作はどちらのコードでも変わりません。

注記

トランジションや Statifc アクションを含まない階層ステートの場合は、コードサイズの削減率は 1 ～ 2% 程度です。

以下に、生成されるコードの違いを、具体例をあげて説明します。



注記

コードの削減率は、生成された C ファイルのサイズではなく、最終的に生成される実行ファイルのサイズを基にしています。

以下の**太字**の部分が top_1 から top_2 へのトランジションに相当します。

階層コード生成

```
switch (self->
      _ASCET_smLevel_0->val)
{
 case top_2 :
 {
  if (self->log_t->val)
  {
   self->x->val = 0.0;
   self->
         _ASCET_smLevel_0->
         val = top_1;
   self->sm->val = middle_1;
   return;
  }
  return;
 }
 default:
 case top_1 :
 {
  if (!self->log_t->val)
  {
   self->x->val = -1.0;
   self->
         _ASCET_smLevel_0->
         val = top_2;
   self->sm->val = top_2;
   return;
  }
  switch (self->sm->val)
  {
   default:
   case middle_1 :
   {
    if (self->log_m->val)
    {
     self->x->val = self->
               x->val + 1.0;
     self->y->val = -1.0;
     self->sm->val =
               middle_2;
```

フラットコード生成

```
switch (self->sm->val)
{
 default:
 case middle_1 :
 {
  if (!self->log_t->val)
  {
   self->x->val = -1.0;
   self->sm->val = top_2;
   return;
  }
  if (self->log_m->val)
  {
   self->x->val = self->
               x->val + 1.0;
   self->y->val = -1.0;
   self->sm->val = middle_2;
   return;
  }
  self->y->val = self->y->
               val + 1.0;
  self->x->val = self->x->
               val + 1.0;
  return;
 }
 case middle_2 :
 {
  if (!self->log_t->val)
  {
   self->x->val = -1.0;
   self->sm->val = top_2;
   return;
  }
  if (!self->log_m->val)
  {
   self->x->val = self->
               x->val + 1.0;
   self->sm->val = middle_1;
   return;
  }
```

```
      return;
     }
     self->y->val = self->
                y->val + 1.0;
     self->x->val = self->
                x->val + 1.0;
     return;
    }
    case middle_2 :
    {
     if (!self->log_m->val)
     {
      self->x->val = self->
               x->val + 1.0;
      self->sm->val =
                middle_1;
      return;
     }
     self->x->val = self->
                x->val + 1.0;
     return;
    }
   }
  }
 }
```

```
  self->x->val = self->x->
                val + 1.0;
  return;
 }
 case top_2 :
 {
  if (self->log_t->val)
  {
   self->x->val = 0.0;
   self->sm->val = middle_1;
   return;
  }
  return;
 }
}
```

トリガとトリガ引数： 入力・出力の代わりにトリガ引数を用いて ASCET コンポーネントとの通信を行うと、RAM の静的使用量が少なくなります。詳細は、次の項に説明されています。

2.5.9 ステートマシンクラス

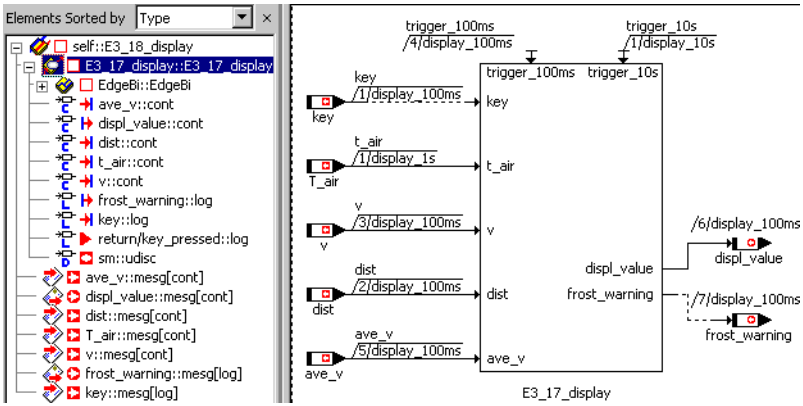
ステートマシンは、特殊な記述手段を備えたクラスです。トリガ、条件、およびアクションは、特殊なメソッドとしてモデリングされます。

- トリガは、戻り値を伴わないパブリックメソッドです。ステートマシンは、トリガが呼び出されると必ず実行されます。
- コンディションは、論理型の戻り値を戻すプライベートメソッドです。
- アクションはプライベートメソッドです。通常は、引数も戻り値も持ちません。

他の ASCET コンポーネントとの通信が必要な場合、上記のメソッドに引数を定義することが可能です。

入力と出力はステートマシンを他のコンポーネントに統合するために使用されます。入力値は内部変数にバッファリングされるので、そのステートマシン内のすべての処理に使用することができます（メソッドの引数がそのメソッド内でしか

使えないのとは対照的です)。出力もやはりバッファリングされるので、特別な処理を実行しなくても読みとることができます。各入力と出力には、シーケンスコールを定義する必要があります。(6.2 項を参照してください)



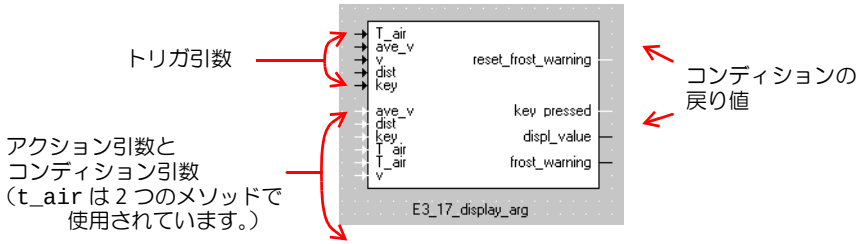
しかしこのような方法で外部通信を行うと、各入力/出力ごとに静的な変数がRAM 上に確保されるので、メモリの使用量が増えてしまいます。このようなRAM の静的使用量を減らすためには、トリガに引数を持たせることができます。また独立したブロックダイアグラムで記述されたアクションやコンディションにも同じく引数を持たせることが可能です。外部との通信は、この引数によって行われます。この場合、C 関数の引数として、RAM を静的に使用しないスタック変数が生成されます。このダイナミックな RAM エリアは、一時的に使用されるだけです。

だたし、ここでは以下の点に注意してください。

- トリガはパブリックメソッドで、その引数はステートマシンの外部で記述することができます。レイアウトエディタにおいて、トリガの引数は黒い引数コネクタとして表示されます。

- アクションとコンディションはプライベートメソッドであるため、それらの引数はステートマシンの外部からは使用できません。レイアウトエディタにおいて、これらの引数は白い引数コネクタとして表示されます。

  ブロックダイアグラムで記述されたアクションあるいはコンディション内においてトリガ引数を使用する必要がある場合は、そのトリガ引数と同じ型と名前を持つ引数を、メソッド内に追加してください。



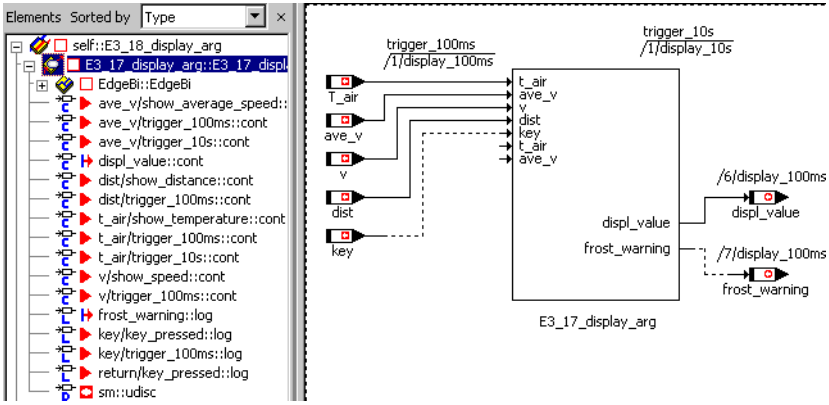
各引数は、その型と名前を基準に表示されるため、もしトリガとアクション／コンディション内において同じ名前で型の異なる引数があると、ワーニングが発生します。また、もしアクションやコンディションに定義されている引数がトリガに定義されていないと、エラーメッセージが表示されます。

さらに、トリガ引数をアクションやコンディション内で使用するには、以下のようなルールがあります。

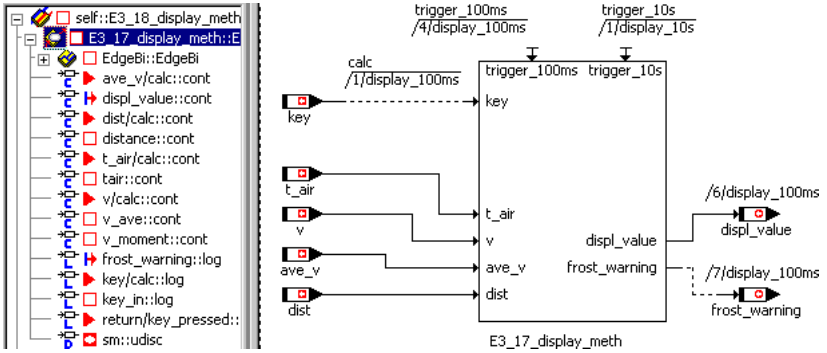
- ステートの Entry アクションで使用されるすべてのトリガ引数は、そのアクション内、およびそのステートを終点とするトランジションに割り当てられたすべてのトリガ内で定義されている必要があります。このアクションは、これらのトリガによって実行されるためです。
- ステートの Exit アクションで使用されるすべてのトリガ引数は、そのアクション内、およびそのステートを始点とする各トランジションに割り当てられたすべてのトリガ内で定義されている必要があります。このアクションは、これらのトリガによって実行されるためです。
- ステートの Static アクションで使用されるすべてのトリガ引数は、そのアクション内、およびステートマシン内のすべてのトリガ内で定義されている必要があります。このアクションは、トランジションの発生を伴わなかった各トリガによって実行されるためです。
- コンディションあるいはトランジションアクションで使用されるすべてのトリガ引数は、そのコンディションあるいはアクション内、およびそのトランジションに割り当てられたトリガ内で定義されている必要があります。これらのコンディションやアクションは、このトリガによって実行されるためです。

上記の条件のうちのいずれかが守られていない場合は、エラーメッセージが表示されます。

他のコンポーネントへの組み込みが終了した後は、そのコンポーネントからトリガ引数に値を代入することが出来ます。入力と出力の場合と異なり、すべての引数について 1 つのシーケンススタートのみを使用します。



さらに、他のクラスと同様、ステートマシンにも通常のパブリックメソッドを定義することもでき、これによって更なる可能性が広がります。これらのメソッドは、ステートやトランジションの外側から、さらにはステートマシンの外側から開始することが可能です。外部コンポーネントとの通信には、入力と出力の代わりに引数と戻り値を使用することができます。この場合も、メソッド全体で必要なシーケンススタートは 1 つだけですが、これは特に実行時間の節約にはなりません。また、ステートマシンに必要な入力値を用意することも可能です。



ステートマシンのパブリックメソッドの他の利用方法としては、たとえばステートマシンの内側と外側の両方から開始される可能性のあるリセット処理、あるいはステートマシンの内側と外側で発生するイベントをカウントするカウンタなどが考えられます。ステートマシンの一部として組み込まれたクラスは、異なるタイムフレームで処理されます。ですから、1 つのステートマシンに、固有のトリガを持たないもう 1 つのステートマシンを組み込んでパブリックメソッドとして実行し、異なるタイムフレームで処理させることが可能となるわけです。

3 型とエレメント

1 つのコンポーネント内のすべてのアルゴリズムは、「エレメント」を対象として実行されます。エレメントはデータを内包していて、そのデータにアクセスしたり計算値（例、特性カーブにおける補間）を戻したりするためのインターフェースが提供されています。各エレメントには明確に定義された「型」が割り当てられます。同じ型のエレメントは複数存在する可能性があるので、エレメントをその型の「インスタンス」と呼ぶことができます。

ASCET には離散変数や連続変数、配列、マトリックスあるいは特性カーブやマップなど、さまざまな「基本型」があり、それらは直接使用することができます。またユーザー定義の型、つまり「クラス」をシステムに追加することもできます。クラスは「複合型」です。通常、クラスは他の型（基本型や他の複合型）を組み合わせて構築されているため、その構造は複雑です。これらの型は下図のように分類されます。

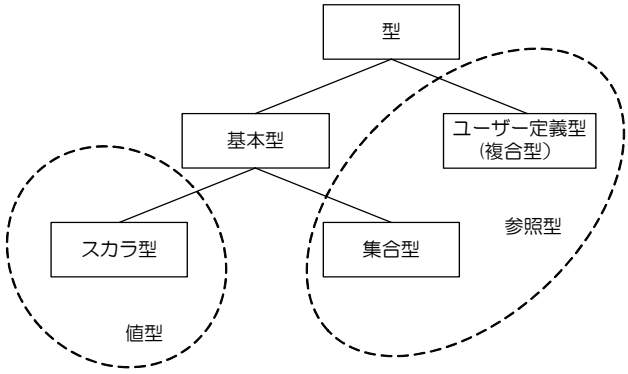


図 3-1　ASCET のデータ型の分類

ASCET でのモデリングは物理レベルで行われるので、モデリングに使用される型（「モデルデータ型」とも呼ばれます）も「物理」型です。この物理型が、コード実装の段階において、モデリング内容とは独立して定義された実装情報（「インプリメンテーション」と呼ばれます）に基づき、特定の実装データ型（例、`unsigned int8`）に変換されます。

エレメントの物理定義には、以下の情報が必要です。

- エレメントの名前
- モデル型
- エレメントの種類
- エレメントのスコープ

以上の各カテゴリの設定内容について、以降の項で詳しく説明します。

エレメントを定義する際、物理単位やコメントなどの補足情報をエレメントに付加することができます。この情報は、物理モデルに対しては何も影響を与えません。

3.1 基本モデル型

ASCET では、基本モデル型はスカラ型と集合型の 2 つに分類されます。

3.1.1 スカラ型

最も一般的な基本モデル型は、スカラ型です。ASCET では、以下の集合型を使用することができ、ASCET のユーザーインターフェース上ではそれぞれ以下のようなシンボルで示されます。

- *Continuous*（連続型）：無限大の値の範囲と任意の精度を持つ連続物理値に使用され、温度や速度などのモデル変数に適しています。通常、「`cont` モデル型」と呼ばれます。

- Signed discrete（符号付き離散型）：任意の大きさを持つモデル整数値に使用されます。通常、「`sdisc` モデル型」と呼ばれます。

- Unsigned discrete（符号なし離散型）：負の値以外のすべての整数値となるモデル整数値に使用されます。通常、「`udisc` モデル型」と呼ばれます。

- *Logical*（論理型）：モデルの論理情報に使用されます。ある制御系が現在アクティブであるかどうか、といった情報の管理に適しています。通常、「`log` モデル型」と呼ばれます。

これら 4 種類の基本スカラ型は、「値型」として扱われます。つまり、これらの型を持つエレメントが使用されるときは、そのエレメント自身ではなく、その値が使用されます。このうちの「算術型」（`cont`、`sdisc`、`udisc`）は、必要に応じて互いに型変換が行われます。

集合型は、複合型（クラス）と同様のインターフェース（つまりアクセスメソッド）を持っていますが、基本モデル型用のメソッドは固定されていて、変更はできません。

基本スカラ型エレメントに格納された値へのアクセス（エレメントへの値の書き込み、およびエレメントの現在の値の読み取り）には、以下の 2 つの単純なアクセスメソッドを使用します。

- `set (type a)`：1 つの値（この例では `a`）を入力し、エレメントの値にその値を上書きします。値の型がエレメントの型に適さない場合には、型変換が自動的に行われます。
- `get()`：エレメントの現在値を戻します。戻される値の型は、そのエレメントの型と同じです。

基本型のアクセスメソッドは、エレメントの名前が式に使用されている場合や代入が行われる際に自動的に呼び出されるため、これらのメソッドを明示的にコーディングする必要はありません。

3.1.2 集合型

集合型は、基本スカラ型を組み合わせて構築される基本型です。ASCET では、以下の集合型を使用することができます。

- 配列 
- マトリックス 
- 特性カーブ 
- 特性マップ 
- ディストリビューション 

集合型は基本スカラ型で構成されます。配列とマトリックスは、4 種類のスカラ型のいずれかで構成することができ、特性カーブ、マップ、およびディストリビューションには、3 種類の算術型だけを使うことができます。基本スカラ型とは異なり、集合型は「参照型」として扱われます。参照型の変数を別の参照型の変数に代入すると、エレメントのコピー、つまり値の代入が行われるのではなく、エレメントのアドレスがコピーされます。

すべての参照型には、そのエレメント用のアクセスメソッドがあります。

- `set (reference type a)`：参照型 a にアドレスを代入します。この代入を行った後は、両方のエレメント（代入されたエレメントと代入したエレメント）は同じ 1 つのエレメントになります。
- `get`：集合型のエレメントのアドレスを返します。

メソッド呼び出しの際の引数渡しにおいては、代入の場合と同様に、アドレスがエレメントに渡されます。したがって、たとえば、メソッド内で引数に値を代入すると、その変更内容はメソッド外にも反映されます。このしくみは、C などのプログラミング言語の「参照呼び出し」に相当します。

配列

「配列」は基本型の 1 つで、同じ基本スカラ型（continuous や logical）の複数のスカラ値を含みます。配列内の各スカラ値の位置は、インデックス値により示されます。インデックス値のモデル型は unsigned discrete でなければなりません。配列のサイズは最大 2048 で、静的に定義する必要があります。配列のインデックスの値は 0 ～「配列のサイズー 1」です。

配列のインターフェースは、以下のメソッドで構成されます。

- `void setAt(scalar type a, udisc i)`: スカラ値 a を配列内の位置 i に代入します。
- `scalar type getAt(udisc i)`：配列の位置 i の値を戻します。

スカラ以外の基本型や複合（ユーザー定義）型の配列は、使用できません。

マトリックス

マトリックスは 2 次元の配列で、2 つのインデックスを持ちます。インデックスの型は配列の場合と同じで、udisc です。各次元のサイズは最大 63 です。つまり、各インデックスの値は 0 ～ 62 です。

マトリックスのインターフェースは、以下のメソッドで構成されます。

- `void setAt(scalar type a, udisc i, udisc j)`: スカラ値 a をマトリックス内の位置 (i,j) に代入します。
- `type getAt(udisc i, udisc j)`: マトリックスの位置 (i,j) の値を戻します。

スカラ以外の基本型や複合（ユーザー定義）型のマトリックスは、使用できません。

特性テーブル（カーブ／マップ）

ASCET では、非線形制御を実現するために、1 次元と 2 次元の「特性テーブル」を使用することができます。1 次元テーブルは「適合カーブ」と呼ばれ、2 次元テーブルは「適合マップ」と呼ばれます。ある 1 つの値が他の 1 つまたは 2 つの値に依存していて、その依存関係が明確でない場合、また、依存関係を関数で表した場合に非常に複雑な演算が必要となるような場合、そのような値は特性テーブルを使用して定義します。

特性テーブルの利用例として、たとえば、入力電圧に依存するダイオードのスループットの変化を挙げることができます。この特性動作はカーブにより記述されます。カーブはブレイクポイント（折れ線グラフの折れ点）の集合を表わす「テーブル」として表現され、各ブレイクポイントには出力のサンプル値が割り当てられます。1 つのブレイクポイントは、関数グラフの x 座標ポイントを表わす入力値（x 値）と、y 座標（ASCET では z 座標と呼ばれます）ポイントを表わす出力値（z 値）とで表わされます。

同様に、特性マップは、2 つの入力値（x 値と y 値）とそれに対応する出力値（z 値）として、2 次元テーブルにより表現されます。

特性テーブルのサイズ（ブレイクポイント数）は、特性カーブの場合は最大 2048 で、特性テーブルの場合は各軸については最大 63 です。特性テーブルはパラメータであるため、モデル内では読み取ることしかできません。

各特性テーブルには、補間ルーチンや補外ルーチンが用意されています。これらのルーチンは、特性カーブの入力値から出力値が求められる際に必要な補間係数を算出します。

ASCET には 2 種類の補間モードがあります。「丸め補間」では、2 つのブレイクポイント間の値は最も近いブレイクポイントのサンプル値が適用され、「線形補間」ではサンプル値の間を結ぶ直線上の値が適用されます。

コントローラ上で実行されるアプリケーションにおいて補間は非常に時間のかかる処理で、実際には 2 段階の処理に分けて行われます。1 つめはブレイクポイント間の正確なインターバルを求めて補間係数を算出する処理で、もう 1 つはその補間係数から出力値を算出する処理です。

「グループテーブル」と「固定テーブル」という 2 つの特殊な特性テーブルを使用することで、補間係数の計算を最適化することができます。グループテーブルはブレイクポイントのディストリビューション情報（x 座標値）を持たず、代わりに「ディストリビューションテーブル」を参照します。この「ディストリビューションテーブル」は、複数のグループテーブル間で共用できます。1 つのディストリビューションテーブルについての補間係数の計算は一度だけ行われ、その後、出力値の計算だけが各グループテーブルごとに個別に行われます。

ディストリビューションテーブル は、各ブレイクポイントの座標ポイントのみを定義する 1 次元テーブルです。したがって、2 次元のグループテーブルは 2 つのディストリビューションテーブルを参照します。

「固定テーブル」は等間隔ディストリビューション、つまり、ブレイクポイント間の間隔が一定のディストリビューションを持つテーブルです。これを利用すれば、補間係数の計算を大幅に高速化できます。ブレイクポイントのリストの代わりにオフセットとポイント間の間隔だけを持っているので、所要メモリも少なくて済みます。固定テーブルとグループテーブルを組み合わせて使うことはできません。

特性テーブルのインターフェースは、その次元と種類（通常、固定あるいはグループテーブル）により異なります。基本的には、以下のような 3 つのアクセスメソッドがあります。

- `void search(arithmetic type a)`：グループカーブによって参照されるディストリビューションテーブルに使用されます。このメソッドでは正確なサンプルポイントが求められ、補間係数が算出されます。2 次元テーブルの場合、パラメータは 2 つなので、`void search(arithmetic type a, arithmetic type b)` となります。
- `arithmetic type interpolate()`：特性カーブや特性マップ（主にグループカーブ／マップ）について、補間係数を使用して補間を行い、正確な出力値を求めます。
- `arithmetic type getAt(arithmetic type a)`： 補間係数の算出と補間の両方を行うメソッドです。2 次元テーブルの場合、パラメータは 2 つなので、`void getAt(arithmetic type a, arithmetic type b)` となります。

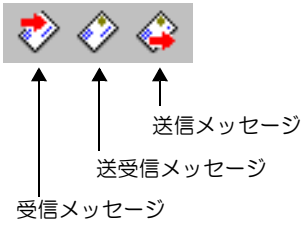
1 つのテーブルについての引数と出力値は、同じ算術型でなければなりません。たとえば、連続型と離散型を一緒に使用できる特性マップはありません。メソッド `getAt` の処理を `search` と `interpolate` の 2 つのメソッドに分けて実行する意味があるのは、グループテーブルの場合だけです。

ディストリビューションテーブルには `search` メソッドのみ、グループテーブルには `interpolate` メソッドのみ、そしてその他の一般的なテーブルや固定テーブルには、3 つのメソッドすべてが用意されています。

3.1.3 リアルタイム言語構造体

ASCET のコンポーネントを記述する際には、リアルタイムアプリケーション用のさまざまな言語構造体を使用できます。

メッセージ



メッセージは、プロセスの入力変数または出力変数として、基本スカラ型と同じ方法でプロセス間通信に使われます。メッセージは、グローバル変数とは異なり、プリエンプティブなスケジューリングにおいて「保護」されています。並行して実行される 2 つのプロセスの両方が同じメッセージにアクセスする場合でも、各プロセスはそれぞれ専用のコピーを使って処理を行うので、データの整合性が保証されます。メッセージはモジュール内でのみ使用できます。用途に応じて、以下の 3 種類のメッセージがあります。

- 受信メッセージ：読み取り専用で、モジュールへの入力として使われます。

- 送信メッセージ ：書き込み専用で、モジュールの計算結果の出力に使われます。

- 送受信メッセージ：読み取りも書き込みも可能です。

リソース

リソース（シンボル：  ）は、タイマや専用デバイスなど、排他的にしか使用できないシステムパーツを表します。リソースにアクセスするためには、以下の 2 つのメソッドを使用します。

- `void reserve()`：リソースを占有します。つまり、そのリソースに対する他のアクセスがブロックされます。
- `void release()`：リソースを解放します。つまり、そのリソースに対する他のアクセスが許可されます。

`reserve` メソッドを実行すると、そのリソースに対する他のアクセスが禁止され、プリエンプティブな環境における排他的アクセスが保証されます。つまり、あるリソースを占有しているプロセスがスケジューリングからはずされた時に別のプロセスがそのリソースを使用しようとした場合、そのアクセスは拒否されます。

リソースに対するアクセスが不要になったら、release メソッドを使用してそのリソースを解放します。これにより、他のコンポーネントがこのリソースにアクセスできるようになります。デッドロックや優先順位の逆転を防ぐために、リソースの占有はプロセスの優先順位を充分に考慮して行ってください。リソースは、グローバルエレメントです。

dT パラメータ

制御用アプリケーションの実行時において、コンポーネント内の計算結果がタスクの実行周期の違いにより異なってしまう場合があります。ASCET では、どのような周期で実行された場合にも 1 つのアルゴリズムで対応できるようにするために、システムパラメータ dT（シンボル： ）が提供されています。このパラメータの値はオペレーティングシステムによってセットされ、現在アクティブなタスクが前回実行開始された時からの経過時間を表します。

注記

dT という名前は、システムパラメータとして予約されたキーワードです。予約されたキーワードは大文字と小文字が区別されないため、DT、dt、Dt といった名前のエレメントを別に作成することはできません。

3.1.4 特殊な型

列挙型データ

列挙型データ（シンボル： ）は、列挙型として定義されている一連の列挙子の 1 つを値として持つ型です。

リテラル

リテラルは基本スカラ型の固定値を表す文字列で、どのような式にも使用することができます。リテラルの値は数値（離散あるいは連続）または論理値（true ／ false）のいずれかです。ブロックダイアグラムエディタでは、string、true、false、0.0、および 1.0 という値が用意されています。

3.2 エレメントの種類

すべてのエレメントには「種類」(“Kind”) が定義されています。これにより、そのエレメントが変数、パラメータ、システム定数、定数のうちのいずれのものとして使用されるかが決まります。

- 変数：モデル内で読み書きができます。つまり、変数に対しては読み取りと書き込みの処理を行うことができます。

  変数は、ECU においては、揮発性（Volatile）あるいは不揮発性（Non-Volatile）メモリに配置されます。新しく作成された変数については、デフォルトで Volatile 属性が割り当てられます。

- パラメータ：モデル内では読み取りのみが可能です。モデルの外側から書き込みを行い、「適合」することができますが、これには適合ツールが必要です。

  パラメータ（特性カーブ／マップを含む）は、自動的に Non-Volatile 属性が割り当てられ、ECU においては不揮発性メモリにに配置されます。

- 定数：モデル内で読み取りのみが可能です。パラメータとは対照的に定義された時点では固定され、モデルの外側からも変更することはできません。またコード実装時に変換は行われません。

  コード生成時には、定数は #define 文で定義されますが、生成されたコード内において必ずしも明示的に定義される必要はありません。たとえば再量子化を行う際に定数が使用される場合、その定数は明示的には定義されません。

- システム定数：定数と同様に扱われ、やはり #define 文で定義されます。ただし、定数と違ってシステム定数はコード実装時に変換が行われます。またシステム定数はコード内において常に明示的に定義されまます。

  システム定数は、コンポーネントマネージャのメニュー **Extras → Convert System Constants to Constants** で通常の定数に変更できます。

- インプリメンテーションキャスト（4.2.4 項を参照してください）：演算やデータフローの任意の個所にインプリメンテーション（実装情報）を定義するためのものです。変数やパラメータとは異なり、インプリメンテーションキャストはメモリを必要としません。そのためメモリ使用量には影響しませんが、適合を行うことはできません。

  インプリメンテーションキャストにはデータは含まれません。cont モデル型のスカラ値で、有効範囲は「ローカル」です（3.3 項を参照してください）。

表 3-1 は、各エレメントの使用法の違いをまとめたものです。

	モデル内	実験	インプリメンテーション
変数	r-w	r-w	使用可
パラメータ	r	r-w	使用可
システム定数	r	r	使用可
定数	r	r	使用不可
インプリメンテーションキャスト	—	—	使用可

表 3-1　変数、パラメータ、システム定数、定数、インプリメンテーションキャストの特徴

エレメントの種類は、ASCET ユーザーインターフェース（コンポーネントマネージャの“3 Contents”ペーンなど）において以下のようなシンボルで表わされます。

	スコープ				
	インポート	エクスポート	ローカル	依存	仮想
変数[a]					 [b]
メッセージ					
パラメータ[c]					*[d]
定数、システム定数					
インプリメンテーションキャスト					
dT					

a：配列、マトリックス、列挙型データを含みます。
b：スコープに依存しません。94 ページを参照してください。
c：特性カーブ／マップ、ディストリビューションを含みます。
d：他の設定（スコープなど）によってシンボルは異なります。

表 3-2　各種エレメント用シンボル

テンポラリ変数

1 つのメソッドまたはプロセス内で同じ計算が何度も実行されるのを防ぐために、演算子やメソッドコールごとに「テンポラリ変数」を定義することができます。テンポラリ変数が定義された式の値は、その式が使用されているメソッドまたはプロセスごとに一度だけ算出され、テンポラリ変数に格納されます。そして同じメソッド内でその式がもう一度使用されるときには、式は再計算されず、テンポラリ変数の値が再利用されます。

テンポラリ変数は各コンポーネントエディタで作成できます。テンポラリ変数には初期値はなく、その値は式の代入によって初めて決まります。ASCET はテンポラリ変数を（たとえば IF 文の分岐点で）内部的に管理して独自に代入を行い、後でその変数が使用されるときに値が未定義ということのないようにします。テンポラリ変数の値は、新しい値が代入されるまで有効です。

以下に、加算 a + b の値を格納して再利用できるようにするためのテンポラリ変数 t の使用例を示します。

```
t = a + b;
c = t;
d = t;
```

仮想変数／仮想パラメータ

仮想変数と仮想パラメータは、定義されたモデルエレメントの意味をわかりやすくするためにコンポーネントエディタ内でのみ使用されるもので、コードジェネレータにはまったく影響しません。

仮想変数は常に、他の仮想変数または非仮想変数に依存します。仮想変数は、単に非仮想変数の別名として使用され、算術的な依存は行えないため、仮想変数のデータ編集時には恒等式（var_virtual = var_real）がプリセットされます。

一方、仮想パラメータとして宣言されたパラメータは、必ずしも他のパラメータに依存するとは限りません。

依存パラメータ

モデルパラメータは、他のシステムパラメータまたはモデルパラメータに対して数学的な依存関係を持つ場合がありますが、このようなパラメータの一方を適合して値を変更すると、その依存関係に矛盾が生じてしまう場合があります。

このような不整合を防ぐため、コンポーネントエディタでパラメータの依存関係を記述しておくことができ、一方のパラメータを適合した際に他方のパラメータも自動的に再計算されるようにすることができます。パラメータの依存関係は数学の式で表現されます。

注記

「依存変数」というものは存在しません。

3.3 エレメントのスコープ

エレメントの中には複数のコンポーネント間のデータ交換に使われるものもあり、このようなエレメントは、1 つのコンポーネント（あるいはプロジェクト）からエクスポートされて、他のコンポーネントにインポートされます。この場合、エレメントの結び付けは名前により行われます。各エレメントのスコープには、以下の種類があります。

- Local（ローカル）エレメント：そのエレメントが定義されているコンポーネントの中でしか使用できません。つまり、そのコンポーネントに含まれるメソッドやプロセスだけが、このエレメントを使用することができます。
- Imported（インポート）エレメント：あるコンポーネントまたはプロジェクトで定義されたエレメントを、他のコンポーネントまたはプロジェクトで「インポートエレメント」としてインポートして使用することができます。このエレメントのプロパティは、そのエレメントを定義してエクスポートしているコンポーネント内でしか変更できません。
- Exported（エクスポート）エレメント：あるひとつのコンポーネントで定義されてエクスポートされる「エクスポートエレメント」は、他のすべてのコンポーネントでインポートしてアクセスすることができます。
- Method/Process-local（メソッド／プロセスローカル）エレメント：そのエレメントを定義しているメソッド／プロセス内でだけ使用できます。メソッド／プロセスローカルエレメントは静的エレメントではないため、データセットやインプリメンテーションを持ちません。

3.4 ユーザー定義されたモデル型

ユーザー定義のモデル型、つまり、モジュールやクラスも「エレメント」として使用できます。ユーザー定義のモデル型は常に「参照型」で、インターフェースは、コンポーネントのインターフェースにより定義されます。

ユーザー定義型エレメントのスコープは、基本型と同様に、インポート、エクスポート、ローカル、メソッドローカルのいずれかにすることができます。また、引数と同じように、参照型のメソッド／プロセスローカルエレメントはインスタンス化されませんが、それらへのポインタは確立されます。つまり、参照型のメソッド／プロセスローカルエレメントを使用する場合、このエレメントへの代入は、そのエレメントを使用する前に最初に行う必要があります。

エレメントの種類は、ユーザー定義のモデル型には適用されません。ユーザー定義のモデル型は常に変数として扱われ、モデル内でのインターフェースには何の制約もありません。

4 データとインプリメンテーション

前の章では、コンポーネントを構成するパーツを、一連のエレメント、コンポーネントのインターフェース、およびメソッド／プロセスに記述されたアルゴリズムとして定義しました。

本章では、コンポーネントで使用されるパーツをさらに 2 つ紹介します。それは、「データ」と「インプリメンテーション」（コード実装情報）です。データとインプリメンテーションはどちらもコンポーネント内のエレメントに属し、両方ともそのエレメントのプロパティを表わすものです。

データとインプリメンテーションを分けて記述する方法は、変数へのデータの代入処理がコード内に直接記述されるような一般的なプログラミング言語においては、通常、使用されません。

コンポーネントの「データ」は、そのコンポーネントのエレメントに初期設定される物理値を定義するものです。この「データ」には物理情報が含まれ、そのコンポーネントの物理記述の一部となります。

なお標準的なプログラミング言語では、通常、機能記述とそのインプリメンテーション（実装コード）は同一のものとなります。

4.1 データ

コンポーネントのデータは、そのコンポーネントのエレメントの初期値を定義するものです。

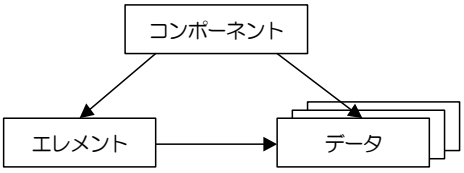


図 4-1 複数のデータセットを持つコンポーネント

データはエレメントとは別に保存されます。なぜなら、コンポーネントはプロジェクト内で複数のインスタンスを持つことができ、その場合、各インスタンスは、エレメント用にそれぞれ異なるデータを使用する可能性があるためです（ただし、データセットはインスタンスの一部分ではありません）。

一例として、比例制御フィルタを挙げてみます。この比例制御フィルタには複数のインスタンスがあり、それぞれ独自の比例ファクタ値が必要ですが、これは、各制御コンポーネントににそれぞれ異なるデータセットを割り当てることで実現されます。

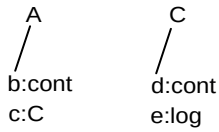
データの定義はそのコンポーネントの記述の一部であり、インスタンスの一部ではありません。これにより、1 つのコンポーネントについて多くのデータセットが存在することになる可能性はありますが、逆に各インスタンスが独自のデータを保持するようにしてしまうと、システムのモジュール構造が損なわれてしまいます。

各エレメント用のデータの構造は、そのエレメントが基本エレメントであるか複合エレメントであるかにより異なります。基本エレメントは常に複合オブジェクトの中で使用され、それらのオブジェクトと切り離して扱われることはないので、各基本エレメントは明示的なデータセットを持ちません。基本エレメント用のデータは、そのエレメントが含まれている複合エレメントのデータセットの一部として定義されます。

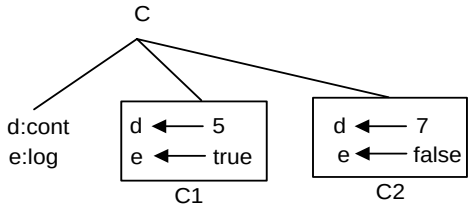
複合エレメントは、ユーザーが定義するコンポーネントで、個々の複合エレメントはそれぞれ個別のデータセットを持ちます。複合エレメントがコンポーネント内で使用される際、そのエレメントのデータセットがコンポーネントにより参照されます。したがって、コンポーネント内のデータは、そのコンポーネント自体と同じ階層構造を持つことになります。

データセットにはオブジェクト ID があります。コンポーネントのデータを参照するときにはこのオブジェクト ID が使用されます。ユーザー定義型への参照と同様に、名前は使用されません。

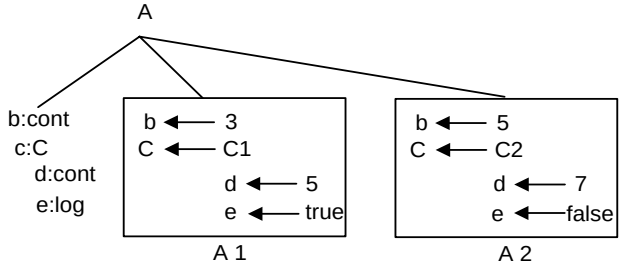
型 A と C についての以下の例について考えてみます。



ここで、型 C には以下のデータセットが定義されています。



すると、C のデータセットを使用する型 A のデータ宣言は、以下のような結果になります。



基本型のデータは直接定義することができます。スカラ型の場合はデータは 1 つの値で構成され、配列や特性カーブのような集合型の場合、データは値のテーブル、または 入力値と出力値で表わされるブレイクポイントのテーブルで構成されます。

4.2 インプリメンテーション（コード実装情報）

「インプリメンテーション」は、コンポーネントのエレメントがコード生成時にどのようなデータとして実装されるかを定義するものです。インプリメンテーションについても、データの場合と同じ図式が適用されます。

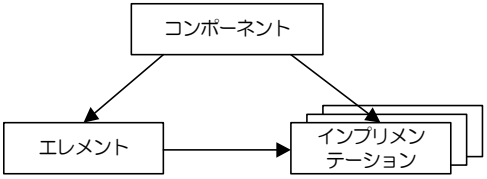


図 4-2 複数のインプリメンテーションを持つコンポーネント

インプリメンテーションは、基本型の場合も複合型の場合も、同じように参照されます。インプリメンテーションの影響は、データセットの影響よりもはるかに広範に及びます。エレメントのインプリメンテーション、たとえば、型 `cont` のエレメントがデータ型 `float` と `signed int` のどちらで表現されるか、という違いは、メソッドやプロセスの機能記述から生成されるコードの内容に直接影響します。

4.2.1 スカラ型のインプリメンテーション

インプリメンテーションは、コード生成時において、基本型のエレメントがどのような C 言語の型に実装されるかを記述したものです。論理エレメント（logical 型）は true と false という 2 種類の値しかとらないため、このエレメントのインプリメンテーションは非常に単純で、インプリメンテーションはデータ型の指定だけで構成されます。論理エレメントの場合は、`byte`、`word`、あるいは `long` を選択することができます。

算術型の場合、インプリメンテーションの内容ははるかに複雑です。特に、データ型に関しては、たとえば `continuous` 型のエレメントを整数型に実装することも可能なため、物理定義から実装コードへの複雑な変換式が必要になります。

物理定義と実装コードでは、その内容な大きく異なります。たとえば、モデル型が `continuous` であるエレメントの場合、物理定義において無限の範囲の値（- ∞ ～ + ∞）を任意の量子化（分解能）で使用することが可能です。一方、実装コードでは、値の範囲はターゲットコントローラのワード長により限定され、量子化は 1（ビット単位）に固定されます。

このように内容がまったく異なる物理定義と実装コードの間の変換を可能にするためには、まず物理定義の範囲を限定する必要があります。この範囲は、「インターバル」として各エレメントに割り当てます。また分解度も制限する必要があるため、各エレメントに固定の解像度を設定する、つまり 1 ビットあたりの量を指定する（「量子化を行う」）必要があります。

たとえば、A を物理定義の一連の値、A = [-1,0.5] とし、q = 0.2 の量子化を想定します。

値の範囲を限定し、さらに量子化を行うと、エレメントの値は等間隔な有限個の値の集合になります。

Aq = {-1, -0.8, -0.6, -0.4, -0.2, 0, 0.2, 0.4}

この有限集合は、整数の範囲にマッピングすることができます。

Aint = {-5, -4, -3, -2, -1, 0, 1, 2}

これは、物理定義から実装コードへの一次変換式 `impl = 5 * phys` によって実現されます。整数変数のデータ型は、必要な値の範囲によって自動的に決まります。この例では、データ型 `signed int8` が選ばれます。

物理定義においてゼロより大きいオフセットがある場合には、実際の値が大きくても、実装コードにおいて小さなデータ型を使用できる場合があります。

たとえば、値の範囲が A = [120,130] で量子化が q = 0.5 の物理定義について考えてみましょう。一次変換の結果は、*Aint* = {240, ..., 260} という整数の範囲になります。

この場合、整数変数の型は `unsigned int16` になりますが、この個数の値なら型 `int8` の変数に収めることも可能です。

これを実現するためには、オフセット付きの一般的な一次変換式を指定することができます。上の例では、以下のような変換式を使えば、{0, ..., 20} の整数インターバルになるので、データ型 `unsigned int8` の変数で十分となります。

```
impl = 2 * phys - 240
```

変換式は、コンポーネントではなくプロジェクトレベルで定義されます。これにより、複数のコンポーネントで同じ変換式を使用することができます。これは ASAM-MCD-2MC 標準にも準拠しています。

4.2.2 集合型のインプリメンテーション

配列、マトリックスあるいは特性テーブルのような集合型の場合、インプリメンテーションはその集合型のインターフェースエレメント（これ自体はスカラ型です）について定義されます。

たとえば配列の場合、配列内のエレメントのインプリメンテーションを定義します。定義されたインプリメンテーションは、配列の入力と出力の両方に有効です。インデックスは離散モデル型なので、そのインプリメンテーションは固定的です。

特性テーブルの場合、テーブルの x/y 座標ポイント（x 値と y 値）のインプリメンテーションと出力値（z 値）のインプリメンテーションとを個別に指定することができます。

4.2.3 ユーザー定義型のインプリメンテーション

ユーザー定義型のインプリメンテーションは、そのコンポーネント内で使用されているすべてのエレメントのインプリメンテーションで構成されます。

クラスの場合、実引数（呼び出し側）と仮引数（関数側）とを適切に相互調整する必要があるので、引数と戻り値にもインプリメンテーションが必要です。この調整は、スカラ型の引数については自動的に行われます。

この自動調整は、集合型や複合型の引数については行われないため、そのような引数が使用される場合、仮引数と実引数のインプリメンテーションが一致していなければなりません。これらの引数は参照渡しで渡されるので、自動調整は不可能です。

テンポラリエレメントは明示的なインプリメンテーションを持ちませんが、コード生成アルゴリズムによって自動的にインプリメンテーションが割り当てられます。この変数を使用する前には、初期処理等において必ず値を代入しておく必要があります。

メソッドまたはプロセス固有のエレメントもテンポラリエレメントと同様に自動的にインプリメントされますが、これらは明示的にインプリメントすることも可能です（『ASCET ユーザーズガイド』の「メソッドローカル／プロセスローカル変数のインプリメンテーション」を参照してください）。インプリメンテーションはメソッド／プロセス内に保存されます。

4.2.4 インプリメンテーションキャスト

ASCET V5.0 から、新しい基本エレメントとして「インプリメンテーションキャスト」が導入されました。これは、一連の演算フロー内の中間値のインプリメンテーションを調整するためのもので、これによって、モデル内の物理的な相互作用（たとえば、あるモデルの一連の処理内のあるポイントにおいて、モデルの値が許容範囲を超えていないか）を、余分な物理メモリを使用することなくチェックすることができます。

注記

インプリメンテーションキャストは、論理エレメント用には使用できません。

以下に、簡単な例をあげてこの機能を説明します。

以下のダイアグラムでは、2 つの変数（a と b）の加算が行われてからその結果にリテラルの 2 が掛けられ、その結果が変数 c に格納されています。

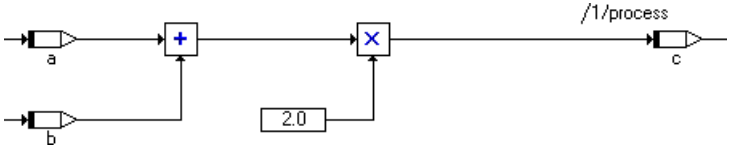


図 4-3 インプリメンテーションキャストを含まない演算処理の例

実装時には、変数 a、b、c は int16 型に割り当てられ、これら 3 つの変数はこの型の範囲をフルに使用します。そのため、コードジェネレータは 32 ビット幅のテンポラリ変数を作成して演算を行い、その値を c に代入する前には、int16 に適応するために右シフトによる再量子化を行います。

ここでもし、物理的な制約条件や、モデル内の相関関係などにより、a と b を加算した値が 16 ビット幅を超えることがなく、しかもその半分の範囲しか使用されない、ということが明らかである場合、図 4-4 に示されるように、インプリメンテーションキャストを使用することができます。

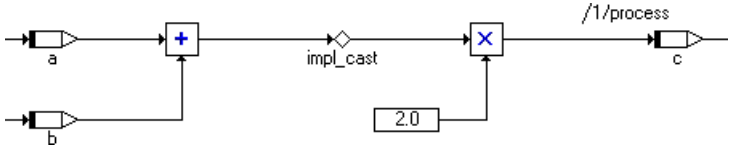


図 4-4　インプリメンテーションキャストを付加した演算処理の例

上図の位置に int16 型で値の範囲が [-16384..16383] であるインプリメンテーションキャスト（**Limit to maximum bit length** および **Limit Assignments** オプションはオフにします）を加えてコードジェネレータが生成する中間結果のプロパティを明確に規定することにより、図 4-3 の場合に必要であった再量子化が行われないようにすることができます。

また、インプリメンテーションキャストの別の応用例として、演算子の入力と出力にターゲットコントローラに応じたインプリメンテーションを与えることができます。この機能を利用して、演算子ごとに算術演算サービス（『ASCET ユーザーズガイド』の「算術演算サービス」の章を参照してください）を選択することができます。この場合、インプリメンテーションキャストは、既存の演算子インプリメンテーションに相当します。

インプリメンテーションキャストは、その名前のとおり、インプリメンテーションに対してのみ影響します。つまり、インプリメンテーションキャストは、インプリメンテーションが使用される以下のタイプの実験用のコード生成を行う際にのみ考慮されます（『ASCET ユーザーズガイド』の「プロジェクトの設定」についての記述を参照してください）。

- Implementation Experiment
- Object Based Controller Implementation

以下のタイプの実験を行う場合は、インプリメンテーションキャストは無視されます。

- Physical Experiment
- Quantized Physical Experiment

実装実験（Implementation Experiment）用のコード生成オプションに応じて（『ASCET ユーザーズガイド』の「プロジェクトの設定プロジェクトのコード生成オプションを設定する：」の項を参照してください）、インプリメンテーションキャストは以下のように扱われます。

- プロジェクト内に定義されている最大のビット幅が 32 ビットよりも小さい場合、インプリメンテーションキャストのコード生成時には、それより大きなサイズを使用することができます。しかし、コード生成時において、もしも許可されたビット幅を超える変数が必要となった場合は、エラーメッセージが出力されます。

  この機能により、インプリメンテーションキャストを一連の算術演算シーケンスの中で使用して、コントローラの最大ビット幅を超える中間結果を使用することが可能となります。

- 除算の分子への入力の部分にインプリメンテーションキャストが付加されていた場合、そのインプリメンテーションの内容が **Allow Double Bit Size for Division Numerators** オプションの設定内容に優先します。

インプリメンテーションキャストのもう 1 つの重要な特性としては、コード生成時においてメモリが（パーマネントメモリも一時メモリも）割り当てられない、という点があげられます。これは、インプリメンテーションキャストはグローバル変数やローカル変数として生成されることはないためです。ただし、演算結果の範囲チェックが有効になっているときにインプリメンテーションキャストを使用する場合は、範囲チェックが行われる前に、ローカルなテンポラリ変数に演算結果を一時的に格納することが必要となる場合もあります。

インプリメンテーションキャストを使用できるのは、ブロックダイアグラムと ESDL エディタのみで、一般的には、モジュールとクラス（ただし CT ブロック、論理テーブル、条件テーブルを含まないもの）、およびステートマシンでのコンディションとアクションの記述に使用します。

4.3 インプリメンテーションを使用したコード生成

コード生成オプションとしてインプリメンテーション（`Implementation Experiment`）を選択すると、コードは固定小数点演算で生成されます。この固定小数点演算は整数演算を使用して行われます。インプリメンテーションの情報は、コンポーネント内の各エレメントに適用され、この情報と機能記述から整数コードが生成されます。

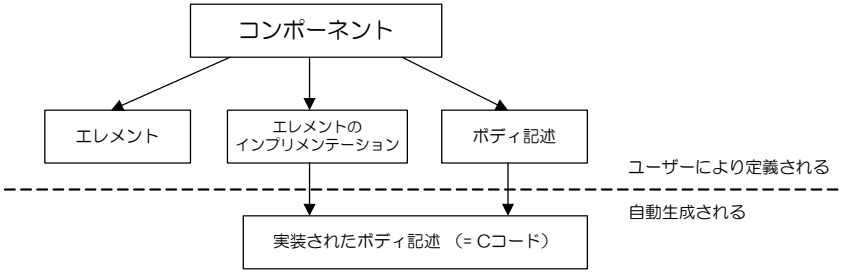


図 4-5 インプリメンテーションを使用したコード生成

整数コード生成の原理をわかりやすくするために、以下に単純な例を紹介します。

例：加算のコード生成

以下の単純な例について考えてみます

```
c = a + b;
```

ここで、a、b、および c は continuous 型のモデル変数です。

これらを実装する際の変換は、オフセットなしの一次変換です。a には 0.01、b には 0.04、c には 0.05 の量子化を使用します。A、B、および C は、生成される C コードにおける、これらのエレメントに対応する「実装変数」です。

上記の例についてコードを生成する場合、量子化を考慮する必要があります。a = 1、b = 0.6、およびそれに基づく c = 1.6 という値に対して前述の量子化を行うと、結果は A = 100、B = 15 および C = 32 になります。このモデルを実装レベルに直接変換しても、正しい結果にはなりません（A + B = 100 + 15 = 115 で、C = 32 と等しくなりません）。

この理由は、量子化が考慮されていないことにあるので、前述のモデル方程式を「インプリメンテーション変換」、つまりインプリメンテーションに定義された変換に変える必要があります。この例では、A と B の量子化を調整してから加算を行い、さらにこの加算の結果を C の量子化に合わせて調整する必要があります。このようにすると、前述のモデルについての C コードは以下のようになります。

```
C = (A + 4 * B) / 5:
```

B に 4 を掛けることにより量子化 0.04 が 0.01 に調整され、5 で割ることにより、量子化 0.01 が 0.05 に調整されます。

4.3.1 インプリメンテーションを使用したデータの実装変換

エレメントと共に格納されるデータには、必ず「モデルデータ」、つまり、物理値が定義されますが、インプリメンテーションはこのデータにも反映されなければなりません。上の例では、変数 a の物理（モデル）データは 1 でしたが、実装された変数 A のデータは 100 でした。

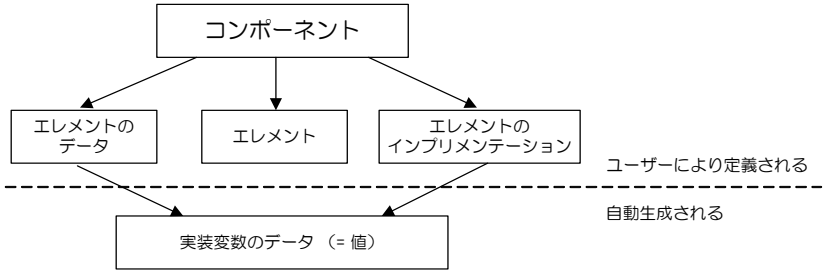


図 4-6　データの変換

4.3.2 インプリメンテーションによる実装変換についての一般的ルール

実装変換は算術値に対して行われます。正しい算術演算が実行されるように、すべての算術式で値が調整されます。

加算と減算 :

これらの演算の引数は量子化に合わせて調整されます。この量子化は、内部コード生成アルゴリズムにより、再量子化を行う回数が最小限になるように決定されます。引数の量子化とオフセットから、結果についての定数オフセットが算出されます。

乗算と除算 :

これらの演算の引数は、その乗算あるいは除算が行われる前に、最初にオフセットなしで作られます。量子化を適応させることはできず、その乗算あるいは除算の結果により決められます。また、あふれや精度の損失を避けるために、引数の量子化に 2 の累乗がかけられる場合があります（シフト演算）。これも、内部コード生成アルゴリズムにより自動的に決まります。

比較、最小および最大 :

加算と同様に、引数どうしが調整されます（量子化についてもオフセットについても）。最小および最大の演算子も、加算の演算子と同様に機能します。

代入 :

変数に値が代入される場合、代入が行われる前に代入する値が再量子化され、オフセットが調整されます。これは、引数渡しの場合も同様です。

4.4 メソッドとプロセスのインプリメンテーション

ASCET 5.0 においてインプリメンテーションの使用範囲が拡張され、メソッドのインプリメンテーションも定義できるようになりました。メソッドとプロセスのインプリメンテーションは、ESDL とブロックダイアグラムの両方の形式で定義できます。

メソッドとプロセスのインプリメンテーションには、メソッドまたはプロセスを実行するために使用されるメモリについての情報や、コード生成時にインスタンスのコードを展開するかどうか、といった情報が含まれます。

一般に、早いレスポンスが要求されるアルゴリズムや非常に頻繁に使用されるアルゴリズムは内部メモリで実行し、あまり頻繁に使用されないアルゴリズム（初期化アルゴリズムなど）は外部メモリで実行します。

さらに、メソッドやプロセスの呼び出しを、関数呼び出しにしたり、インラインコードに展開することもできます。

5 ESDL によるボディ記述

本章では、クラスとモジュールの記述に使用される「ESDL」の一般的な機能について説明します。本章の記述は以下のように大きく 3 つに分かれています。

最初の部分では、ESDL の一般的な特徴について簡単に説明します。それに続く項では、ESDL の構文とエレメントについて総合的に説明します。

本章の最後には、ESDL とブロックダイアグラムの違い、および ESDL と C や Java のプログラミング言語の違いについてまとめてあります。

この章を読む際は、C または Java のプログラミング言語に習熟していることが前提条件となります。C や Java についての詳細は、各言語の一般的なマニュアルをご覧ください。

以下は、Java と C の一般的なリファレンスマニュアルの一覧です。

- Arnold, Ken, Gosling, James,『The Java Programming Language』(Reading, Mass.: Addison Wesley, 1996)
- Flanagan, David,『Java in a Nutshell』(Cambridge, Mass.:O'Reilly,^{2}1997).
- Kernighan, Brian W., Ritchie, Dennis M.,『The C Programming Language』(Englewood Cliffs: Prentice-Hall, ^{2}1988).

5.1 モデリング言語としての ESDL

ESDL は、自動車関連のアプリケーションに特化して設計されたモデリング言語です。ASCET では、この言語を使用してクラスやモジュールの中のメソッドやプロセスのボディを記述することができます。この項では、記述内容を単純化するため、「クラスとモジュール」を「クラス」という語で一括して表記します。

短期間で習熟できるように、ESDL の構文とエレメントの外見は Java プログラミング言語に似ていますが、機能的には、そういったプログラミング言語とはまったく異なるものです。

ESDL の主な特徴は以下のとおりです。これらの特性の一部は、他の言語とは明らかに異なります。

- ESDL は、プログラミング言語ではなくモデリング言語です。ASCET でよく使われるブロックダイアグラムと同じく、抽象レベル、つまり物理レベルの記述に使用されます。ポインタやシフト演算子、といった実装コードに依存する概念はありません。
- ESDL はリアルタイム環境で稼働するシステムに使用されるため、リアルタイム動作の要件を満たしている必要があります。したがって ESDL の「オブジェクト指向」は、限られた範囲内で実現されています。モデル構造体をクラスやモジュールにマップすることはできますが、インスタンス生成は静的に行われ、継承はありません。
- ESDL は自動車関連のソフトウェア を対象に設計されているため、複雑なソフトウェアモデルを構築することはできますが、文字列操作など、組み込みシステムにはあまり重要でない機能はサポートされていません。

- ESDL は ASCET の開発環境にスムーズに組み込むことができます。この言語はブロックダイアグラムと同じレベルで、メソッドまたはプロセスのボディ に含まれる関数の記述に使われます。ESDL エディタには、エレメントをインポートしたり変数を宣言したりする機能が用意されています。

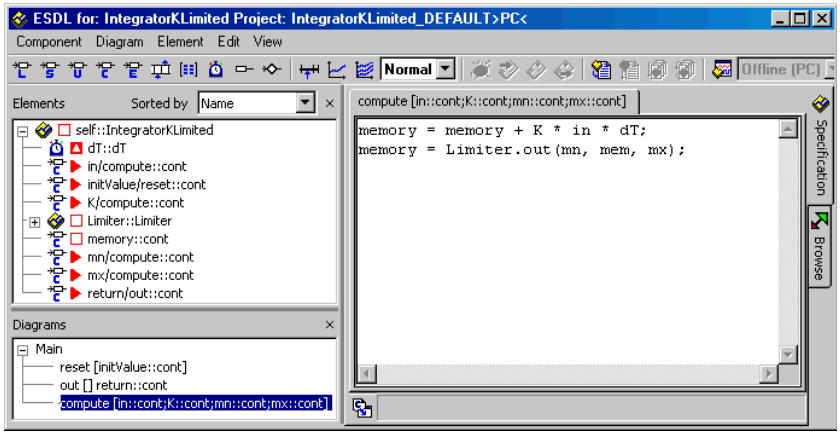
以上 4 つの特徴により、ESDL の機能範囲と用途が明確になります。ESDL は Java プログラミング言語を特化したものではありませんので、ご注意ください。

5.2 基本エレメント

5.2.1 メソッドとプロセス

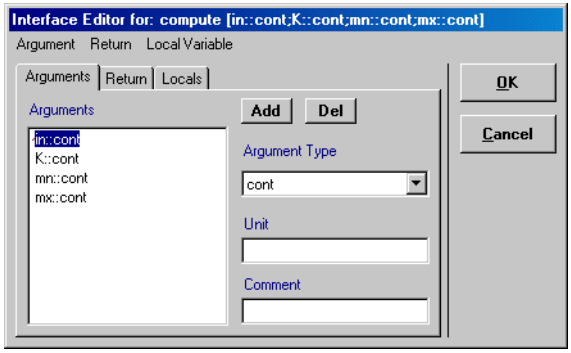
ESDL におけるファンクション記述の基本となるエレメントは、メソッドとプロセスです。メソッドは、変数等の定義部分であるメソッドヘッダと、実行される処理について記述するメソッドボディで構成されます。

メソッドヘッダはメソッド名、引数のリストおよび戻り値で構成されます。メソッド名は、新しいアイテムを ESDL エディタのメソッドリスト（“Diagrams” ペーン）に追加した時に割り当てられます。リストのアイテム名を変更すれば、メソッド名を変更することができます。



ESDL では、メソッド名はユニークでなければなりません。メソッドの 多重定義はサポートされていません。つまり、パラメータ数やパラメータ型が異なる 2 つのメソッドを同じ名前で定義することはできません。

引数と戻り値は、メソッドインターフェースとして任意に作成できるエレメントです。メソッドのヘッダとインターフェースの変更は、ESDL エディタウィンドウのインターフェースエディタを使用して行います。インターフェースエディタは、引数や戻り値を必要に応じて追加したり変更したりするために使用します。



モデルの機能記述はメソッドボディに含まれ、ESDL エディタのテキストペーンで編集されます。

5.2.2 ESDL の構文

ESDL の構文は Java プログラミング言語の構文と同じで、文の終わりにセミコロン（;）を付けます。

```
Timer.calculate();
x = a + b;
tmp = Timer.out();
```

複合文つまりブロックは、中かっこ { ... } で囲みます。

```
if (x > 0) {
   y = f(x);
   z =1; }
```

式、およびメソッドの引数は、小かっこ (...) で囲みます。

```
while (z > 4) {
   z--;}

Integrator.reset(15);
Limiter.out(0, 15, 100);
```

代入には等号（=）を使用します。

```
low = -1;
xVar = a * (b-5);
tmp = xVar.max(15);
```

5.2.3 変数名

ESDL では、変数名は文字と数字で構成されます。変数名の 1 字目は文字でなければなりません。アンダースコアも 1 文字として扱われます。変数名に空白を含むことはできません。

有効な ESDL の変数名の例を以下に示します。

```
i, j2a, aVar, a_Var
```

すべての変数名は現在のエレメントのスコープ内でユニークでなければなりません。この制約は、インポートされたクラスやモジュールを使用して作業する場合に重要です。この段階では、ESDL は名前の衝突を解決しません。

予約されているキーワード :

以下のキーワードは予約されているので、変数名として使用することはできません。

```
auto, break, case, char, cond, const, continue, default,
df, do, double, dt, else, enum, exit, extern, false,
float, for, get, getat, getatat, goto, header, if,
inactive, int, interpolate, long, monitorprocess,
normal, null, receive, register, return, search, self,
send, set, setat, setatat, short, signed, sizeof, static,
struct, switch, true, typedef, undef, union, unsigned,
void, volatile, while
```

キーワードについては大文字と小文字が区別されないので、上記の名前は、大文字／小文字が異なっていても使用できません。

5.2.4 データ型

ESDL ではデータ型が厳密に規定されていて、変数の宣言が必要です。この場合の手順はブロックダイアグラムを編集する場合と同じです。変数はエレメントリストに追加されるので、必要に応じて編集できます。

ESDL では、`udisc`、`sdisc`、`cont` および `log` の 4 つのデータ型を使用できます。エディタのツールバーから対応するエレメントを選択して、これらの型をクラスやモジュールに追加することができます。

ESDL のメソッドまたはプロセスのボディ自体には、変数の宣言は含まれません。変数が現在のメソッド／プロセスのローカル変数である場合だけは、以下のような文を使用してメソッドボディ内で宣言し、初期化することができます。

```
cont set = 12.34;
cont temp = 0.78e4;
udisc i = 3, j, k;
sdisc aVar = -12;
log trigger = true;
```

5.2.5 型変換

+、-、*、/ などの基本算術演算子に、互いに異なる型のオペランドを与えると、その計算結果はその式に使われている中で最も強い型に自動的に変換されます。

型の強さの順序は、（弱い方から順に）sdisc、udisc、cont です。

```
cont result = varUdisc + varCont;
```

ある値を変数に代入するときには、値と変数のデータ型が一致していなければなりません。明示的な型変換はありません。ESDL は符号付き離散、符号なし離散、および連続の基本算術型についてだけ、暗黙の変換を行います。

```
cont tmp = 2;
```

論理型から算術型へ、またはその反対の変換はできません。

5.2.6 基本メソッド

すべての算術型についてはあらかじめインターフェースが定義されていて、一連の基本数学関数を利用できます。各算術型ごとに、以下のメソッドを使用することができます。

メソッド	受信側	戻り値	用途
`val.abs()`	算術	算術	`val` の絶対値を取得
`val1.max(val2)`	算術	算術	2 つの値のうち大きい方を取得
`val1.min(val2)`	算術	算術	2 つの値のうち小さい方を取得
`var.between(val1, val2)`	算術	論理	`var` の値が `val1` と `val2` の間であるかを評価

表 5-1　算術型の基本メソッド

`var.between(val1, val2)` メソッドは、ブロックダイアグラムの `between:And:` エレメントに相当します。

5.2.7 リテラルと定数

リテラルは、`12`、`6.1e4`、`true` などの値です。どの基本型（論理型および算術型）のリテラルも、ESDL メソッド内に生成することができます。リテラルのデータ型は暗黙的に決まります。

定数は `g = 9.81` のような名前付きの値です。定数はクラスに追加され、変数と同じ方法で宣言されます。エレメントエディタを使用して、値を割り当て、変数を定数として定義します。

以下にその例をあげます。

- `x = g.abs();`

  定数 g の絶対値を、変数 x に代入します。

- `out1 = myvar.max(g);` あるいは `out1 = g.max(myvar);`

  `myvar` と g の値を比較し、大きい方の値を変数 `out1` に代入します。

- `out2 = myvar.min(.04);` あるいは `out2 = (.04).min(myvar);`

  `myvar`（変数）の値と 0.04 とを比較して、小さい方の値を変数 `out2` に代入します。

5.2.8 コメント

コメントは、ESDL コードの内容を説明するものです。コメントには単一行コメントと複数行コメントの 2 種類があります。

単一行コメントは、ダブルスラッシュ（`//`）で始めます。その後ろから行末までのテキストは無視されます。複数行コメントは `/*` と `*/` で囲みます。

ESDL で記述されたコメントは、その ESDL コードから生成される C コードには反映されません。

5.2.9 演算子

ESDL では、メソッド呼び出しが他のどの演算子よりも優先されます。式にかっこを追加すると、優先順位を操作することができます。

単項演算子 :

単項演算子は +、- および !（NOT）です。! は論理型に使用されます。その他、インクリメント演算子（++）およびデクリメント演算子（--）を使用することができます。これらはプレフィックス演算子あるいはポストフィックス演算子として使用できます。

単項演算子は、他のどの演算子よりも優先されます。これらの演算子は、右から左の順序で結合されます。

算術演算子 :

ESDL では +、-、* および / の 4 つの算術演算子を使用できます。整数の除算の剰余を求めるモジュロ演算子 % も使用できます。

*、/ および % 演算子は、2 項演算子の + や - より優先されます。算術演算子は左から右の順序で結合されます。

比較演算子と等号演算子 :

比較演算子は >、>=、< および <= です。これらは算術型に適用され、等号演算子より優先されます。

等号演算子の == と != は値型と参照型の両方に適用でき、関係演算子の次に優先されます。

関係演算子と等号演算子は 2 項演算子です。左から右の順序で結合されます。

論理演算子 :

次に優先されるのは論理演算子の && と ||（AND と OR）です。AND の方が OR よりも優先されます。

論理式は、式全体の真偽が決まるまでの間しか評価されません。たとえば、式 `a && b` を評価する場合、a が `false` なら、この式の残りの部分を評価する必要はありません。b の評価は式全体の真偽に何の影響も与えません。

論理演算子は 2 項演算子です。左から右の順序で結合されます。

条件演算子（MUX）：

条件演算子 ?: はブロックダイアグラムエディタの MUX 演算子に相当します。この演算子の一般形は (a ? n : m) で、a は論理型でなければならず、n と m は同じ型でなければなりません。n と m は論理型や算術型のどの基本型でもかまいません。

条件式の値は、a の値により決まります。上の例では、a が true なら式の値は n になり、false なら m になります

条件演算子には 3 つの成分があります。この演算子の優先順位は、すべての 2 項演算子の次です。右から左の順序で結合されます。

複合代入演算子：

ESDL では、+= や *= などの一般的な複合代入演算子を使用できます。a += 4 という演算は、a = a + 4 という代入演算の省略表記です。以下の演算子の省略表記が可能です。

```
*=, /=, %=, +=, -=
```

省略表記の演算子の優先順位は最低です。これらの演算子は、右から左の順序で結合されます。

まとめ：演算子の優先順位と結合規則：

下の表は、前の項で説明した ESDL の演算子の優先順位と結合規則をまとめたものです。

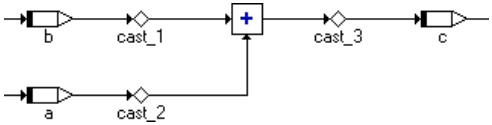
演算子	結合規則
++ --	右から左
+ -（単項）	右から左
!	右から左
* / %	左から右
+ -（2 項）	左から右
< <=	左から右
> >=	左から右
==	左から右
!=	左から右
&&	左から右
||	左から右
?:	右から左
=	右から左
*= /= %= += -=	右から左

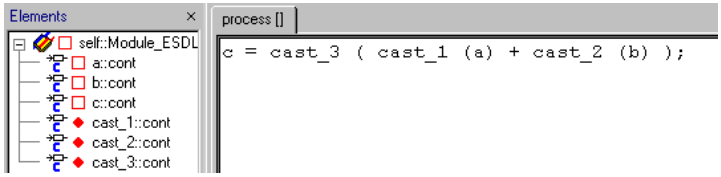
表 5-2　演算子の優先順位と結合規則

5.3 ESDL でのインプリメンテーションキャストの使用

インプリメンテーションキャスト（4.2.4 項を参照してください）は、ESDL で記述されるモジュールとクラス（CT ブロック以外）でも使用することができます。

ESDL で記述される演算において、インプリメンテーションキャストはその名前で表わされます。以下の例は、同じ加算処理をブロックダイアグラムと ESDL で記述したものです。





インプリメンテーションキャストはメソッド呼び出しのような形式で記述します。つまり、インプリメンテーションキャストを関数名とすると、型変換されるエレメントは括弧 ( ) で囲んで引数のように記述します。またインプリメンテーションキャストを演算の結果に使用する場合は、演算の全体を括弧 ( ) で囲み、その前にインプリメンテーションキャストの名前を記述します。

上の例では、`cast_1` が変数 a を型変換し、`cast_2` が b、`cast_3` が演算式 `a + b` の結果を型変換します。

算術演算の中間結果にインプリメンテーションキャストを使用する場合は、その中間結果を算出する部分を括弧 ( ) で囲みます。

つまり、

```
x = cast_1 ( (cast_2 ( (a + b) * c - d ) ) / e );
```

この式において、`cast_2` は `(a+b)*c-d` という演算部分を型変換し、`cast_1` が演算 `((a+b)*c-d))/e` 全体の結果を型変換します。

注記

ESDL においてインプリメンテーションキャストは、必ずその直後に続くコードの値の型を変換します。

なお上記の構文は、インプリメンテーションキャストにのみ使用できます。インプリメンテーションキャストの代わりに通常の型を記述すると（例、`uint8 (a)`）、エラーメッセージが出力されます。

インプリメンテーションキャストは、論理値には使用できません。インプリメンテーションを論理値に使用すると、コード生成時にエラーが発生します。

5.4 処理フロー制御

処理フロー制御エレメントは、ESDL の関数や文を実行する順序と条件を決定するために使用されます。最も一般的なものは、条件文とループ文です。

条件文には `if...else` と `switch...case...default` の 2 種類があり、ループ文にも `while` と `for` の 2 種類があります。

また `break` 文も使用できます。

以下に、ESDL での処理フロー制御の構築について詳しく説明します。

5.4.1 If...Else

`if...else` 文を使えば、単純な条件付き処理フローを構築できます。一般形は次のとおりです。

```
if (expressionLog){
   statementTrue;}
else {
   statementFalse;}
```

`else` ブロックは省略可能です。`expressionLog` が評価されると、プログラムは `statementTrue` ブロックを実行するかどうかを決定します。実行しない場合には、プログラムは `statementFalse` ブロックがあればそれを実行し、なければ何も実行しないで次に進みます。

評価に使用される `expressionLog` は、明示的に `log` 型でなければなりません。算術型の 1 や 0 の値は認められません。

常に評価結果が `true` となる `if` 文があると、そのような処理フロー制御部分は、C コード生成時において最適化されます。

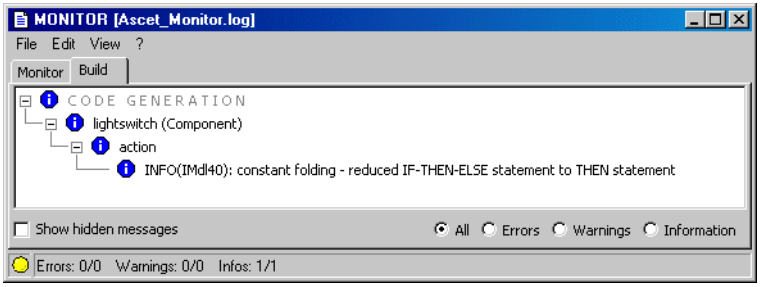
たとえば、

```
if (true || testlog_a) {
   cont=1; }
else {
   cont=0; }
```

という記述は、最適化によって以下のように短くなります。

```
cont=1;
```

最適化が行われると、ASCET モニタウィンドウに情報メッセージが表示されます。



生成された C コード内にはこれらの情報は出力されません。

注記

最適化を行うかどうかは、それぞれの `if` 文ごとにローカルに判定されます。`if` 文の判定結果が、その前のプログラム部分の処理結果によって左右される場合、最適化は行われません。

5.4.2 Switch...Case...Default

`switch...case...default` 文（`switch` 文）を使えば、さらに複雑な条件付き構造を構築できます。一般形は次のとおりです。

```
switch (expressionsDisc) {
   case sDiscM: {
      statementM }
   …
   case sDiscN: {
      statementN }
   default: {
      statementDefault }
}
```

`switch` 文は、引数 `expressionsDisc` が `sDiskM` から `sDiscN` の定数値の 1 つと等しいかどうかを判断し、その結果に応じて分岐します。

各ケースには、必ず `sdisc` 型の定数式のラベルが付きます。`expressionsDisc` がいずれかの定数式の値と一致すると、それに対応するブロックが実行されます。`default` ケース（オプション）のブロックは、`expressionsDisc` の値が他のどのラベルの定数式とも一致しない場合に実行されます。

`default` ケースの指定がなく、`expressionsDisc` の値が他のどのラベルの定数式とも一致しない場合には、`switch` 文は何も行わず、制御はそのソフトウェアモデルの残りの部分に戻ります。

下の例では、argSdisc の値に応じて、変数 scont に値を設定します。

```
switch(sdiscArg) {
   case 1 : {
      scont = 1.123;
      break; }
   case -1: {
      scont = 0;
      break; }
   default: {
      scont = -1; }
}
```

この例では、各ブロックが break 文で終わっています。これにより、この switch 文は、1 つのブロックの実行が終わるとすぐに終了します。

case ブロックが明示的に終了されていない（つまり break 文が挿入されていない）場合は、一致が見つかった直後の処理から switch 文の最後までが続けて実行されます。つまり、上の例から break 文を省いてしまうと、sdisc=-1 の場合、scont にはそのブロックにより最初に 0 が設定され、それから、default ブロックにより -1 が設定されます。

一般に、この現象は「フォールスルー」と呼ばれています。あるブロックが明示的に終了されていない場合には、switch 文の残りの部分が必ず実行されます。これは多層フィルタリングには便利ですが、一般にはよくない形式と考えられているので、すべての case 文を break で終わらせてこの現象を避けてください。

5.4.3 While

while ループは単純なループのモデリングに使用します。一般形は以下のとおりです。

```
while (expressionLog) {
   loopStatement; }
```

ループの条件 expressionLog が評価され、true なら loopStatement ブロックが実行され、expressionLog が再び評価されます。expressionLog が false になると、このループは終了します。

ESDL では、ループ条件 expressionLog は logical 型でなければなりません。

5.4.4 For

for ループは、ESDL でしか使用できないモデリング機能の 1 つとして際立っています。ブロックダイアグラムにはこのループに相当するものはありません。

for ループの一般形は以下のとおりです。

```
for ( initExpression; expressionLog; incrExpression ){
loopStatement; }
```

これにより、以下のループと同じ処理が行われます。

```
initExpression;
while (expressionLog) {
   loopStatement;
   incrExpression; }
```

for ループでは、ループヘッドの initExpression、expressionLog、および incrExpression の各コンポーネントの記述は任意です。ループ条件 expressionLog は logical 型でなければなりません。ループ条件を指定しないと、条件は常に true となり、無限ループとなります。

注記

ESDL では、ループヘッドのコンポーネントは、単純な式でなければなりません。i=0, j=1; や i++, j--; のように式をカンマで区切ったリストは使用できません。つまり、initExpression や incrExpression には、それぞれ 1 つの文しか使用できません。

次の例は、if...else 文と for ループ文を組み合わせた単純な例です。

```
if (log) {
   for (index=0; index < array.length(); index++) {
      array[index] = index * index; }
   }
else {
   for (index=0; index < array.length(); index++) {
      array[index] = index; }
   }
```

この例は、array に値を書き込む処理を表しています。if...else 文の log 条件により、2 つのループのどちらを使用して array に値を書き込むかが決まります。

各ループは array 全体について反復実行され、各セルに 1 つずつ値が設定されます。設定される値は index * index の結果か、あるいは index の値です。

5.4.5 Break

break 文を使えば、以上の各制御エレメントを即座に終了し、次の囲みの始まり、あるいはモデルの残り部分に戻ります。

ESDL のモデル記述ではラベルはサポートされていないので、制御をラベルに戻すラベル付き break 文はありません。

5.5 メソッド

ESDL におけるソフトウェアモデルのファンクション記述はメソッドに内包されます。メソッドは計算を実行してデータを操作するもので、「あるオブジェクトに対する処理」として呼び出されます。

メソッド呼び出しの一般形は以下のとおりです。

```
receiverClassName.doSomething(parameterList)
```

ここで、receiverClassName は、doSomething メソッドを「実行する」受信側オブジェクト名です。カンマで区切られたリストあるいは 1 つのパラメータをパラメータとして parameterList で渡すことができます。メソッド呼び出しも含めて、どのような式でもパラメータにすることができます。

ESDL では、以下のようなメソッド呼び出しを行えます。

```
loader.resolve(false, 1.76);
//do not use characteristic, calculate value for 1.76


numbers.setAt(10*index, index);
//set array numbers to 10*index at index


(12.4)between(valA, valB);
//check if 12 is between valA and valB


array.length();
//return array length
```

注記

メソッドにパラメータがない場合でも、そのメソッド名の後ろには必ずかっこを付けて、その文がメソッド呼び出しとして解釈されるようにする必要があります。

メソッド呼び出しは 1 つの値を戻すことができるので、その値を変数に代入することができます。この変数は戻り値と同じ型でなければなりません。

```
aNumber = anArray.getAt(index);
//assign value from index position


anOffset = loader.resolve(true, 2.14);
//assign value for 2.14, calculate using characteristic
```

戻り値を戻すメソッドの場合、そのボディは必ず return 文で終わっていなければなりません。return 文の後ろには、メソッドの戻り値の型になる任意の式を指定できます。

```
return in.between(ub, lb);
// returns a logical value


return intVar;
//returns the value of intVar
```

メソッドから戻すことができる値は 1 つだけです。2 つ以上の値をモジュール間あるいはオブジェクト間で受け渡ししたい場合には、その値を格納するオブジェクトを使用します（ 126 ページの「構造体」の項を参照してください）。

ESDL では、メソッド呼び出しをネストさせることはできません。次の文は無効です。

```
loader.resolve(true, 2.14).sqrt();
```

これは、次のような文に置き換える必要があります。

```
aNumber = loader.resolve(true, 2.14);
aNumber.sqrt();
```

あるオブジェクトの変数へのアクセスとして、直接アクセスメソッドが有効な場合に限り、メソッド呼び出しをネストさせることができます。したがって、変数 aa が anObject 内で定義されている場合には、次のようなネスト文は有効です。

```
anObject.aa().sqrt()
```

5.5.1 This

ESDL では疑似識別子 this を使用して、現在のコンポーネントのメソッドを呼び出すことができます。たとえば、現在のオブジェクトのプライベートメソッド initCounter を呼び出す場合には、以下の文を使用できます。

```
this.initCounter();
```

initCounter メソッドが戻り値を戻す場合には、それを次のように代入することができます。

```
aValue = this.initCounter();
```

以上のどちらの場合も現在のオブジェクトが暗示されているので、識別子 this を使用するかどうかは任意です。そこで、前ページの文を以下のように書くこともできます。

```
initCounter;
aValue = initCounter();
```

ただし現在のオブジェクトがパラメータとして別のメソッドに渡される場合は、this は必ず必要です。

```
OtherObject.evaluate(this);
```

この場合、識別子 this は現在のオブジェクトの参照を渡します。

注記

ESDL では self と this の両方の識別子を使用できますが、this を使用して Java の構文との互換性をとることをお勧めします。

5.5.2 アクセス制御

ESDL では、クラスのメソッドと変数をパブリックまたはプライベートとして宣言することができます。エレメントをプライベートとして宣言することにより、そのエレメントに対する外部からのアクセスを制限し、それらのコードを他のオブジェクトから隠蔽することができます。

このように、「プライベートメソッド」の呼び出しやプライベート変数の操作は、現在のオブジェクトの中からしか行うことができません。それとは対照的に、「パブリックメソッド」の呼び出しやパブリック変数へのアクセスは、現在のオブジェクトの中と外の両方から行うことができます。

メソッドは、ESDL エディタ内でパブリックダイアグラムまたはプライベートダイアグラムに割り当てることにより、パブリックメソッドまたはプライベートメソッドとして宣言されます。新しいオブジェクトには、`Main` というデフォルトのパブリックダイアグラムが 1 つ存在し、その中に `calc` というデフォルトメソッドがあります。

この新しいオブジェクトには、デフォルトのパブリックダイアグラム内に別のパブリックメソッドを作成したり、新しいダイアグラムを追加することができます。プライベートメソッドは、必ずプライベートダイアグラムの一部として作成されます。メソッドを 1 つのダイアグラムから別のダイアグラムに移動すれば、そのメソッドに対するアクセス権も変更されます。

オブジェクト `Caller` から、もう 1 つのオブジェクト `Receiver` のパブリックインターフェースにアクセスできるようにするには、それを `Caller` の Elements リストに追加してインポートします。

新しい変数を ESDL エディタの Elements リストに追加すると、その変数は「プライベート変数」として作成されます。これらの変数は現在のオブジェクトの外からはアクセスできません。また変数のステータスは、そのオブジェクト用のエレメントエディタでしか変更できません（『ASCET ユーザーズガイド』の「エレメントプロパティの編集」の項を参照してください）。

5.5.3 直接アクセスメソッド

パブリック変数を作成すると、現在のオブジェクトのインターフェースに 2 つのメソッドが自動的に追加されます。これらのメソッドは直接アクセスメソッドと呼ばれます。直接アクセスメソッドを呼び出せば、パブリック変数のデータにアクセスできるので、このメソッドをパブリック変数に対する読み取りと書き込みの両方のアクセスに使用できます。

以下の例では、`VisibleObject` には `free` と `all` という 2 つのパブリック変数があると仮定します。以下のようにすれば、外側からメソッドを呼び出すことができます。

```
sdisc tmp = VisibleObject.all();
VisibleObject.free(120);
```

変数がパブリック変数として宣言されると、直接アクセスメソッドが自動的に生成され、オブジェクトのパブリックインターフェースに追加されます。これらのメソッドを明示的にコーディングする必要はありません。

5.6 集合データ型

ESDL でサポートされる集合データ型は、2 つのグループに大別されます。1 つは一般的な配列やマトリックスで、もう 1 つは 1 次元テーブル、2 次元テーブル、およびディストリビューションを含む集合型です。後者は特性カーブやマップに使用されます。

以下に、まずは配列やマトリックスについて説明し、続いてテーブルやディストリビューションについて説明します。

5.6.1 配列

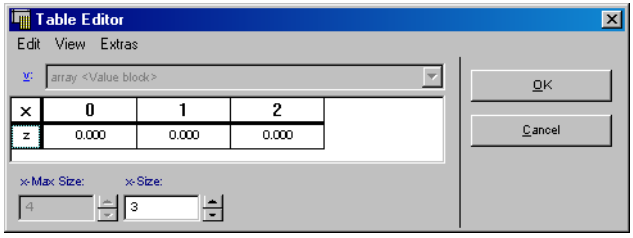
「配列」（「Array」）は、同じデータ型の変数の 1 次元配列で、インデックスによって参照されます。ESDL では、どの基本データ型でも配列を作ることができます。配列内の変数には配列のインデックスを使ってアクセスします。先頭の配列要素のインデックス位置は 0 です。

配列をモジュールに追加するには、その配列を ESDL エディタの Elements リストに追加します。配列の型は、エレメントエディタで任意の基本型として指定できます。

配列のサイズとデータの編集には、テーブルエディタを使用します。このテーブルエディタでは、配列の現在のサイズと最大サイズの両方を指定することができます。

配列のサイズを実行時に変更することはできません。配列の最大サイズは 1024 です。

配列データはテーブルエディタで編集することができます。あるいは、データをタブで区切った ASCII ファイルを配列データとして入力することもできます（『ASCET ユーザーズガイド』の「データの編集」-「配列エディタ」の項を参照してください）。



ESDL では、配列要素の読み取り／書き込みには以下の構文を使用します。

```
val = myArray[index];
myArray[index] = val;
```

上の 1 つめの文は配列 `myArray` の位置 `index` の要素の値を読み取り、その値を変数 `val` に代入します。`val` は、myArray と同じ型でなければなりません。配列のインデックスカウントは 0 から始まるので、`myArray[3]` は `myArray` の 4 番目の要素を返します。

2 つめの文は配列 `myArray` の位置 `index` の要素に `val` の値を設定します。`val` は、myArray と同じデータ型でなければなりません。

パブリックインターフェース：

表 5-3 は、配列用のパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`length()`	udisc	配列の要素数を取得する
`getAt(index)`	配列の型	配列の位置 `index` の要素を取得する
`setAt(val, index)`	void	配列要素に `val` の値を設定する

表 5-3　配列のパブリックインターフェース

5.6.2　マトリックス

「マトリックス」（「Matrix」）は同じデータ型の変数の 2 次元配列で、インデックスによって参照されます。ESDL では、どの基本データ型でもマトリックスを作ることができます。マトリックス内の変数へのアクセスには x と y の 2 つのインデックスを使用し、先頭のインデックスは 0 です。

ESDL エディタでのマトリックスの追加と操作は、配列と同じ方法で行うことができます（120 ページの「配列」の項を参照してください）。マトリックスのサイズを実行時に変更することはできません。マトリックスの各次元の最大サイズは 64 です。

ESDL では、マトリックスの要素の読み取り／書き込みには以下の構文を使用します。

```
val = matrix[indX, indY];
matrix[indX, indY] = val;
```

上の 1 つめの文は配列 `matrix` の列 `indX`、行 `indY` の要素の値を変数 `val` に代入します。`val` は、matrix と同じ型でなければなりません。インデックスカウントは 0 から始まるので、`myMatrix[2, 3]` はマトリックス `myMatrix` の第 4 行第 3 列の要素を返します。

2 つめの文は配列 `matrix` の列 `indX`、行 `indY` の要素に `val` の値を設定します。`val` は、matrix と同じデータ型でなければなりません。

パブリックインターフェース：

表 5-4 は、マトリックスに使用できるパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`xLength()`	udisc	マトリックスの列数を取得する
`yLength()`	udisc	マトリックスの行数を取得する
`getAt(indX, indY)`	マトリックスの型	マトリックスの位置 `indX`、`indY` の要素を取得する
`setAt(val, indX, indY)`	void	マトリックスの位置 `indX`、`indY` の要素に `val` の値を設定する

表 5-4　マトリックスのパブリックインターフェース

5.6.3 1 次元テーブル

1 次元テーブルは、アルゴリズムを使用せずに一連のブレイクポイントによって出力値を求めるための「特性カーブ」のモデリングに使用されます。

テーブル内に複数の x 座標ポイント x_n（入力値）とそれに対応する出力値 y_n を定義しておき、現在の入力値に対する出力値をこの 1 次元テーブルから取得します。また線形補間や丸め補間を使用して、ブレイクポイント間のあらゆるポイントの値を出力することができます。

1 次元テーブルをモジュールに追加するには、そのテーブルを ESDL エディタの Elements リストに追加します。データ型として、任意の算術型をエレメントエディタで指定できます。

1 次元テーブルには、最大 1024 組の座標ポイントと出力値を設定できます。テーブルは、配列やマトリックスとは異なり、「パラメータ」として扱われるため、モデル内で座標ポイントや出力値への書き込みは行えません。

テーブルのデータはテーブルエディタで編集することができます。また、タブで区切ってデータを記述した ASCII ファイルをテーブルデータとして読み込むこともできます（『ASCET ユーザーズガイド』の「データの編集」の項を参照してください）。

ブレイクポイントの補間モードは、テーブルエディタにおいて「丸め補間」または「線形補間」のいずれかを指定することができます。丸め補間では、隣接する下側（左側）のブレイクポイントの出力値が現在の入力値に対する出力値として使用され、線形補間では前後のブレイクポイントの出力値の間を結ぶ直線上から現在の出力値が求められます。

パブリックインターフェース：

ESDL では、テーブルにはパブリックインターフェースを使用してしかアクセスできません。表 5-5 は、1 次元テーブル用のパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`search(index)`	void	テーブルの座標ポイントとして `index` の値を設定するか、または `index` の補間係数を算出する
`interpolate()`	テーブルの型	テーブルから、現在の座標ポイントの値を取得するか、または補間によって出力値を算出する
`getAt(index)`	テーブルの型	座標ポイントに `index` を設定し、対応する値を取得するか、`index` の補間係数を求めて値を補間する

表 5-5 1 次元テーブルのパブリックインターフェース

線形補間：

次の例では、1 次元テーブルの線形補間について説明します。以下の値が設定されているテーブル LLpr を使用します。

0.0	1000.0	2000.0	3000.0	4000.0	5000.0	6000.0
0.0	0.8	1.1	1.5	1.8	2.0	2.2

1 次元テーブルの評価を行うには、通常は `getAt(index)` メソッドが適しています。この場合、線形補間は以下のように行われます。

```
tmpVal = LLpr.getAt(3000);
// tmpVal に 1.5 を代入する
tmpVal = LLpr.getAt(2280);
// 2280 の補間係数を求め、
// その補間係数に対応する値 1.212 を求めて
// それを tmpVal に代入する
tmpVal = LLrp.getAt(9000);
// 9000 の補間係数を求め、
// その補間係数に対応する値 2.2 を求めて
// それを tmpVal に代入する
```

場合によっては、座標ポイントの検索と補間処理を別々に行った方がより効率的な場合があります（実験ターゲットのコード生成時など）。この場合、線形補間は以下のように行われます。

```
LLpr.search(1000);
// 入力座標値として 1000 をセットする
tmpVal = LLpr.interpolate();
// tmpVal に 0.8 を代入する
LLpr.search(2780);
// 2780 の補間係数を求める
tmpVal = LLrp.interpolate()
// 2780 の補間係数に対応する値 1.412 を求めて
// それを tmpVal に代入する
```

5.6.4 2 次元テーブル

アルゴリズムを使用せずに一連のブレイクポイントによって出力値を求めるための「特性マップ」のモデリングに使用されます。

テーブル内に複数の x/y 座標ポイント（x_n:y_n）とそれに対応する出力値 y_n を定義しておき、現在の入力値に対する出力値をこの 2 次元テーブルから取得します。また線形補間や丸め補間を使用して、ブレイクポイント間のあらゆるポイントの値を出力することができます。

ESDL エディタでの 2 次元テーブルの追加と操作は、1 次元テーブルと同じ方法で行うことができます。2 次元テーブルには、最大 64x64 組の座標ポイントと値を設定できます。

パブリックインターフェース：

ESDL では、テーブルにはパブリックインターフェースを使用してしかアクセスできません。表 5-6 は、2 次元テーブル用のパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`search(indX, indY)`	void	テーブルの現在の座標ポイントに `indX` および `indY` の値を設定するか、`indX` および `indY` の補間係数を算出する
`interpolate()`	テーブルの型	テーブルから、現在の座標ポイントの値を取得するか、補間によって値を算出する
`getAt(indX, indY)`	テーブルの型	座標ポイントに `indX` および `indY` を設定し、テーブルから対応する値を取得するか、`indX` および `indY` の補間係数を求めて値を補間する

表 5-6　2 次元テーブルのパブリックインターフェース

線形補間：

以下の例で、2 次元テーブルの線形補間について説明します。以下の値が設定されているテーブル `LLpr1` を使用します。

```
y \ x     0.0       1.0       8.0       15.0
1.0       -5.0      -3.0      0.0       1.0
3.0       0.0       1.0       4.0       6.0
5.0       8.0       5.0       4.0       4.0
```

適合マップの評価を行うために必要な機能は、すべて `getAt(indX, indY)` メソッドに含まれています。この場合、線形補間は以下のように行われます。

```
tmpVal = LLpr2.getAt(8,5);
// tmpVal に 4.0 を代入する
tmpVal = LLpr.getAt(0.5,1.5);
// x=0.5、y=1.5 についての補間係数を求め、
// その補間係数によって補間された値 -2.875 を
// それを tmpVal に代入する
tmpVal = LLrp2.getAt(20,10);
// x=20、y=10 の補外係数を求め、
// その補外係数によって補外された値 5.0 を求めて
// それを tmpVal に代入する
```

適合マップの場合も、座標ポイントの検索と補間のステップを別々に行った方がより効率的になる場合があります。この場合、線形補間は以下のように行われます。

```
LLpr.search(1,3);
// x と y の座標ポイントとして、1 と 3 をセットする
tmpVal = LLpr2.interpolate();
// tmpVal に 1.0 を代入する
LLpr2.search(4,4);
// x=4、y=4 の補間係数を求める
tmpVal = LLrp2.interpolate()
// 補間係数によって補間された値 3.143 を求めて
// それを tmpVal に代入する
```

5.6.5 ディストリビューションとグループテーブル

2 つ以上の特性カーブやフィールドに同じ座標ポイントセットを使用して、それらの特性カーブやフィールドを相互に関連させることができます。ASCET では、このように共用される座標ポイントセットは「ディストリビューション」としてモデリングされ、ディストリビューションに定義された座標ポイントを使用するテーブルは「グループテーブル」と呼ばれます。

ディストリビューションは座標ポイントの配列です。この配列の要素の値は必ず順に増加していなければなりません。ディストリビューションは、1 次元テーブルと 2 次元テーブルの両方に使用できます。2 次元テーブルの場合、各次元に 1 つのディストリビューションが必要です。

ディストリビューションとグループテーブルを使用すると、補間係数が 1 回だけ算出され、それがグループテーブルのすべてのテーブルについて再利用されるので、計算の所要時間と所要メモリが大幅に少なくなります。

ESDL エディタでグループテーブルを作成する場合、最初にディストリビューションを作成してから、それを使用するグループテーブルを作成します。グループテーブルを作成すると、システムはそれに対応するディストリビューションの指定を要求します。ESDL エディタにおいては、ディストリビューションを既存のグループテーブルに割り当て直す、ということはできないので、必ずテーブルを追加する前にディストリビューションを作成しておく必要があります。

テーブルエディタを使用して、ディストリビューションとグループテーブルの両方のデータを編集することができます。値の連続性に反するディストリビューション（値が順に増加していないディストリビューション）は無効です。また、タブで区切った ASCII ファイルとして作成されたデータを読み込むこともできます。

パブリックインターフェース：

グループテーブルには、プレーンなテーブルの場合とは異なり、`getAt` メソッドがありません。パブリックインターフェースはディストリビューション用のものとグループテーブル用のものに分かれ、前者には `search` メソッドが、後者には `interpolate` メソッドが割り当てられます。

2 次元テーブルでは、座標ポイントが両方の次元について指定されていないと、補間による出力値の算出を行うことができません。

表 5-7 に、ESDL のディストリビューションのパブリックインターフェースを示します。

メソッド	戻り値の型	用途
`search(index)`	void	ディストリビューションの座標ポイントに `index` を設定するか、または `index` の補間係数を算出する

表 5-7　ディストリビューションのパブリックインターフェース

表 5-8 に、ESDL のグループテーブルのパブリックインターフェースを示します。

メソッド	戻り値の型	用途
`interpolate()`	テーブルの型	テーブルから、現在の座標ポイントに対応する出力値を取得するか、または補間によって算出する

表 5-8　グループテーブルのパブリックインターフェース

5.7　構造体

ESDL では、構造体（「レコード」とも呼ばれます）はクラスを使用してモデリングされます。クラスは任意の数の変数を保持する複合的なコンテナエレメントとして使用できます。変数が、あるクラス内のパブリック変数であれば、直接アクセスメソッドを使用して ESDL からその変数の読み取り／書き込みを行うことができます。

コンテナエレメントとして使われるクラスには、ESDL の他のクラスと同じ方法でアクセスします。最初のステップとして必ず行うことは、クラスを ESDL エディタの Elements リストに追加して、現在のクラス内で使用できるようにすることです。変数は、親オブジェクトのレイアウトエディタでパブリックとして宣言できます。

次のようなシンプルな直接アクセスメソッド構文を使用して、ESDL で変数にアクセスすることができます。

```
theVar = VisibleObject.aVar()
VisibleObject.aVar(5.12);
// 単純な変数への read/write アクセス

theVar = VisibleObject.anArray().getAt(2)
VisibleObject.anArray().setAt(2.14, 3);
// 配列変数への read/write アクセス
```

グループテーブルとディストリビューションの場合、この構文は使用できません。

ESDL では、クラスをネストさせて、自己参照構造体をモデリングすることができます。

注記

`VisibleObject.anArray(myArray)` のような複雑な代入は ESDL では無効なので、このように指定しても `myArray` パラメータの値は `anArray` の要素には代入されません。しかし、複雑な文を使用して、別のオブジェクトに参照を渡すことはできます。

5.8 メッセージ

ASCET における「メッセージ」は、リアルタイム言語構造体としてプロセス間通信に使用され、リアルタイム環境において「保護されたグローバル変数」として扱われます。

メッセージはモジュール内でのみ使用できます。モジュール内から見ると、メッセージは読み取りまたは書き込み、あるいはその両方を行うことができる通常の変数のように見えます。しかし実際には、あるプロセスが実行されると、オペレーティングシステムはそのすべてのメッセージのコピーを作成します。これらのコピーには、そのプロセスのインスタンスからしかアクセスできません。

こうして、複数のプロセスが同じメッセージを使用する場合、各プロセスはそのメッセージの自分専用のコピーを取得します。これは、リアルタイムオペレーティングシステムによるマルチプロセス環境でのデータの整合性を保証するための方法です。

メッセージはすべてのモジュールで使用することができます。メッセージの追加は、ESDL エディタのツールバーから対応するアイコンを選択すれば、Elements リストで他のエレメントと同様に行うことができます。次の 3 種類のメッセージを追加することができます。

- 送信メッセージ：現在のモジュール内でこの変数に書き込むことができます。
- 受信メッセージ：現在のモジュール内でこの変数を読み取ることができます。
- 送受信メッセージ：現在のモジュール内でこの変数の読み取りと書き込みができます。

EDSL では、メッセージには以下のような代入文を通じてアクセスします。

```
theVar = receiveMsg + 1.24;
sendMsg = 12;
theMessage = 3 * tmpVar;
```

パブリックインターフェース：

表 5-9 は、メッセージ用のパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`receive()`	void	メッセージを読み取る
`send()`	void	メッセージを書き込む

表 5-9　メッセージのパブリックインターフェース

5.9　リソース

リソースは、メッセージと同様、モジュール内でのみ使用できます。3.1.3 項に説明されているとおり、リソースには 2 つのメソッド（`reserve` および `release`）があります。ESDL では、これらのメソッドを以下の例のように使用します。

```
resource1.reserve();
do_something();
resource1.release();
```

表 5-10 は、リソース用のパブリックメソッドをまとめたものです。

メソッド	戻り値の型	用途
`reserve()`	void	リソースを予約する
`release()`	void	リソースを解放する

表 5-10　リソースのパブリックインターフェース

5.10　数学関数

ASCET には、定義済みのエレメントで構成される汎用ライブラリが付属していて、これらを新しいモジュールやクラスの基礎ブロックとして使用することができます。

ESDL のモデル記述用には、いくつかの数学関数がシステムライブラリとして提供されています。数学関数はクラス `ETAS_Systemlib_CT¥Classes¥MathFcn` に記述されていて、このクラスを ESDL エディタの Elements リストに追加することによりアクセス可能となります。

下の例は、ESDL のモデル記述から数学関数にアクセスする方法を示しています。

```
// 変数 x のサインを求める
x = x + MathFcn.pi()/2;
y = MathFcn.sin(x);

// 変数 arg の平方根を求める
if (arg > 0) return MathFcn.sqrt(arg);
```

```
// continuous 変数の arg を、logical 変数に型変換する
return (MathFcn.Sign(arg) = 0 ? false : true);

// 配列内の x-1 で示される要素に、x1/x を入れる
udisc x
cont tmp, y;
for (x = 1; x < array.length() + 1; x++) {
   tmp = x;
   array[x-1] = MathFcn.pow(tmp, 1/tmp); }
```

下の表は、MathFcn クラスの関数をまとめたものです。どの数学関数でも、戻り値と引数の型は同じで、continuous 型の変数を引数として入力し、同じく continuous 型の値を戻します。

メソッド	演算
pi()	3.141592654 を戻す
sin(x)	x のサイン
cos(x)	x のコサイン
tan(x)	x のタンジェント
asin(x)	$\sin^{-1}(x)$（アークサイン）
acos(x)	$\cos^{-1}(x)$（アークコサイン）
atan(x)	$\tan^{-1}(x)$（アークタンジェント）
sinh(x)	x のハイパーボリックサイン
cosh(x)	x のハイパーボリックコサイン
tanh(x)	x のハイパーボリックタンジェント
sch(x)	x のハイパーボリックセカント
csch(x)	x のハイパーボリックコセカント
coth(x)	x のハイパーボリックコタンジェント
exp(x)	指数関数 e^x
log(x)	自然対数 $\log_e(x)$、x > 0
log10(x)	底 10 の対数 $\log_{10}(x)$、x > 0
pow(x, y)	x^y
sqrt(x)	x の平方根
abs(x)	絶対値 |x|
sign(x)	x < 0 なら -1、x = 0 なら 0、x > 0 なら 1 を戻す符号関数
limit(m, x, n)	x <= m なら m、m < x < n なら x、x => n なら n を戻すリミッタ
max(x, y)	x と y のうち大きい方の値を戻す
min(x, y)	x と y のうち小さい方の値を戻す

メソッド	演算
`fmod(x, y)`	x/y の浮動小数点演算の剰余。符号は x と同じ
`ceil(x)`	x 以上の最小の整数を戻す（切上げ）
`floor(x)`	x 以下の最大の整数を戻す（切捨て）

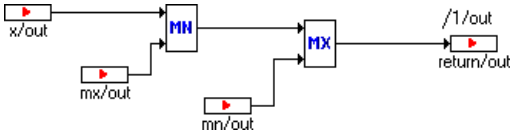
表 5-11 ESDL の数学関数

5.11 ESDL からブロックダイアグラムへのアクセス

この項では、ESDL で単純な制限付き積分器を構築する方法について説明します。この積分器では、`Systemlib_ETAS` フォルダ内のリミッタエレメントを使用して、出力シグナルの帯域幅を決定します。

リミッタエレメントには 3 つのパラメータ `mn`、`x`、`mx` を伴う 1 つのメソッド `out` があります。`out` メソッドは `x < mn` なら `mn` を、`mn <= x <= mx` なら `x` を、あるいは `x > mx` なら `mx` を戻します。

このリミッタエレメントを示すブロックダイアグラムは以下のようになります。



積分器エレメントを構築する

- コンポーネントマネージャで新しい ESDL モジュールを作成し、その名前を `IntegratorLimit` に変更します。
- `IntegratorLimit` 用の ESDL エディタを開きます。
- “Elements” リストに、continuous 型の mem という変数を追加します。積分器のこのメモリには、出力シグナルの値が格納されます。
- “Elements” リストに、フォルダ `Systemlib_ETAS¥Nonlinears¥Limiter` にあるリミッタモジュールを追加します。
- Methods リストに、メソッド `out`、`reset` および `compute` を追加します。デフォルトのメソッド `calc` の名前を `compute` に変更することもできますが、そうしない場合には `calc` を削除します。

- インターフェースエディタを使用して、対応するメソッドインターフェースを以下のように編集します。

メソッド	引数	戻り値の型
compute	cont mx cont in cont mn	void
out	void	cont
reset	cont initVal	void

- 各メソッド用の ESDL コードを入力し、そのメソッドを保存します。各メソッド用の ESDL コードは以下のとおりです。

```
reset(initVal)
mem = initVal;
```

```
cont out()
return mem;
```

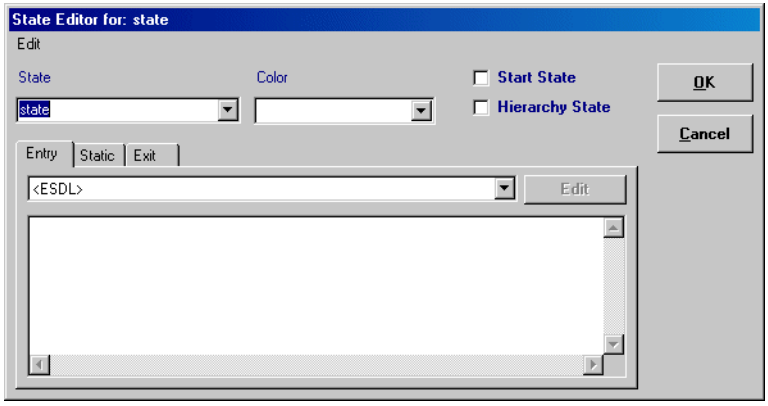
```
compute(mn, in, mx)
mem = mem + K * in * dT;
mem = Limiter.out(mn, mem, mx);
```

この例は、既存モジュールを新しいモジュールの基礎単位として使用する方法を示しています。compute メソッドの第 2 文が、積分器シグナルの範囲を制限します。リミッタの out メソッドはシグナルの値、上限値あるいは下限値のいずれかを戻します。この値が、積分器のメモリに代入されます。

5.12 ステートマシン内での ESDL の使用

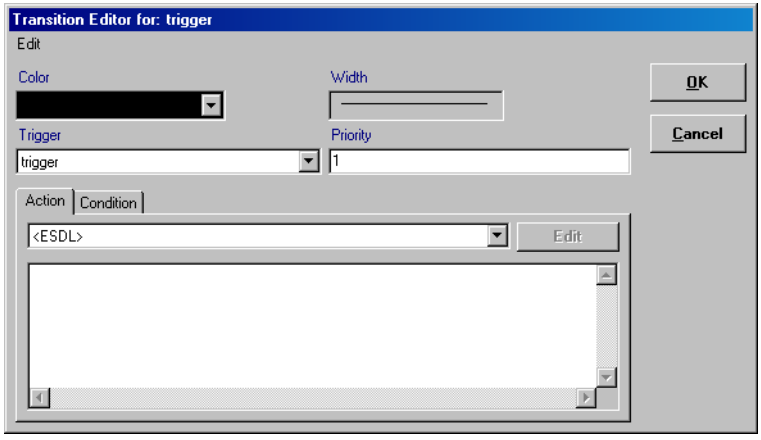
ASCET でステートマシンをモデリングする場合、ESDL で記述する方がブロックダイアグラムよりもコンパクトになることがよくあります。ステート、およびステート間のトランジションのどちらの記述にも、ESDL を使用することができます。

一般に、ステートには最大 3 種類のアクションを持たせることができます。これらのアクションには、それぞれ Entry、Static および Exit のラベルが付いています。これらはそれぞれ、そのステートに入ったとき、そのステートがアクティブな間、およびそのステートが終了するときに実行されます。



ステートのアクションはステートエディタで編集します。このステートエディタで <ESDL> オプションが選択されているアクションは、ESDL で記述することができます。<ESDL> オプションが選択されると、そのアクション用のテキストフィールドが有効になります。このフィールドはシンプルな ESDL エディタになっていて、ここから、そのステートマシンの出力変数と入力変数、およびそのステートマシンの Elements リスト内の他の全アイテムにアクセスすることができます。

通常、ステート間のトランジションには、別のステートへの遷移を行うための条件が記述されます。また、遷移が行われるときに実行されるアクションを指定することもできます。



ステート間のトランジションは、トランジションエディタで編集することができます。これについても、<ESDL> オプションを選択すれば ESDL のテキストフィールドで条件とアクションを記述することができ、Elements リストの全アイテムにアクセスすることができます。

両方のエディタのすべてのテキストフィールドでは、上の例のように標準的な ESDL コードが使用されます。ここで ESDL の構文について注意すべき点は、“Condition” タブに入力される式は論理値を戻すものであるため、最後にセミコロンを付けない、ということです。ESDL でアクションやトランジションを編集する方法については、『ASCET ユーザーズガイド』の「コンディションとアクションを ESDL で定義する」の項を参照してください。

5.13 ESDL と他の記述方式の機能比較

ESDL とブロックダイアグラム

下の表は、モデル記述に ESDL を使用する場合とブロックダイアグラムを使用する場合の相違点をまとめたものです。〇は使用可能、×は使用不可能であることを示します。

	ESDL	ブロックダイアグラム
`this`	〇	×
`self`	〇	〇
% 演算子	〇	×
++、-- 演算子	〇	×
`for` 文	〇	×
アトミックシーケンス	×	〇

表 5-12 ESDL とブロックダイアグラムの比較概要

ESDL と ANSI C（参考）

下の表は、ESDL モデリング言語と ANSI C プログラミング言語の主な相違点をまとめたものです。

	ESDL	ANSI C
ビットデータ型、シフト演算	×	〇
文字列データ型、文字列演算	×	〇
`continue` 文	×	〇
ポインタ演算	×	〇
プリプロセッシング	×	〇

表 5-13 ESDL と ANSI C の比較概要

ESDL と Java（参考）

下の表は、ESDL モデリング言語と Java プログラミング言語の主な相違点をまとめたものです。

	ESDL	Java
継承	×	○
動的インスタンス生成	×	○
ポリモフィズム	×	○
メソッドの多重定義	×	○
明示的型変換	×	○
エラー処理	×	○
ガーベッジコレクション	×	○

表 5-14 ESDL と Java の比較概要

6 ブロックダイアグラムによるボディ記述

ASCET のブロック指向記述言語は、組み込み制御システムのグラフィック記述をサポートします。これによって、ESDL 言語で記述される制御システムを、ほとんどそのままブロックダイアグラムで記述することができます。

本章では、ASCET でブロックダイアグラムを使用してソフトウェアを記述する方法について説明します。次の項ではまずグラフィックによるコンポーネント記述を簡単に紹介し、次にグラフィックモデリング言語の概要を紹介します。

最初の概要の部分では、ブロックダイアグラムで使用できる以下の記述方法をご紹介します。

- エレメント
- 式
- 文

ASCET のブロックダイアグラムと ESDL は、機能的にはだいたい同じです。ブロックダイアグラムと ESDL の相違については、133 ページの「ESDL とブロックダイアグラム」の項にまとめてあります。

6.1 エレメントのグラフィック記述

コンポーネント内で使われる各種エレメントや演算子は長方形のダイアグラムで表され、これらのエレメント間の相互作用は各エレメントを結ぶ接続線で表されます。

エレメントのインターフェースは、ピンで表されます（図 6-1）。メソッドの各引数は、エレメントを表すブロックのフレームに付けられたアーギュメントピン（ブロックを指す小さい矢印が付いているピン）で表され、戻り値はリターンピンで表されます。メソッドの呼び出しはそのリターンピンと関連付けられ、引数も戻り値もないメソッドは、メソッドピンで表されます。

↦ メソッドピン
→ アーギュメントピン
— リターンピン

図 6-1 グラフィックブロックにおけるピンの表記

エレメントの名前はエレメントを表す長方形の下に配置され、またアイコンを使用して、エレメントの機能を図示することもできます。ピンの位置は変更可能です。

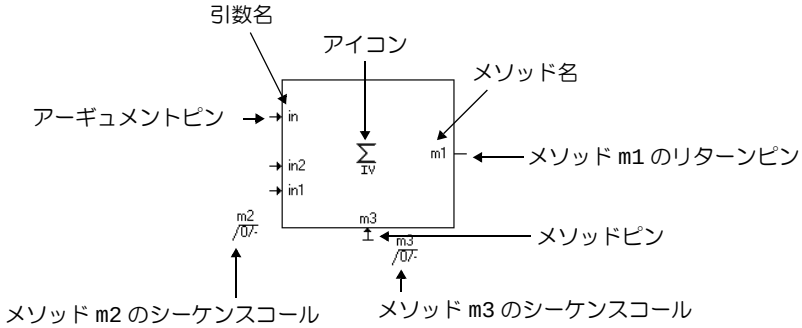


図 6-2　複合エレメントを表すグラフィックブロック

図 6-2 の例は、3 つのメソッドを持つ複合エレメントを示します。メソッド `m1` は 1 つの引数を取り、戻り値を戻します。メソッド `m2` は戻り値を戻さないので、その引数で表されています。メソッド `m3` には引数も戻り値もないので、メソッドピンで表されています。

6.1.1　基本エレメント

エレメントは、ピンで表される引数と戻り値を持つ長方形のブロックとして表されます。各エレメントには名前があり、それはデフォルトではエレメントのブロックの下に表示されますが、この位置は変更することもできます。

基本エレメントには、所定のインターフェースとグラフィック表記が割り当てられています。

基本スカラエレメント

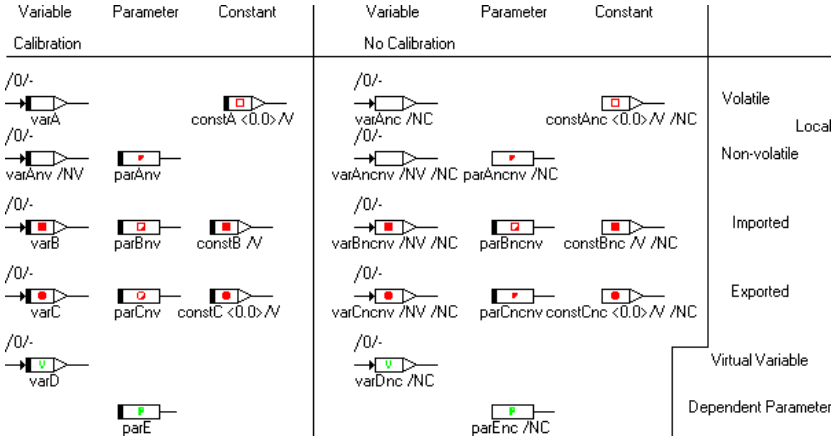
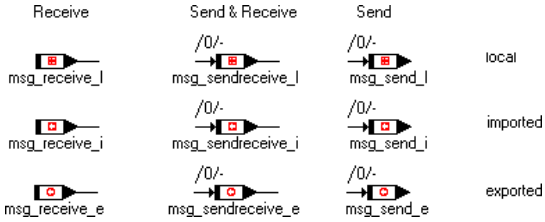


図 6-3　基本エレメントのグラフィック表記

基本スカラエレメントには、値を設定するためのアーギュメントピンが 1 つあり（値の設定が可能な場合のみ）、また現在値を読み取るためのリターンピンが 1 つあります。ブロック内のアイコンがエレメントの種類とスコープを示します。赤く塗りつぶされた正方形は、インポートされたエレメントを表し、塗りつぶされた円はエクスポートされるエレメントを表します。値を書き込むことができない種類の基本スカラエレメント（例、パラメータ）には、アーギュメントピンはありません。適合可能なエレメントの左側には、小さな黒いボックスが表示されます。

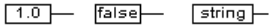
メッセージ

メッセージはプロセスの入出力用変数で、そのタイプにより、1 つまたは 2 つのピンと共に表示されます。以下の図は、Calibration および Volatile 属性の付いたメッセージを示します。これらの属性のいずれか一方または両方を変更すると、表示は図 6-3 のように変わります。



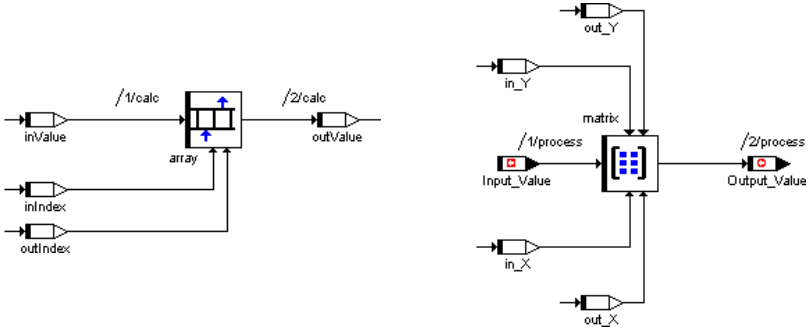
リテラル

リテラルは、以下のような小さいブロックで表され、そのブロックの中にリテラルの値が表示されます。



配列とマトリックス

配列とマトリックスには 2 つのメソッドがあります。1 つは、指定された要素の内容を設定するためのメソッドで、もう 1 つはそれを取得するためのメソッドです。読み取りと書き込みの処理はそれぞれ独立して実行することができます。配列に書き込まれる値は左のアーギュメントピンで表され、それに対応するインデックスは左下のピンで表されます。配列からの読み取りの結果はリターンピンで表され、そのインデックスは右下のアーギュメントピンで表されます。



マトリックスの表示も同様ですが、各メソッドがインデックス引数を 2 つ取る点が異なります。x インデックスは、配列のインデックスのようにブロックの下の 2 つのピンで表されます。y インデックスは、ブロックの上の 2 つのピンで表されます。マトリックスに書き込みを行うための y インデックスは左上のピンで、読み取るときの y インデックスは右上のピンで表されます。

配列やマトリックスがメソッドの引数として渡す場合、あるいは戻り値として戻す場合は、Get あるいは Set ポートを経由して行うことができます。これらのポートを利用するには、描画エリアのショートカットメニュー内の **Get/Set Ports** 機能を使用します。

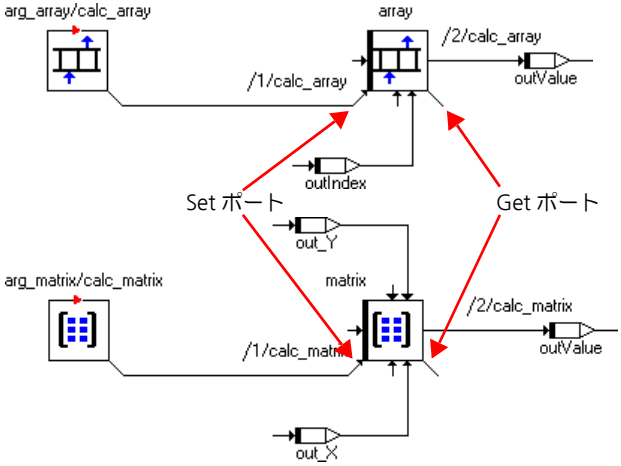


図 6-4　配列およびマトリックス用の Set ポートおよび Get ポート

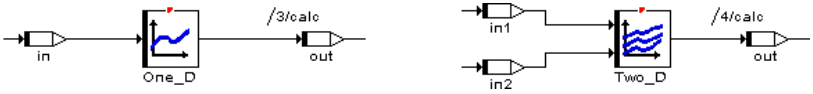
Get ポートは、エレメントのデータ全体を参照するためのポインタを出力し、Set ポートは、エレメントに対してエレメントが参照するメモリエリアの位置を与えます。

例：図 6-4 において、`array` は `agr_array` が使用しているメモリエリアの内容を読み込み、`matrix` は `agr_matrix` が使用しているメモリエリアの内容を読み込みます。ここでは各メモリエリアへのポインタが Get ／ Set ポートで渡されます。なおデータの不整合が発生することを防ぐため、Set ポートへの接続は必ずメソッドの最初のステップで実行するようにしてください。

注記

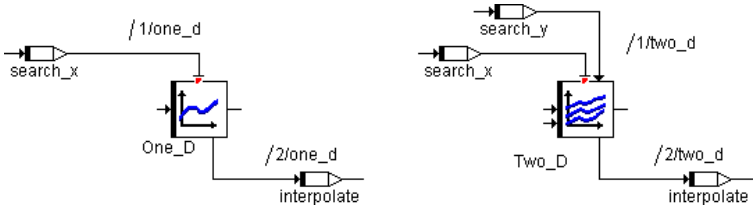
クラスへの受け渡しを行う時も、同じメカニズムが使用されます。

特性テーブル（固定テーブルを含む）の場合、左側にはそのテーブルの次元に応じて、1 つあるいは 2 つのアーギュメントピンがあり、そこから入力値（x 値または x/y 値）が供給されます。また、1 つのリターンピンがあり、そこから補間された出力値が出力されます。

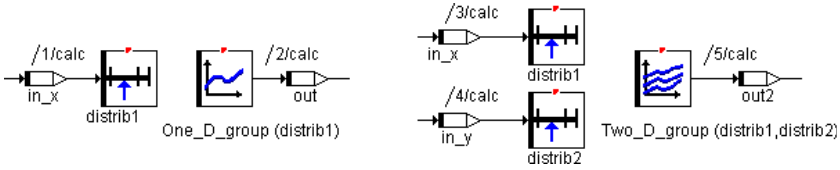


上記の内容は、ESDL で `getAt` メソッドを使用した場合と同じです。（122 ページと 123 ページを参照してください。）

ESDL の場合と同じく、ブロックダイアグラムにおいても座標ポイントの検索と補間を別々に行うことができます。このためには、描画エリア内のショートカットメニューの **Extended Interface** 機能によって、拡張テーブルインターフェースを使用可能にしておく必要があります。



ディストリビューションテーブルには、左側に x 値を供給するためのアーギュメントピンが 1 つあり、グループテーブルには、右側にリターンピンが 1 つあります。グループテーブルにはブレイクポイントのディストリビューションは含まれておらず、1 つまたは 2 つのディストリビューションを参照します。グループテーブルとディストリビューションには拡張インターフェースは含まれません。



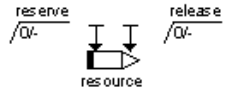
配列やマトリックスと同様、ショートカットメニューの **Get/Set Ports** で Get ／ Set ポートを設定します。

注記

特性テーブルを引数としてメソッドに渡す場合、まずそれをクラスに組み込み、そのクラスを Get ポートで渡す必要があります。

リソース

リソースは、ブロックの上側に reserve と release という 2 つのメソッドが付いています。これらのメソッドは引数も戻り値も伴わないので、メソッドピンで表されます。



インプリメンテーションキャスト

インプリメンテーションキャストは、2 本のピンを持つ小さなひし形で表わされます。

6.1.2 ユーザー定義型のエレメント

ユーザー定義型のエレメントのメソッド、引数、および戻り値は、グラフィックブロックのアーギュメントピンやリターンピンで表されます。これらのレイアウトは任意に変更することができ、また Get ／ Set ポートも利用可能です。

6.2 式

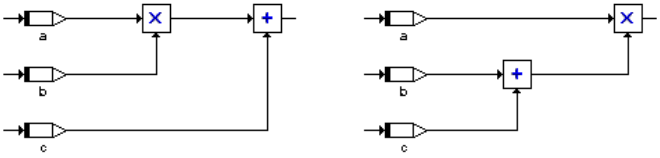
式は、ブロックダイアグラムでエレメントや他の式を演算子で接続して作ることができます。ESDL の場合と同様に、式は以下のように再帰的に定義されます。

- エレメントは式です。
- 演算子の結果は式です（オペランド自体も式です）。
- メソッド呼び出しの戻り値は式です。そのメソッドに引数が供給される場合、それらの引数も式に属します。

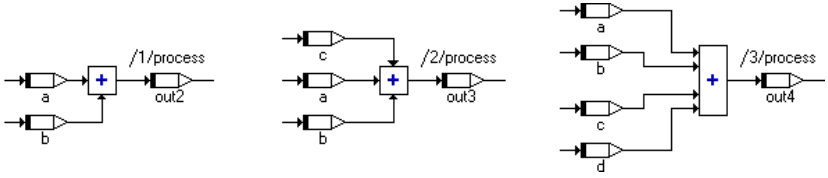
したがって、1 つの式の範囲はその式の中にある基本式（エレメントや、引数を伴わないメソッドの戻り値）により制限されます。

式は、グラフィックではエレメントのリターンピンや演算子をメソッドのアーギュメントピンや他の演算子と接続することにより構築されます。

BDE では、式は接続線と演算子を接続する方法により「まとめられて」いるので、演算子の優先順位の規則はありません。次の例は、式 `(a*b)+c` と `a*(b+c)` の違いを示します。



演算子の引数の評価順序は非常に重要な意味を持つ場合があります。この順序はグラフィック表記では必ず上から下の順ですが、4 つの基本算術演算子の場合は例外で、評価の順序は、以下の図に示すとおりです。



a + b　　a + (b + c)　　a + (b + (c + d))

多くのブロックダイアグラムでは、演算子の引数の数が最大 10 あるいは 20 に制限されます。メソッドの引数の評価順序はそれらが定義されている順序と同じです。エレメントのレイアウトは変更することができるので、レイアウトの順序と定義されている順序とが一致しない場合があります。

6.2.1 算術演算子

算術演算子の機能は ESDL の場合と同じです。加算、減算、乗算、除算、モジュロ（余りを求める）の演算子を使用することができます。加算と乗算の演算子は 2 ～ 10 個の引数を取ることができます。減算と除算の演算子の引数は 2 個だけです。

6.2.2 比較演算子

比較演算子の機能は、ESDL で使われる比較演算子と同じです。以下の比較演算子を使用することができます。

- ＞（より大きい）
- ＜（より小さい）
- ≦（より小さいか等しい）
- ≧（より大きいか等しい）
- ＝（等しい）
- ≠（等しくない）

＝と≠は、算術型以外のエレメントにも適用できます。

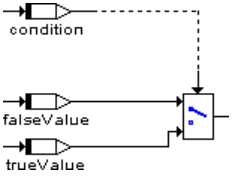
6.2.3 論理演算子

論理演算子 AND、OR および NOT の機能は、ESDL の論理演算子と同じです。AND と OR は 3 つ以上のオペランドにも適用できます。

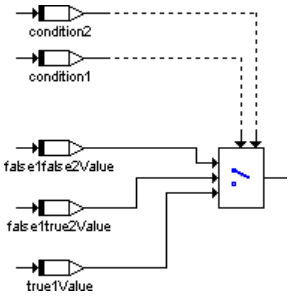
6.2.4 条件演算子

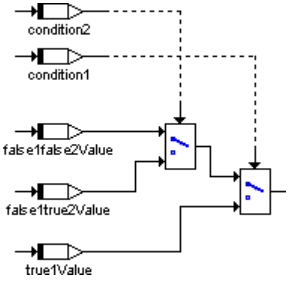
マルチプレクス演算子

条件演算子 (? :) は、グラフィック表記ではマルチプレクス演算子（Mux）と呼ばれます。(condition ? trueValue : falseValue) は以下のように表されます。



マルチプレクス演算子は、複数の引数を直接入力することができます（下図左）。また同じ機能を複数の Mux 演算子をカスケード接続すると、下図右のようになります。

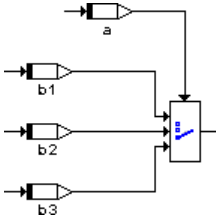




上の例は、`(condition1 ? (true1Value : condition2 ? (false1true2Value : false1false2Value)))` に相当します。つまり、最初の引数は他の引数よりも優先されます。論理型の条件引数を n 個伴うカスケード接続された Mux 演算子は、n+1 個の引数の切替を行い、その中から 1 つを選択することができます。引数の型は任意ですが、すべての引数は互換性のある型でなければなりません。

ケース演算子

ケース演算子は特殊な条件演算子です。論理値ではなく unsigned discrete 型のスイッチ値を取り、このスイッチ値に応じて引数の 1 つを選択します。スイッチ値が 1 なら最初の引数が選択され、2 なら 2 番目の引数が戻されます（以下同様）。スイッチ値が範囲外の場合には、最後の引数が選択されます。



上の例は、`((a=1) ? b1 : ((a=2) ? b2 : b3))` に相当します。

6.2.5 その他の演算子

これまでに説明した演算子の他に、以下の演算子も使用できます。

- Max と Min
- Between

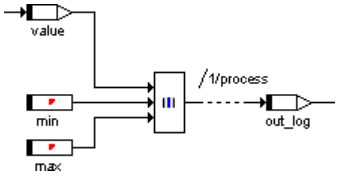
- Abs
- Negation

Max 演算子と Min 演算子

Max 演算子と Min 演算子は、それぞれ引数の最大値と最小値を戻します。どちらの演算子も 2 ～ 20 個の引数を取ることができます。これらの演算子は、算術型のエレメントだけに適用できます。

Between 演算子

Between 演算子は、引数 `value` の値がリミッタ `min` と `max` の間の値かどうかを調べます。もし真なら論理型の戻り値 `out_log` は `true` になり、そうでなければ `false` になります。



このグラフィック表記は、`out_log = (( value >= min ) && ( value <= max ))` に相当します。引数と両方のリミッタは `cont` 型か `discrete` 型でなければなりません。

Abs 演算子

この演算子は、引数の絶対値を戻します。引数と戻り値は、共に `cont` 型か `discrete` 型でなければなりません。

Negation 演算子

Negation 演算子は引数の値の正負の符号を(+ を - に、あるいは - を + に)変えて戻します。引数と戻り値には `cont` 型か `discrete` 型を使用できます。引数が `cont` 型なら、戻り値の型も `cont` になります。

6.3 文

グラフィックによるコンポーネントの定義は、複数のダイアグラムに階層的に分割させることができます。1 つのダイアグラムには、互いに独立して実行できる 1 つまたは複数のメソッドやプロセスが定義されます。どの計算がどのメソッドやプロセスに属し、どのような順序で実行されるかは、「シーケンスコール」により定義されます。

ブロックダイアグラムの各文ごとに、その文をプロセスやメソッドに割り当てるためのシーケンスコールが付加されます。プロセス内あるいはメソッド内での順序は、シーケンスコールの一部である「シーケンス番号」により決まります。シーケンスコールは以下のように表されます。

呼び出されるメソッド

シーケンス番号／呼び出し側のメソッド

ユーザーによって割り当てられたシーケンス番号により、1 つのメソッドまたはプロセスに属する操作の実行順序が決定されます。また組み込みのシーケンシングアルゴリズムを使用して、標準的なブロックダイアグラムの評価順序に対応するシーケンス番号を割り当てることもできます。

一般に、シーケンスコールは以下の 3 つのフィールドで構成されます。

- 呼び出されるメソッドの名前
- 呼び出し側のメソッドまたはプロセスの名前
- 呼び出されるメソッドの、呼び出し側のメソッドまたはプロセス内での位置を決めるシーケンス番号

スカラエレメントの場合、呼び出されるメソッドは、必ず新しい値を代入するメソッドであるため、メソッドの名前はブランクのままになります。

文には以下の 3 種類があります。

- 代入文
- メソッド呼び出し
- 処理フロー制御文（例、`if…then…else`、`while`）

6.3.1 代入

代入文は、ある式の値をあるエレメントに代入します。複合エレメントへの代入の場合、同じ型のエレメントにだけ代入することができます。その場合、代入は値の代入ではなく、参照の代入になります。

特殊なケースとして、メソッドの戻り値に値を代入する場合があります。この代入に割り当てられるシーケンスコールは、そのメソッドの最後のシーケンスコールでなければなりません。

6.3.2 Break 文

`break` 文によりメソッドまたはプロセスの実行が終了し、呼び出し元へ戻ります。この文はメソッドまたはプロセス内の任意の位置で実行できます。

6.3.3 メソッド呼び出し

代入も、メソッド呼び出しの特殊なケースです。ブロックダイアグラムでメソッドを呼び出す場合、関連するシーケンスコールを適切に設定し、メソッド用の引数が正しく供給されるようにする必要があります。

6.3.4 処理フロー制御

ブロックダイアグラムでは、以下の処理フロー制御文を使用することができます。

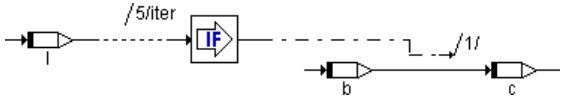
- `If…Then`
- `If…Then…Else`
- `Switch`
- `While`

すべての処理フロー制御文は論理式を評価し、その結果に応じた処理フローの「枝」を有効にします。処理フロー制御の 1 つの枝に複数の文が含まれる場合があります。シーケンスコールによって表される文を、コネクタにより処理フロー制御に接続します。

シーケンスコールのシーケンス番号は、有効になった処理フローの枝に接続されている文の順序を決定します。

If...Then

`If...Then` 文は論理式を評価し、その結果が `True` ならその先の枝が有効になります。処理フロー制御の出力は 1 つあるいは複数のシーケンスコールに接続され、これらのシーケンスコールは、処理フロー制御の枝が有効になると必ずトリガされます。つまり、入力式の値が `True` なら、接続されているシーケンスコールが必ず実行されます。

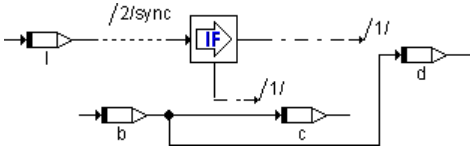


上の例は、次のコードに相当します。

```
if (l) {
   c = b
};
```

If...Then...Else

If…Then…Else は If…Then に似ていますが、処理フロー制御の枝が 2 つあります。論理式の値に応じて、右か下の枝が実行されます。論理式の値が True なら右の枝が実行され、False なら下の枝が実行されます。



上の例は、以下のコードに相当します。

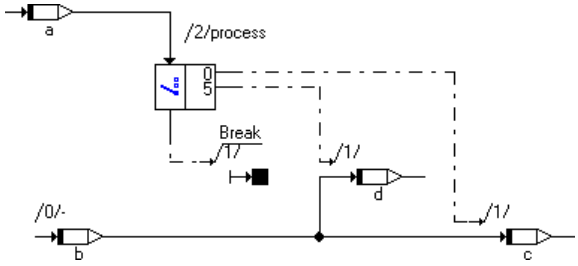
```
if (l) {
   d = b}
else {
   c = b
};
```

ESDL の if...else 文の場合、評価結果が常に true となるものについては、コード生成時に最適化が行われます。詳しくは、113 ページの「If...Else」の項を参照してください。

Switch

Switch 構文は Case 演算子に似ています。Switch は signed discrete または unsigned discrete の値を評価し、その値に応じて、処理フローのいずれかの枝を有効にします。これらの枝は互いに独立しているので、C の switch 構文の場合のような「フォールスルー」は起こりません。

各枝に対応する値を、ユーザー定義することができます。一番下にある最後の枝はデフォルト枝で、入力値がどの枝の値とも等しくない場合に実行されます。



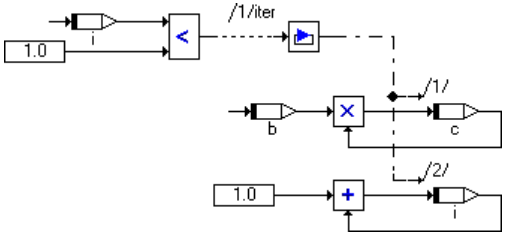
上の例は、以下のコードに相当します。

```
switch (a) {
  case 0: c = b; break;
  case 5: d = b; break;
  default: break;
};
```

While

ブロックダイアグラムで使用できるループ構文は `While` ループだけです。無限ループや、リアルタイムアプリケーションに不適当なループにならないように注意してください。

`If…Then` 文と同様に、論理式の値が `True` ならその先の処理フローが有効になります。この論理入力が `True` である限り、同じ処理が実行されるので、while ループの論理式の値を操作する必要があります。無限ループになるのを避けるためには、ループの最大反復回数を固定の値で定義することをお勧めします。



上の例は、次のコードに相当します。

```
while (i<10) {
   c = b * c;
   i = 1 + i;
};
```

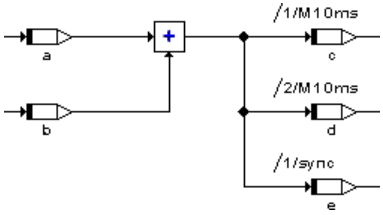
6.4 ブロックダイアグラムの基本動作

ブロックダイアグラムの各要素はそれぞれいずれかのプロセスまたはメソッドに割り当てられ、その実行順序はシーケンスコールのシーケンス番号により決まります。プロセスまたはメソッドが起動されると、シーケンスコールによってそのプロセスまたはメソッドに結びつけられているすべての文が、シーケンス番号の順に実行されます。

一般的なブロックダイアグラムとは異なり、その処理は、必要な時に、つまりその処理のシーケンスコールが起動される時にのみ実行されます。実行の順序は一般的なブロックダイアグラムの「左から右へ」の原則に従っているため、演算（例、加算）を実行するためには、その前にそのすべての引数の値が算出されていなければなりません。

ユーザー定義コンポーネント内のメソッドの引数の評価順序は、それらが宣言されている順序と同じです。ただし、アーギュメントピンはブロックフレームのどこにでも自由に配置し直すことができるので、引数が宣言されている順序とダイアグラムが示す順序とは一致しない場合があります。

オペランドなどの評価は、その結果を使用する文に直接関連付けられます。ここでは、1 つの式に対して複数の評価を行うことができます。



この例では、加算が 3 回行われ、その各回の結果が変数 c、d、および e に代入されます。この加算は 2 つのプロセスにおいて代入に使われているため、複数回実行しないと、どのプロセスで加算が実行されるのかがはっきりしません。式 a + b はプロセス 10ms 内においても 2 回評価されます。

6.4.1 グラフィック階層

グラフィック記述を構造化するために、グラフィック階層を使用することができます。グラフィック階層は視覚的な構造化のためだけに使用され、ブロックダイアグラムの機能には影響を与えません。ブロックダイアグラムの一部分が別の階層に記述され、階層の境界を横切る接続線、つまり、ある階層内のエレメントと階層外のエレメントの接続は、ピンで表されます。ASCET では、ブロックダイアグラムエディタで階層にアイコンを割り当てることができるようになっています。

7 C によるボディ記述

メソッドとプロセスのボディの記述は、ブロックダイアグラムや ESDL 以外に、C コードで行うこともできます。この場合も、他の記述形式と同様、メソッドまたはプロセスのボディのみを記述します。メソッド宣言、関数のヘッドとフレーム、およびデータのインスタンス生成と初期化の処理は、自動的に生成されます。

ESDL やブロックダイアグラムでの記述とは異なり、C コードのコンポーネントはモデルレベルではなく実装レベルで記述されます。

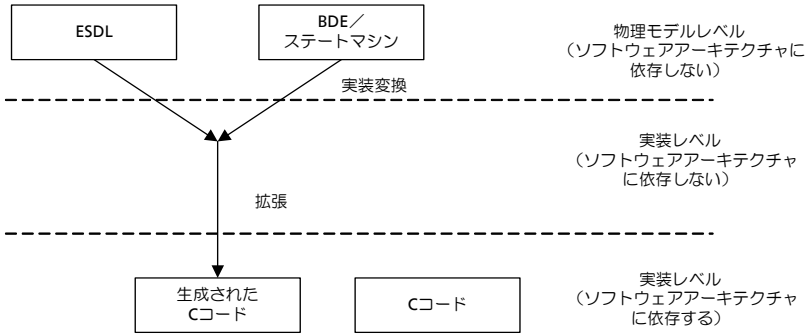


図 7-1　物理モデルからコード実装まで

そのため、C コードの記述には、いくつかの重要な注意点があります。

- モデルレベルから実装レベルへの自動変換は行われません。
- コードのバリエーション（さまざまなターゲット、記述レベル、実装変換）ごとに異なる C コードを記述する必要があります。
- ユーザー定義型を使用する場合、ASCET により生成されるコードのソフトウェアアーキテクチャに C コードを適応させる必要があります。これは、生成される C 関数の命名規則はエクスパンダにより異なり、ユーザーには明らかではない場合があるからです。現在のエクスパンダでは、ユニークなネーム空間を保証するために、生成される関数の名前にクラスの識別タグが使用されます。

7.1 構造体

C コードで記述されるコンポーネントも、ESDL やブロックダイアグラムで記述されたものと同じような構造体になり、メソッドまたはプロセスのボディのみを記述します。またソフトウェアのバリエーションごとに個別のコードを作成する必要があります。

C コードで記述されるコンポーネントの内容は、以下の条件に応じてそれぞれ異なります。

- ターゲット：そのCコードがPC用か、PPC用か、あるいはECUのマイクロコントーラ用かによりコードは変わります。たとえば、あるコントローラのCPUには直接アドレス指定する必要がある特殊なレジスタがあったり、エンディアンフォーマットが異なるなどの理由で、コードが変わります。
- 機能記述レベル：Cコードでは物理レベルを表記することもできます。その場合、実装コードができる限り物理記述と近くなるように、たとえば、`continuous`型は64ビットの浮動小数点数として定義します。またあるいは、固定小数点演算の実装レベルで記述することもできます。
- 選択されたインプリメンテーション（実装レベルのCコードの場合）：変数の実装変換、特にその量子化に依存してコードは変わります。

7.1.1 メソッドとプロセス

各メソッドまたはプロセスについて、1つのC関数が生成されます。関数のヘッドは自動生成されるので、Cコードは関数のボディ内でのみ使われます。

例：

```
a = b + d;
c = a * c;
```

上のようなメソッド`calc()`のボディは、実験ターゲットに要求されるソフトウェアアーキテクチャに応じて、たとえば以下のようなコード（関数のヘッドも含む）として生成されます。

```
void QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc (struct
QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj *self) {
...
   /* BEGIN handwritten code */
   /* calc 1  */a = b + d;
   /* calc 2  */c = a * c;
   /*  END  handwritten code */
...
}
```

コンポーネントのメソッドやプロセス用に生成される関数の名前は、コードエクスパンダと生成されるコードのソフトウェアアーキテクチャに依存します。ユーザーにはこれらの名前を指定することはできません。ユニークなネーム空間がコードエクスパンダにより確保されるため、異なるクラスで同じメソッド名を使っても、名前の競合は起こりません。上の例では、コンポーネントの識別タグを使用して、メソッド`calc`用のユニークな名前`QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc`が生成されています。

7.1.2 変数、および関数のパラメータ

コンポーネントの変数は、関数のヘッドと同様に自動生成されるデータ構造体内に保持されます。ユーザーには、このデータ構造体に対するアクセス権はありません。このデータ構造体の一部は、コンポーネントのインスタンス変数で構成されていて、どのメソッドでも使用することができます。ですから、これらの変数は生成されるすべての関数に渡されなければなりません。このデータ構造体もコードエクスパンダに依存するので、その正確な名前はユーザーからは隠されています。

前ページの例では、コンポーネントには独自のデータ構造体があり、このデータ構造体が、メソッド calc 用に生成される関数に渡されています。このデータ構造体は以下のようになっています。

```
struct QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_Obj  {
   ASDObjectHeader objectHeader;
   real64_Obj *a;
   real64_Obj *b;
   real64_Obj *c;
   real64_Obj *d;
};
```

エレメント名は ANSI C で有効な識別子でなければなりません。C で予約されているキーワードと、`self` および `this` という名前は使用できません。

注記

コンポーネントを C コードで記述する場合、メソッドボディで呼び出される関数の名前が、同じコンポーネントのインターフェースで定義されている変数の名前と同じにならないように注意してください。

エレメントへのアクセス：

コンポーネントのエレメントに容易にアクセスできるようにするために、各エレメント用のマクロが自動的に生成されます。これにより、各エレメントには単純にそのエレメント名を使ってアクセスすることができるようになります。

他のコンポーネントで定義されているパブリックエレメントには、C 関数内から `DefiningObject.PublicElement` という表記を使用してアクセスすることができます。アクセスの対象は基本エレメント、配列およびマトリックスに限定されます。他のコンポーネントで定義されている複合エレメントのパブリックインターフェース（たとえば、ESDL の場合のように `getAt`、`setAt` や `search` および
`interpolate` メソッドの使用）には、C 関数からはアクセスできません。

インスタンス変数について自動生成される `#define` 文：

```
#define a self->a->val
#define b self->b->val
```

```
#define d self->d->val
#define c self->c->val
/* BEGIN handwritten code */
/* calc 1  */a = b + d;
/* calc 2  */c = a * c;
/*  END  handwritten code */
#undef a
#undef b
#undef d
#undef c
```

基本エレメントの扱い：

3 章「型とエレメント」で説明されているように、基本型にはその型のメソッド名を使用することができます。配列やマトリックスにアクセスするときには、C のような方法でインデックス演算子‘[]’を使用することができます。

ユーザー定義型のメソッド名はエクスパンダに依存するので、そのメソッドの呼び出しは、そのメソッド用に生成された関数の名前が正確に使用しなければなりません。前ページの例では、メソッド `calc` 用に生成された関数名は `QX040H28HJ8HAMDJ870S4G7MDIBQQLSM_calc` でした。

ASCET で定義されたエレメントを使用する場合、これらのエレメントの型はモデル型（基本型またはユーザー定義型）です。基本型には以下のようなデフォルトのインプリメンテーションが与えられます。

- `continuous = real64`
- `udisc = unsigned int32`
- `sdisc = signed int32`
- `log = int16`

注記

`logical` 型のエレメントを C コードで数値として使用することは避けてください。ASCET の今後のリリースにおいて、デフォルトインプリメンテーションが変更される予定になっています。

記述レベルを（たとえば固定小数点コードに）切り替えるときには、デフォルトのインプリメンテーションはユーザー定義のインプリメンテーションに置き換えられます。モデル型 `logical` のエレメントをビットとして表記することも可能ですがそうすると、このエレメントは C コードの数値として使用できなくなります。

メッセージ：

メッセージは、ASCET モデル内で使用されるタスク間（プロセス間）通信のコンセプトの 1 つです（1.3 項を参照してください）。ASCET でのコード生成時において、データの整合性を保つために使用されるメッセージコピーが生成されます。

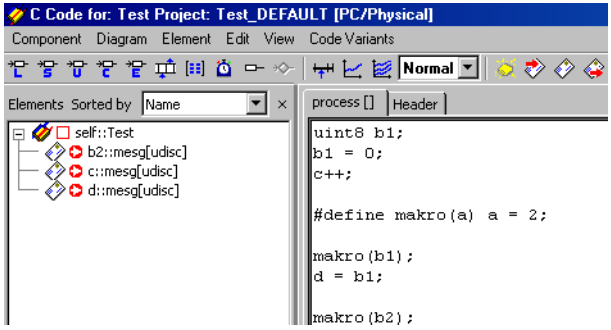
C 関数内でメッセージを使用する場合、オリジナルメッセージからローカル変数（「ローカルコピー」と呼ばれます）に確実に値をコピーするコードを追加する必要があります。プロセスボディ内ではこのローカルコピーのみを使用し、最後に、関数内で値が変更された可能性のあるすべてのローカルコピーをオリジナルメッセージに書き戻します。

ESDL またはブロックダイアグラムで記述されたコンポーネントの場合は、プロセス内でどのメッセージが変更されるかを ASCET が検知することができます。しかし C コードでボディを記述する場合にはユーザーの責任においてデータの整合を維持する必要があります。

ASCET は一般的に、ユーザーによって記述された C コード内において変数がどこでどのように使用されているかを検知することができません。ASCET が検知できるのは非常に限られた場合のみで、たとえば、変数名の次に = が記述されていたり、++ などの代入演算子が使用されているような箇所に限られます。変数の値がマクロ内や外部関数内で変更されたり、ポインタ演算などににによって間接的に変更されたような場合は、検知できません。

そのため、メッセージが使用される際、プロセスの先頭で生成されたメッセージコピーの値がプロセス内で変更されたにも関わらず、状況によってはプロセスの最後で元のメッセージに書き戻されない、ということが起こる場合があります。

この状況を、簡単な例をあげて説明します。以下に示されるモジュールには b2、c、d という 3 つのメッセージが含まれています。ここで、メッセージ c および d には値が直接書き込まれていますが、b2 にはマクロ内で値が書き込まれています。



このモジュールについて生成されたコードを見てみると、3 つのすべてのメッセージについてコピーが生成されます（1）が、ASCET が認識できる方法でアクセスされるのは c と d だけであるため、モジュールの最後では、これらの 2 つのメッセージコピーの値だけが書き戻されています（2）。つまり、マクロ内で b2 の値が変更されることを ASCET が認識できないため、変更された値は失われてしまいます。

```
/*
 *---------------------------------------------------------------------------
 *     Function definitions - Algorithms
 *---------------------------------------------------------------------------
 */

void initClass_TEST (TEST_Class *class)
{
   class->b2 = v3_initInstance_uint32 (0, ASD_VARIABLE);
   class->d = v3_initInstance_uint32 (0, ASD_VARIABLE);
   class->c = v3_initInstance_uint32 (0, ASD_VARIABLE);
}

/*public*/
void TEST_process (void)
{
   uint32 _t1uint32;
   uint32 _t2uint32;
   uint32 _t3uint32;
   SUSPEND_HS_INTERRUPTS
   _t1uint32 = TEST_ClassObj.b2->val;
   _t2uint32 = TEST_ClassObj.c->val;
   _t3uint32 = TEST_ClassObj.d->val;
   RESUME_INTERRUPTS
   {
/* process 1    */       uint8 b1;
/* process 2    */       b1 = 0;
/* process 3    */       _t2uint32 ++;
/* process 4    */
#define makro(a) a = 2;
/* process 6    */
/* process 7    */       makro(b1);
/* process 8    */       _t3uint32  = b1;
/* process 9    */
/* process 10   */       makro(_t1uint32 );
/* process 11   */
/* process 12   */
   }
   SUSPEND_HS_INTERRUPTS
   TEST_ClassObj.c->val = _t2uint32;
   TEST_ClassObj.d->val = _t3uint32;
   RESUME_INTERRUPTS
}
```

(1)

(2)

引数：

メソッドの引数は、そのメソッド用に生成される関数のパラメータリスト内のパラメータに対応付けられます。これらの引数にも、その引数名を使用してアクセスすることができます。

ローカル変数：

一般的な C の規則に従って、関数のローカル変数をメソッドボディ内で宣言することができます。メソッドボディ内では C のデータ型の変数だけを宣言でき、ASCET のモデル型の変数は宣言できません。特に、ユーザー定義型のローカル変数をコンポーネント内の C によるボディ指定内で使用することはできません。

```
real64 i;
for (i=0; i < 10; i++)
{
```

```
    sum = sum + a[i];
}
```

インプリメンテーションの種類ごとに対応するコードのバリエーションがあるので、ユーザーはインプリメンテーションごとのローカル変数とそのデータ型を定義することができます。

特性カーブおよびマップ：

コンポーネント内で定義された特性カーブおよびマップは、ESDL の場合と同様、それぞれ 3 つのサブルーチンによって評価されます。

122 ページの「1 次元テーブル」で説明されている LLpr を、ここで再び特性カーブの例として使用します。

```
0.0     1000.0  2000.0  3000.0  4000.0  5000.0  6000.0
0.0     0.8     1.1     1.5     1.8     2.0     2.2
```

特性カーブの評価には、通常はサブルーチン CharTable1_getAt_real64_real64(charline, index) が適しています。ここでは線形補間は以下のように行われます。

```
tmpVal = CharTable1_getAt_real64_real64(LLpr,3000);
// tmpVal に 1.5 を代入する
tmpVal = CharTable1_getAt_real64_real64(LLpr,2280);
// 2280 の補間係数を求め、
// その補間係数に対応する値 1.212 を求めて
// それを tmpVal に代入する
tmpVal = CharTable1_getAt_real64_real64(LLpr,9000);
// 9000 の補間係数を求め、
// その補間係数に対応する値 2.2 を求めて
// それを tmpVal に代入する
```

場合によっては、ブレイクポイントの検索と補間のステップを別々に行った方がより効率的になる場合があります。この場合、サブルーチン CharTable1_search_real64(charline, index) および CharTable1_interpol_real64_real64(charline) が使用されます。

```
CharTable1_search_real64(LLpr, 1000);
// 入力座標値として 1000 をセットする
tmpVal = CharTable1_interpol_real64_real64(LLpr);
// tmpVal に 0.8 を代入する
CharTable1_search_real64(LLpr, 2780);
// 2780 の補間係数を求める
tmpVal = CharTable1_search_real64(LLpr)
// 2780 の補間係数に対応する値 1.412 を求めて
// それを tmpVal に代入する
```

さらに、123 ページの「2 次元テーブル」で説明されている LLpr2 を、ここで再び特性マップの例として使用します。

```
y \ x     0.0       1.0       8.0       15.0
1.0       -5.0      -3.0      0.0       1.0
3.0       0.0       1.0       4.0       6.0
5.0       8.0       5.0       4.0       4.0
```

特性マップの評価には、通常、サブルーチン CharTable2_getAt_real64_real64_real64(charline, indX, indY) が適しています。ここでは線形補間は以下のように行われます。

```
tmpVal = CharTable2_getAt_real64_real64_real64(LLpr2,8,5);
// tmpVal に 4.0 を代入する
tmpVal = CharTable2_getAt_real64_real64_real64(LLpr2,2,2);
// x=2、y=2 についての補間係数を求め、
// その補間係数によって補間された値 -0.571 を
// それを tmpVal に代入する
tmpVal =
      CharTable2_getAt_real64_real64_real64(LLpr2,20,10);
// x=20、y=10 の補間係数を求め、
// その補間係数によって補外された値 5.0 を求めて
// それを tmpVal に代入する
```

ここでも、ブレイクポイントの検索と補間のステップを別々に行った方がより効率的になる場合があります。この場合、サブルーチン CharTable2_search_real64_real64(charline, indX, indY) および CharTable2_interpol_real64_real64_real64(charmap) が使用されます。

```
CharTable2_search_real64_real64(LLpr2, 1,3);
// x と y の座標値として、1 と 3 をセットする
tmpVal = CharTable2_interpol_real64_real64_real64(LLpr2);
// tmpVal に 1.0 を代入する
CharTable2_search_real64_real64(LLpr2,4,4);
// x=4、y=4 の補間係数を求める
tmpVal = CharTable2_interpol_real64_real64_real64(LLpr2)
// 補間係数によって補間された値 3.143 を
// tmpVal に代入する
```

インプリメンテーションの種類ごとに対応するコードのバリエーションがあるので、ユーザーはインプリメンテーションごとのローカル変数とそのデータ型を定義することができます。

7.1.3 ヘッダ

メソッド以外にも、マクロやインクルードされるファイルのヘッダも C コードで定義することができます。このヘッダのスコープはローカルで、そのコンポーネントに限られています。したがって、余分なヘッダファイルは生成されず、ヘッダの定義は生成される C コードファイル内にコピーされます。

7.2 外部ソースコード

外部の C コードソースファイルをインポートすることにより、既存の C コードを ASCET システムに統合することができます。このためには、ヘッダファイルを持つ 1 つの C コードファイルをコンポーネントの各コードバリエーションに添付します。添付できる C コードファイルは、標準的な C 関数が記述され、またヘッダファイルはそれに対応する関数の宣言と構造体定義が含まれているものです。これらのファイルに定義されている関数を、標準の C の規約に従って呼び出すことができます。コンポーネントのメソッドやプロセスと、統合された C コード内の関数との間でポインタを受け渡ししたり、定義済みの構造体を共用したりすることができます。

C コードファイルを使用するためのもう 1 つの方法として、オブジェクトファイルとそれに対応するヘッダファイルをコンポーネントに添付することができます。コンポーネント自体のヘッダファイルと同様に、統合されたソースのヘッダファイルのスコープもローカルなので、このヘッダファイルの内容も、生成される C コード内にコピーされます。

添付された C ファイルは個別にコンパイルされ、他の（生成されコンパイルされた）C ソースファイルとリンクされます。その結果、コンパイル済みのユニットは 1 つしか存在しないため、コードやインクルードされるデータを同じコンポーネントの複数のインスタンスで共用する場合は、すべてのインスタンスが同じコンパイル済みユニットを共用します。

注記

コンポーネントのコードに添付された C ファイルのデータは、そのコンポーネントの複数のインスタンス間で共用されるので、各インスタンスごとにはインスタンス化されません。

さらに、C コード内に `include` 文を使用することもできます。ただし、インクルードファイルはデータベース内ではなくファイルシステム内に格納されるものであるため、`include` 文には、これらのインクルードファイルのファイルパスを指定する必要があります。このように、`include` 文が使用された C コードはデータベース内のアイテムだけでなく、インクルードファイルが保存されている

ディスクのファイル構造にも依存することになります。これらのファイルは、ファイルのパスが変わると ASCET システムに認識されなくなる恐れがあるので、PC 間でデータを交換する際には注意が必要です。

注記

`include` 文でファイルをインクルードする場合、インクルードされるファイルが常に正しく参照されるように注意してください。

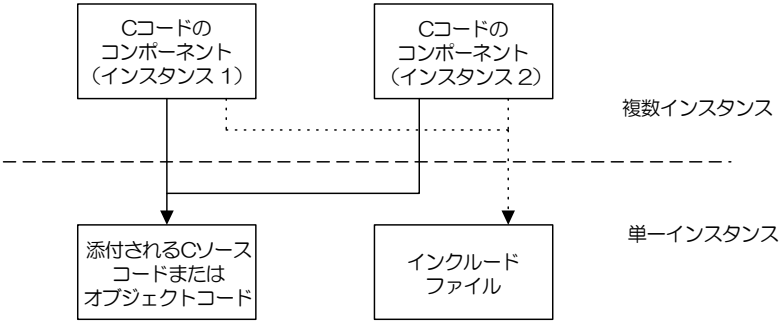


図 7-2　外部ソースコードの使用

7.3　プログラミングモデルインターフェース

旧バージョンの ASCET では、エスケープ文字‘@’の付いたクラスとメソッドの名前は C コードでしか使用できませんでした。このエスケープ文字によってプログラミングモデルインターフェース（PMI：**P**rogramming **M**odel **I**nterface）が起動するためです。

しかし ASCET 4.x および 5.x ではクラス名とメソッド名は自動的に認識されるようになったため、このエスケープ文字は不要になり、デフォルトで PMI が使用されるようになりました（『ASCET ユーザーズガイド』の「コード生成オプション」に関する説明を参照してください）。このため、C コードでモデリングする際にはこの文字は使わないでください。ただし、旧バージョンとの互換性を維持するために、コード生成オプションを変更することによりエスケープ文字を使用することができます。

7.4　アクセスマクロ

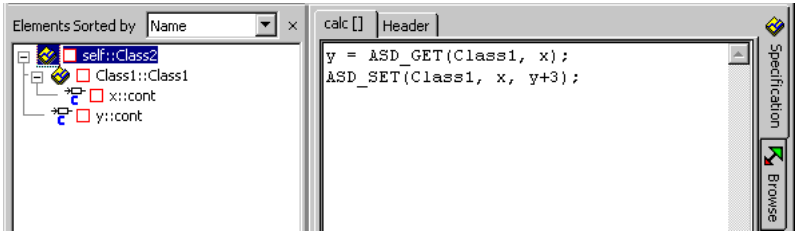
ESDL のアクセスメソッドと同様に、C コードコンポーネント用にも以下のようなアクセスマクロが用意されています。これらのマクロにより、定義済みの演算を利用できます。

直接アクセス

以下のマクロで、C コードコンポーネントに組み込まれたクラスのエレメントに直接アクセスすることができます。

```
ASD_GET(receiver, variable);
ASD_SET(receiver, variable, value);
```

例：



配列長

以下のマクロで配列の現在の長さを取得できます。

```
ASD_LENGTH (receiver);
```

例：

```
z = ASD_LENGTH(array);
```

リソースアクセス

以下のマクロでリソースの占有と解放を行えます。

```
ASD_RESERVE(resource);
ASD_RELEASE(resource);
```

プライベートメソッドへのアクセス

以下のマクロでプライベートメソッドにアクセスできます。

```
self.
```

例：

```
y = self.method_private(x);
```

ASCET の配列への外部 C コードからのアクセス

以下のマクロで、配列のアクセス処理が、ASCET の内部表記から標準の C コード表記に変換されます。

```
ASD_USE_ARRAY_EXTERNAL(array)
```

このマクロは以下の表記と同等になります。

```
&array[0]
```

例：

```
y = c_function(ASD_USE_ARRAY_EXTERNAL(array));
```

8 連続系

ASCET の包括的なモデリング機能によって、コントローラソフトウェアのファンクション開発や制御ユニットのシミュレーションに使用される離散系のモデリングが可能です。これに対し、制御ユニットで処理される制御システムは、微分方程式で定義される連続物理系です。

連続系の例としては、自動車の動力伝達系路やホイールサスペンション（機械系）、シリンダ室の燃焼プロセス（熱力学系）、自動車のブレーキ回路（油圧系あるいは気圧系）、カーバッテリ（電気あるいは電気機械系）などが挙げられます。また、アクチュエータの機械的構造を局所的な電子制御機構に接続したり、物理信号をインテリジェントセンサで電子工学的に処理するメカトロニクスシステムもますます増えてきています。

ASCET では、いわゆる「CT ブロック」を用いてこのような連続系のモデルを設計したりシミュレートしたりすることができます。CT は Continuous Time（連続時間）の略で、疑似連続的な時間増分でモデリングされ計算されるエレメントのことです。ASCET での連続系モデリングは、連続系を設計する際の標準的な記述形式である、ステートスペース表記をベースとしています。この方法では、CT 基本ブロックを非線形 1 次常微分方程式と非線型出力方程式で定義することができます。は、これらの微分方程式を効率的に解くための、リアルタイム積分メソッドを提供します。

連続系モデルは、モジュール式の階層ブロックを用いて構成することができます。連続系モデルを ASCET の制御記述を使用して組み合わせ、いわゆるハイブリッドプロジェクトと呼ばれる結合モデルを構築することができます。これらのハイブリッドプロジェクトを用いて、定義された制御仕様を、その実際の制御対象である技術プロセスのモデルを対象にしてテストすることができます。

モデルとシミュレーション実験とは厳密に区別されています。モデルはモジュール式かつ階層的なシステム記述で定義されているのに対し、実験には選択されたデータセット、積分アルゴリズム、および選択された表示設定（パラメータの入力メソッドなど）が含まれています。この結果、正確で再利用可能なモデルと、実験における高度な柔軟性が得られます。実験レベルでは、各モデル変数を柔軟に変更したり読みとることができます。選択された積分ステップ幅と積分アルゴリズムについて、時間を割いてモデルや現在の実験をコンパイルし直さなくても、シミュレーション中に変更することができます。

8.1 連続系モデルの構成

以下の項では、基本ブロック、構造ブロック、およびグラフィック階層を使用してモデルを構成するためのさまざまなオプションについて説明します。

8.1.1 基本ブロックと構造ブロックを用いたモデリング

連続系のモデルは、モジュールと階層を用いて構成することができます。基本的な要素は連続系基本ブロックつまり CT 基本ブロックで、そこには部分的なモデルが、ESDL あるいは C[1] といった高水準言語により、微分方程式、代数式、公式および代入の形で定義されています。

連続系ブロック（CT ブロック）はさまざまなディメンション、スコープおよびデータ型の入力、出力、パラメータ、離散および連続ステートで構成されます。さらに、連続系や離散系の方程式、出力方程式、初期化や終了のシーケンスもサポートします。ステートイベント、ソフトウェアイベントおよびハードウェアイベント（割り込み）を扱うこともできます。

ブロックダイアグラムエディタ（BDE）を用いて、さらに複雑な連続系モデルを組み合わせて CT 構造ブロックを構築することができます。ブロックダイアグラムエディタを使用すれば、複数の CT 基本ブロックや CT 構造ブロックを作成し、組み合わせることができます。2 つの CT 基本ブロックで構成される単純な CT 構造ブロックを図 8-1 に示します。

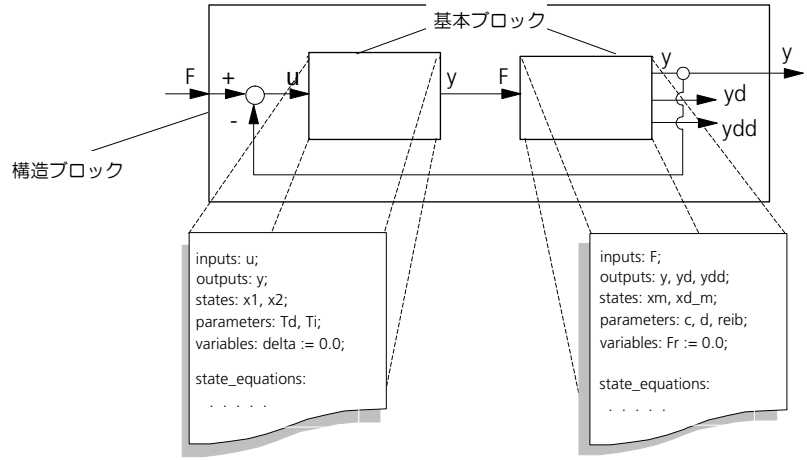


図 8-1　2 つの CT 基本ブロックで構成される CT 構造ブロックの例

[1] C は、どうしても避けられないような例外的な場合に限り使用してください。ASCET は、ESDL についてのみ自動検証（セマンティックスチェック、実行順序決定）を行います。

さらに、複数の CT 構造ブロックと CT 基本ブロックを組み合わせて、新たに 1 つの CT 構造ブロックにすることができます。図 8-2 のような CT 構造が可能です。

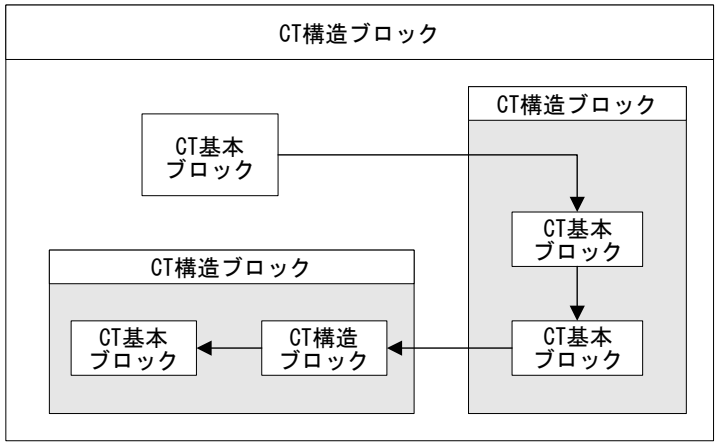


図 8-2　CT 構造ブロックを用いたモデリング

CT ブロックの適切な実行順序は自動的に決まります。CT 基本ブロックや CT 構造ブロックを標準の ASCET 構造と組み合わせて、ハイブリッドプロジェクトを構築することができます。

CT 基本ブロックは、ブレーキ、ホイールなどの小さい物理コンポーネントの定義に用います。CT 構造ブロックは、動力伝導機構や車両全体のモデルなど、複合的な実体を定義するのに役立ちます。CT 基本ブロックと CT 構造ブロックは個別にデータベースに格納されるので、他のモデルの定義にも利用することができ、モデルライブラリを容易に構築することができます。ブロックや構造に加えた変更は、1 つのデータベース内のすべてのモデルに自動的に反映されます。そのため、基本エレメントのメンテナンスを 1 カ所で行うだけで、その修正内容を同じデータベース内のすべてのモデルに自動的に反映させることができます。もちろん、基本エレメントの互換性は確実に維持する必要があります。

グラフィック階層を用いるモデリング

関連性のある複数の CT ブロックを 1 つのグラフィック階層（図 8-3 を参照してください）に結合して、複数の CT 基本ブロックや CT 構造ブロックからなる CT 構造ブロックを一層明快に設計することができます。グラフィック階層と CT 構造を組み合わせて、新しい階層構造を構築することもできます。グラフィック階層を採用しても、処理の順序には影響はありません。ブロックダイアグラムエディタでは、グラフィック階層は二重線のフレームで示されます。

グラフィック階層は、CT ブロック同士の結合性が重要で、1 つの積分ステップ内で固定の計算シーケンスを行う必要があるような場合に、特に用いられます。グラフィック階層を用いて、CT 構造ブロックに起因する代数ループ（187 ページの「代数ループ」、および 191 ページの「グラフィック階層と CT 構造ブロックの違い」という項を参照してください）を回避することができます。また、自動シー

ケンシングにより適切な実行順序が保証されます。グラフィック階層を 1 階層ずつ保存することはできないので、全グラフィック階層を上位の構造ブロックと共に格納します。

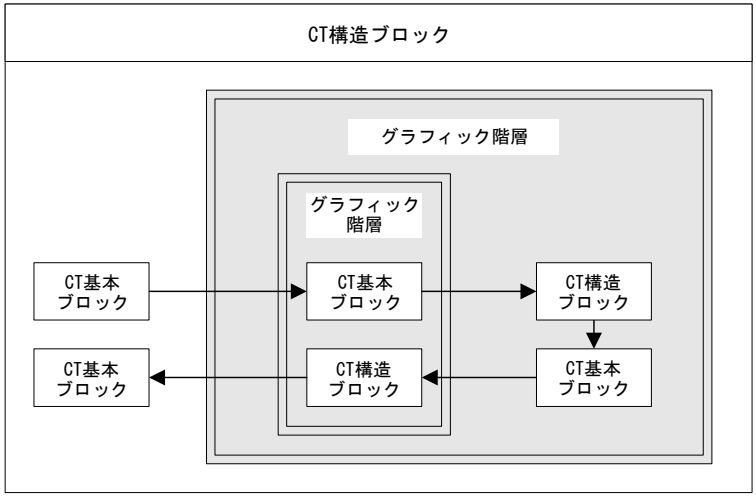


図 8-3　グラフィック階層

実験

基本ブロックや構造ブロックを、シミュレーション実験で評価することができます。実験では、積分メソッド、モデルへの入力データ、および結果の表示形式が選択され指定されます。1 つの（部分的）モデルについて、複数の実験設定を格納することもできます。

プロジェクトとハイブリッドプロジェクト

リアルタイム実験はプロジェクト内に定義されます。プロジェクトには基本ブロックと構造ブロックの両方を使用することができます。さらにプロジェクト内においてのみ、積分メソッドとそのステップ幅を、各基本ブロックや構造ブロックごとに個別に指定することができます。これにより、プロセッサ能力が限られている場合、ダイナミクスの高いモデル部分に、ダイナミクスの低いモデル部分よりも多くの CPU 時間を割り当てることが可能となります。

1 つのモデル内でコントローラのモデルと制御システムのモデルを結合する場合、ハイブリッドプロジェクト、つまり、CT ブロックと標準の ASCET コンポーネントの両方を内包するプロジェクトを定義することができます。ハイブリッドプロジェクトを使えば、1 つのモデル内で制御システムと制御ユニットのシミュレーション（ハイブリッドシミュレーション）を行うことができます。

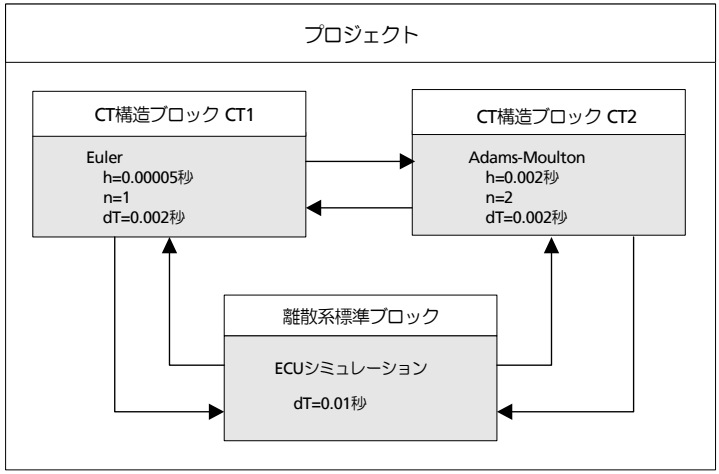


図 8-4　プロジェクト

個々の CT ブロックと個々のコントローラモジュールの間の通信は、ブロックダイアグラムエディタで入力と出力を明示的に接続することで実現します（プロジェクトについての詳細は、15 ページの「プロジェクト」という章を参照してください）。

実験はリアルタイムシミュレーションハードウェアを用いてオンラインで、あるいは特殊なハードウェアを使用したり組み込む必要がない実験であれば PC を用いてオフラインで行うことができます。

8.2　微分方程式の解法 — 積分のアルゴリズム

連続系モデルや、時間間隔が短い場合の非線形方程式は複雑なため、解析的方法で解くことは一般に不可能です。そこで、数値積分アルゴリズムを用いて連立微分方程式を解く必要があります。

CT 構造内で CT ブロックだけをシミュレートする場合には、ASCET はグローバル積分アルゴリズムを使用します。離散型コントローラモデルとの組み合わせは、プロジェクトレベルでのみ可能です（ハイブリッドプロジェクトでの結合モデリング）。プロジェクトでは、さまざまな積分メソッドを用いる複数の CT 構造からなるモデリングも行うことができます。

モデリングとシミュレーションにおける高度の柔軟性と短い反復周期を保証するために、積分メソッドの設定、つまり、実際の積分メソッドとその積分ステップ幅を、実験中に対話形式で選択して変更することができます。

あらゆるタイプのモデルに適するような理想的な積分メソッドは存在しません。個々のアルゴリズムの速度と精度は、非線形性、不連続性、および力学的性状などのモデル特性により異なります。ステップ幅はモデルと積分メソッドに最適になるように調整されるので、各メソッドの速度に関して一般論を述べることはできませんが、適切な積分メソッドを選択するためのガイドラインを以下に紹介します。詳細は、以下の文献などをご覧ください。

> ACM Transitions on Mathematical Software 第 17 巻『A Decision Tree for the Numerical Solution of Initial Value Ordinary Differential Equations』(Addison, C. A.、Enright, W. H. 他著、1991 年 3 月 1 日出版）の「Continuous Time Integration Algorithms」という章

ASCET では、以下の積分メソッドが提供されています。

- Euler
- Mulstep 2（LABCAR-DEVELOPER では Adams-Bashforth と呼ばれています）
- Heun
- Adams-Moulton 2
- Runge-Kutta 4

また、さらに複雑で細密な微分方程式の場合、より精度の高い計算が必要となるため、以下のような、可変ステップによる累次積分メソッドも用意されています。

- Dormand/Prince RK5
- Calvo 6(5)
- Dormand/Prince RK8
- Implicit RK2
- Implicit RK4
- Implicit Gear 1
- Implicit Gear 2

これらの積分メソッドは、要求された精度を実現するため、ステップ幅を調整します。そのため、リアルタイム計算には使用できません。

技術的な理由から、Implicit 積分メソッドは、新しいバージョンの Borland コンパイラと Microsoft コンパイラでしか 使用できず、ASCET 製品に含まれる Borland C 4.5 では使用できません。これらのメソッドは GNU Scientific Library に含まれています。また以前のバージョンの ASCET 製品に含まれていた Gear 4 メソッドは、サポートされなくなりました。

8.2.1 積分メソッドの概要

以下のステート形式の微分方程式を想定します。

`x'(t) = f(x,t);` ただし `x(t=0) = x0`

この方程式に各積分メソッドを使用した場合の特徴を、以下のような項目についてまとめます。

- h^p に比例する大域的誤差指数 P：ここで、h は積分ステップ幅です。
- 積分ステップあたりの関数評価数：積分ステップごとに、ローカル変数がリセットされ、`nondirectOutputs`、`directOutputs`、`derivatives` というメソッドが実行されます。積分ステップ当たりの関数評価数と積分ステップ幅を用いて、メソッドの処理速度を見積ることができます。
- シングルステップ／マルチステップメソッド（SSM/MSM）：シングルステップメソッドでは一番最後の推定値だけを次のステップのために使用しますが、マルチステップメソッドでは最後の n 個の推定値を考慮します。
- 予測子修正子法（P-C）：最初に 1 つの積分メソッドを使用して推定値を算出し、第 2 のメソッドを用いてその推定値を修正します。
- 固定あるいは可変のステップ幅

下の表は、固定ステップ幅の積分メソッドの特徴をまとめたものです（MSM については、関数が実行されるタイミングや、ブレークポイントが考慮に入れられるタイミングを、かっこ内に示しました）。

積分メソッド	誤差指数 P	積分ステップ当たりの関数評価数	SSM/MSM	P-C	ステップ幅
Euler	1	1 (t)	SSM	no	固定
Mulstep 2	2	1 (t)	MSM (t-h, t)	no	固定
Heun	2	2 (t, t+h)	SSM	yes	固定
Adams-Moulton	2	2 (t, t+h)	MSM (t-h, t)	yes	固定
Runge-Kutta 4	4	4 (t, t+h/2, t+h/2, t+h)	SSM	no	固定

積分メソッドをリアルタイムに確実に実行できるようにするために、各メソッドは、積分ステップ当たりの関数評価数を比較的少なくし、かつ誤差指数を低くして実装されます。

Euler

Euler 積分メソッドは、使用できる積分メソッドの中で最も単純なメソッドです。積分ステップ当たり関数評価を 1 回のみ行うシングルステップメソッドで、サイクルタイムも最小で比較的高速なため、リアルタイムシミュレーションに特に向いています。

数式
`x(t+h)=x(t)+h*f(x,t)`

このメソッドは安定範囲は広いですが、誤差が大きく、一般的に同じステップ幅の他のメソッドよりも高い（最低の指数を持つ）のが難点です。

Mulstep

Mulstep 積分メソッドはマルチステップメソッドで、急激に変化する固有値を伴わないモデルで使用されます。積分ステップごとに 1 つの関数評価しか行われないので、1 積分ステップのサイクルタイムは Euler メソッドよりもわずかに長いだけですが、誤差指数は 2 です。

数式

```
x(t+h)=x(t)+h(3/2*f(x,t)-1/2f(x,t-h))
```

Heun

Heun 積分メソッドは、あまり多彩な固有値を伴わないモデルに使用されます。サイクルタイムは Euler メソッドの 2 倍です。

数式

予測子：`x(t+h)=x(t)+h*f(x,t) (Euler)`
修正子：`x(t+h)=x(t)+h/2*(f(x,t)+f(x,t+h))`

Adams-Moulton

Adams-Moulton 積分メソッドも、あまり多彩な固有値を伴わないモデルに適しています。先に紹介したアルゴリズムとは対照的に、モデルはなめらかな性状を呈していなければなりません。Adams-Moulton アルゴリズムと Heun アルゴリズムのサイクルタイムはほぼ同じです。

数式

予測子：`x(t+h)=x(t)+h/2(3f(x,t)-f(x,t-h)) (Adams-Bashforth)`
修正子：`x(t+h)=x(t)+h/2(f(x,t)+f(x,t+h))`

Runge-Kutta 4

Runge-Kutta 積分メソッドは、極端に変化する固有値を伴わないモデルに最適で、そのようなモデルに対する耐性が大きいメソッドです。さまざまな積分メソッドの中で、所要時間は最長ですが、同じステップ幅で比較した場合の精度は最高です。そのため、ステップ幅をかなり大きくすることができます。

数式

```
x(t+h)=x(t)+h/6(K1+2K2+2K3+K4)
```

以下の条件に基づく：

```
K1 = f(x,t)
K2 = f(x + K1*h/2, t + h/2)
K3 = f(x + K2*h/2, t + h/2)
K4 = f(x + K3*h, t + h)
```

可変ステップ幅の積分メソッド

非常に高い計算精度が要求される実験においては、一般的にステップ幅を短くすることによって計算時間を増大させますが、細密な積分方程式を用いたモデルの場合、固定ステップの積分メソッドでは処理しきれないことがあります。そのような場合は、ステップ幅がターゲットの許容誤差に応じて調整される「可変ステップ幅」の積分メソッドが用いることができます。ただしステップ幅の減少は、モデル内で必要な箇所についてのみ行われます。これは、ステップ幅が変わって処理時間が変化すると、積分メソッドのリアルタイム性が失われてしまうためです。

不適切なパラメータ設定によって適切な計算精度が得られない場合、ASCET のモニタウィンドウにワーニングメッセージが表示されます。このような状況が発生するのは、Max. iterations パラメータ（最大ステップ数）の値が小さすぎるか、または Minimum h パラメータ（最小ステップ幅）の値が大きすぎる場合です。

9 連続系基本ブロック

連続系基本ブロック（CT 基本ブロック）は一般に、さまざまなモデリングのシナリオにおいて用いることのできる、小規模で独立的な物理コンポーネントの定義に用いられます。基本ブロックの定義は、CT ブロックエディタを用いて行います。ブロックインターフェースは対話形式で定義され、物理コンポーネントのダイナミクスは微分方程式や代数式で定義されます。

9.1 基本事項

連続系基本ブロックの定義は、C コードエディタまたは ESDL エディタで行いますが、各エディタではわずかに方法が異なります。ブロックの内部、つまり微分方程式や代数式、および制御構造は、定義済みメソッド内に記述されます。連続系を適切にモデリングするための正しい実行順序は、自動的に導出されます（シーケンシング）。あらかじめ定義されているメソッド構成をユーザーが変更することはできません。

基本ブロックは、モデルを非線形一次常微分方程式（ODE）と非線形出力方程式で定義するために用いられます。より高次の系を定義するためには、複数の 1 次微分方程式に変換する必要があります。下の表は、2 次系から状態空間表記への変換を示しています。

1 つの 2 次微分方程式	2 つの 1 次微分方程式
$T^{2*}x'' + 2.0*d*T*x' + x = K*in;$	$x' = xp;$ $xp' = (K*in - (2.0*d*T*xp) - x) / T^2;$

表 9-1 2 次微分方程式を分解する

方程式は ESDL か C で記述することができます。ESDL で作成すれば、ターゲットに依存しない定義が可能なうえ、高度のセマンティックスチェックを行うこともできます。C を使用する場合、C プログラミング言語の全機能を使用することができますが、セマンティックス分析ができないという難点もあります。ANSI C を使用すれば、たいていはターゲットに依存しないモデリングが可能ですが、特殊なハードウェアの制御などに特殊な言語表記が用いられている場合には、全く依存しないとは言いきれません。しかも C では、ブロックの性状を「直接」であるか「間接」であるかを定義する必要があります（188 ページの「直接出力と間接出力」を参照してください）。

9.2 使用できるエレメントとメソッド

連続系基本ブロックでは、一部のエレメントが離散型のモジュールやクラスのものとは異なります。以下のエレメントが存在します。

- 入力
- 出力
- 連続ステート

- 離散ステート
- ステップローカル変数
- パラメータ
- 依存パラメータ
- 定数
- 1D/2D テーブルパラメータ

各タイプのエレメントにはさまざまなディメンション、スコープ、およびデータ型を持たせることができます（174 ページの「ブロックインターフェース」という項を参照してください）。下図は、さまざまなデータ型（とそれらに対応するアイコン）およびこれらのデータを扱うことのできるメソッドを示しています。

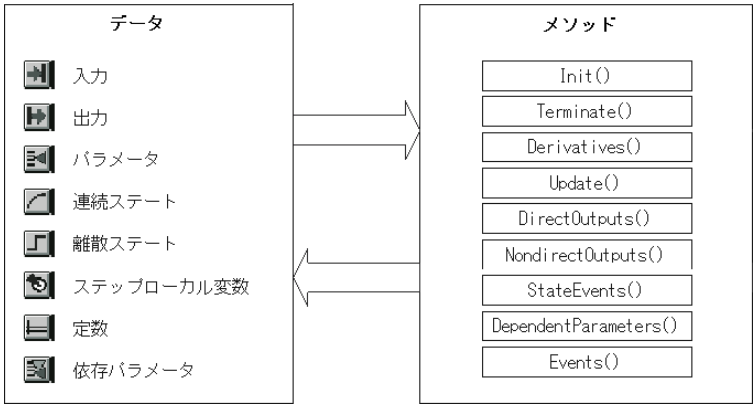


図 9-1　各種データ型とメソッド

9.2.1　連続系基本ブロックを用いるモデリング

連続系基本ブロック内には、モデリング対象となるシステムの内部を ESDL モデル記述言語で、あるいは C で直接、定義することができます。ターゲットに依存しない ESDL モデリング言語には高度のセマンティックス検証機能があり、モデルを確実に正しく定義することができるので、C で直接モデリングするのは、ターゲットに依存するリアルタイムブロックだけにすることをお勧めします。

ブロックの動作は、固定の枠組み、つまり固定数のメソッドで定義されます。各メソッドには、たとえば、微分や出力値の計算など固有の目的があります。標準の ASCET モデルとは対照的に、実行順序は固定（176 ページの「実行順序」という項を参照してください）で、メソッドは自動的にスケジューリングされます。

9.3 ブロックインターフェース

連続系基本ブロックのモデリングに用いられるエレメント（インターフェース、ストレージエレメント）は、離散型のモジュールやクラスに用いられるものとは少し異なります。使用できるエレメントのタイプについて、以下に説明します。

入力：ブロックへの入力は入力（Inputs）を用いて定義する必要があります。各評価ステップで、すべての入力変数の値が読みとられます。

出力：ブロックからの出力は出力（Outputs）を用いて定義する必要があります。各評価ステップで、すべての出力変数の値が更新されます。

連続ステート：常微分方程式の定義には、ステート変数が必要です。各ステート変数は「ストレージエレメント」として機能します。一例として、移動質量点の距離と速度が挙げられます。連続ステート変数は、微分演算子 `ddt` によってのみ使用されます。

離散ステート：離散ステート変数はストレージエレメントです。たとえばカウンタなどの変数の値を、1 つの計算ステップから次のステップまでの間保持するために用いることができます。離散ステート変数は離散型のクラスやモジュール内の変数と同等です。微分演算子 `ddt` が離散ステート変数を扱うことはできません。

ステップローカル変数：ステップローカル変数は、評価ステップの計算中の中間値を格納するために使用されます。これらの変数はすべてのブロックメソッドで見ることができます。ステップローカル変数の値は、反復されるステップの冒頭で毎回初期化されるので、1 評価周期内でだけ有効です。この値を別のメソッドで評価する必要がある場合には、必ず値が設定されてから読みとられるように、メソッドの実行順序をよく考えて決める必要があります。

パラメータ：パラメータは物理モデルの作成に使用されます。普通、パラメータは実際のシステムの特性プロパティ（質量、長さ、減衰定数など）に対応します。パラメータを定義するという効率的な手段で、包括的なモデルライブラリを体系的に構築することができます。シミュレーション中に実験環境でパラメータの値をいろいろ変化させることが可能です（適合エディタを使用します）。

依存パラメータ：あるパラメータが別のパラメータ（たとえば、別の座標系に定義されているパラメータ）に依存する場合、前者のパラメータは後者のパラメータが変化した場合だけ計算し直せばよいことになります。このような性状のパラメータを依存パラメータとして定義することができます。依存パラメータは、変化があった場合に限り、`dependentParameters` メソッドで非同期的に計算されます。

例：`m_vehicle = m_empty + m_payload`

実験中に荷重が変化すると、`dependentParameters` メソッドでは車重が再計算されます。

定数：重力定数など、実験中に変化しない、システム全体で用いる値は定数として定義されます。

ディメンション、スコープ、およびデータ型：エレメントの各タイプごとに、さまざまなディメンション、スコープ、データ型を定義することができます。定義可能な組み合わせを以下に示します。

組み合わせ / エレメント	ディメンション			スコープ		データタイプ			
	スカラ	配列	レコード	ローカル	グローバル	logic	sdisc	udisc	cont
入力	○	○	○	○		○	○	○	○
出力	○	○	○	○		○	○	○	○
離散ステート	○	○		○		○	○	○	○
連続ステート	○	○		○					○
ステップローカル変数	○	○		○		○	○	○	○
パラメータ	○	○		○	○	○	○	○	○
依存パラメータ	○	○		○		○	○	○	○
定数	○	○		○	○	○	○	○	○

図 9-2　ディメンション、スコープ、データ型

9.4　ブロックメソッド

CT ブロックで使用できるメソッドの型と数はあらかじめ定義されているので、ユーザーが変更することはできません。各メソッドには、たとえば、微分や出力値の計算など固有の目的があります。メソッドは、固定的な順序で自動的に実行されます。各 CT 基本ブロック内に必ずしもすべてのメソッドを使用しなくてもかまいません。

CT 基本ブロックでは、以下のメソッドを使用することができます。

`init()`：`init()` メソッドが実行されるのは、実験の開始時と再開時だけです。このメソッドを使用してブロックを初期化するためのコードを定義し、たとえば、モデルの起動時動作を定義したり、ステート変数を初期化したりできます（例、`resetContinuousState(x,5.3)`）。計算文から導かれる初期値は、`init()` メソッドを用いて明示的に設定する必要があります

`terminate()`：`terminate()` メソッドは実験の終わりに実行されます。このメソッドを使用して、ブロックを終了するためのコードを定義し、たとえば、システムのシャットダウン処理を定義することができます。

`derivatives()`：常微分方程式（ODE）は、必ず `derivatives()` メソッド内に定義します。たとえば、移動する質点を用いて静的および動的摩擦をシミュレートするモデルのように、シミュレーション中にモデル構成が変わる場合でも、**通常の処理フロー制御構文**（`if(...) then... else`）を用いて構成の変化をコントロールすることができます。

`update()`：`update()` は、外部通信周期 dT の粒度において実行されます。このメソッドを用いて、この粒度においてのみ（また実験環境との通信において）必要な値を算出することができます。

directOutputs()：directOutputs() メソッドには、入力に直接依存する直接通過の出力方程式を定義します。これらの方程式は nondirectOutputs() に依存する可能性のある入力に直接依存するため、このメソッドは nondirectOutputs() の後に実行されます。

nondirectOutputs()：nondirectOutputs() メソッドには、間接通過の（つまり、入力に直接依存しない）出力方程式を定義します。

dependentParameters()：dependentParameters メソッド内には、他のパラメータに依存するパラメータについての方程式を定義します。このメソッドは、シミュレーション実験中にパラメータが変更された場合に限り実行されます（パラメータ変化時の非同期実行）。これにより、処理時間を短縮することができます。

例：`m_vehicle = m_empty + m_payload`

車重は、実験中、荷重が変化した時にのみ、dependentParameters メソッドで再計算されます。

stateEvents()：stateEvents() メソッド内では、ステートあるいは時間に依存する非連続性を定義することができます。このメソッドは、整合積分ステップの終わりに評価されます。離散ステート方程式は、必ず stateEvents() メソッド内に定義します。

events()：events() メソッドは、非同期のソフトウェアおよびハードウェア割り込みの処理に使用されます。このメソッドは時間同期的には実行されず、対応するイベントが発生したときに非同期的に実行されます。

9.5 実行順序

シミュレーションの実行中、CT ブロック内のメソッドはさまざまな周期で実行されます。一般に、以下の 3 種類の周期があります。

- 外部通信周期 dT
- 積分ステップ幅 h
- 内部積分メソッドに依存するステップ幅 h/n

外部通信周期：dT

通信周期はモデルの一環としては定義されず、シミュレーション実行時にのみ選択されます。dT サイクルでは以下の通信が発生します。

- CT ブロックと実験環境の間の通信（例、データ入力や表示）
- ハイブリッドプロジェクト内でのCTブロックとコントローラモジュールの通信
- 複数の積分メソッドを使用するハイブリッドプロジェクト内での複数の CT（構造）ブロック間の通信
- update() メソッドの実行

積分ステップ幅：h

積分ステップ幅はモデルの一環としては定義されず、シミュレーション実行時にのみ選択されます。h サイクルでは、1 つの連続系構造ブロック内の複数の連続系ブロック間で通信が行われます。すべてのブロックで積分ステップが実行された後、`stateEvents()` メソッドが実行されます。

転送される値は数値的に認識され、選択された積分メソッドに応じて異なります。h の値が非常に小さくなければならないような非常に変動の激しいモデルをシミュレートする場合、dT の値を h よりもはるかに大きくすることで、大幅に高速化することができます。

内部積分メソッドに依存するステップ幅：h/n

h サイクル以外に h/n サイクルも、選択された積分メソッドに応じて変わります。たとえば、Euler 積分メソッドではサイクルタイムとして h/1 を用い、Heun 積分メソッドでは h/2 を用います。

h/n サイクルでは、積分の中間ステップが計算されます。h サイクルの場合、通信は連続系構造ブロック内の連続系ブロック間で行われます。積分の中間ステップを外部に転送することはできません。

この周期内では `stateEvents()` メソッドは呼び出されないので、数値的にはこの周期内で非連続性を扱うことはできません。

各ステップ幅の間には、以下の関係があります。

```
dT >= h >= h/n
```

さまざまなメソッド呼び出しの周期を、図 9-3 に示します。

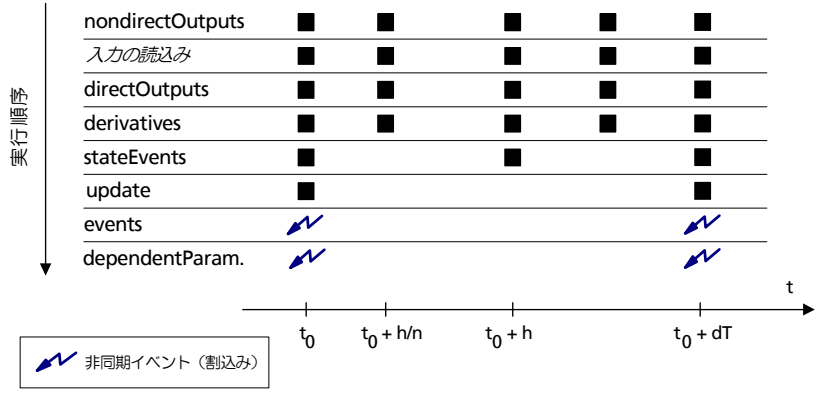


図 9-3　連続系ブロック内のメソッド呼び出しの周期

`events()` および `dependentParameters()` メソッドは、dT サイクル中でないときに明示的な非同期イベントが発生した場合に限り、呼び出されます。

図 9-4 は、シミュレーション開始から終了までのすべてのメソッドの実行順序を示しています。

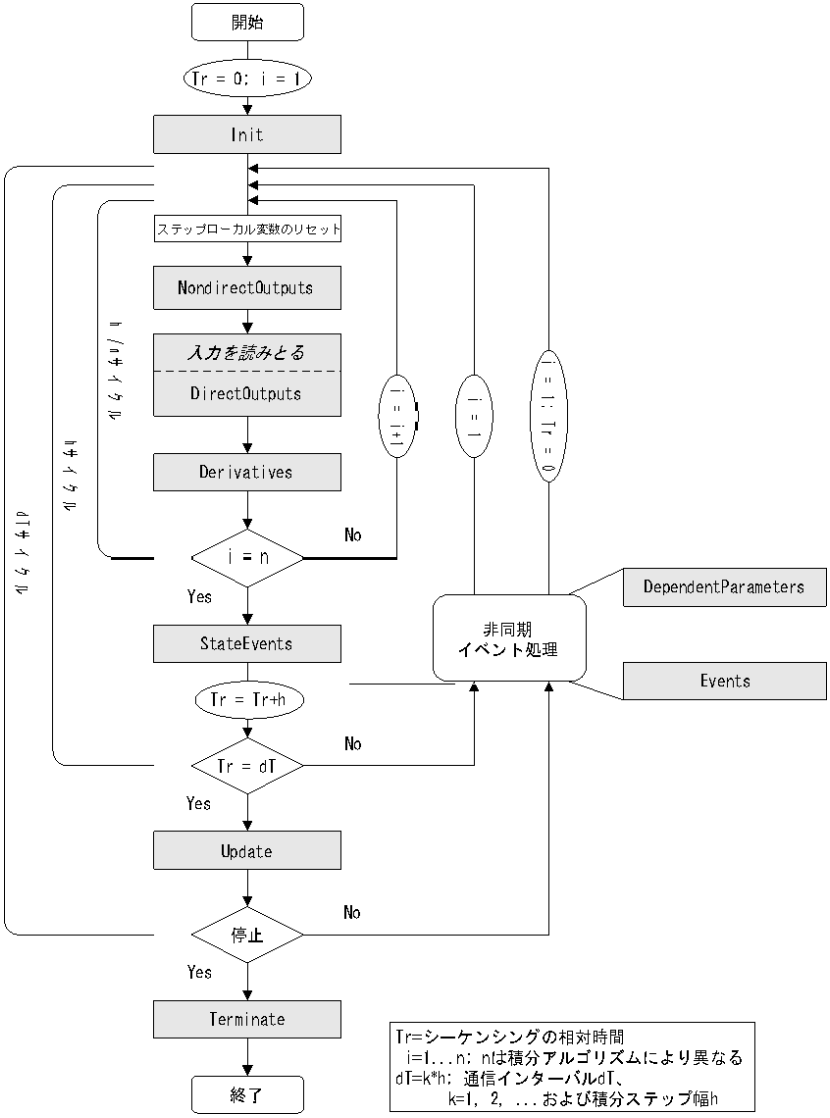


図 9-4　CT ブロック内のメソッドの実行順序

基本ブロックにおけるメソッドの実行順序について、以下の例を使って説明します。

たとえば、`n=1`（Euler）かつ `h=dT` の場合、各タイミング（時間 t）における同期的呼び出しの評価順序は以下のとおりです。

- 時間 t=*dT*：`nondirectOutputs` -（入力の読み込み）- `directOutputs` - `derivatives`
- 時間 t=*dT*：`stateEvents`
- 時間 t=*dT*：`update`

たとえば、*n*=2（Adams-Moulton）かつ *h=dT* のようなやや複雑な積分メソッドの場合、評価順序は以下のとおりです。

- 時間 t=*dT*/2：`nondirectOutputs` -（入力の読み込み）- `directOutputs` - `derivatives`
- 時間 t=*dT*：`nondirectOutputs` -（入力の読み込み）- `directOutputs` - `derivatives`
- 時間 t=*dT*：`stateEvents`
- 時間 t=*dT*：`update`

n=1 かつ *h=dT*/2 の場合、評価順序は以下のとおりです。

- 時間 t=*dT*/2：`nondirectOutputs` -（入力の読み込み）- `directOutputs` - `derivatives`
- 時間 t=*dT*/2：`stateEvents`
- 時間 t=*dT*：`nondirectOutputs` -（入力の読み込み）- `directOutputs` - `derivatives`
- 時間 t=*dT*：`stateEvents`
- 時間 t=*dT*：`update`

連続系基本ブロックを適切に使用するためには、その実行順序を理解して動作を把握することが不可欠です。ESDL をモデリング言語として使用すれば、自動分析が行われるので、複数の CT ブロックを接続する際、確実に一貫性のあるモデリングを行うことができます。直接出力（directOutputs）を伴うブロックの場合、対応する入力には同じ反復周期の値を適用させる必要があるので、実行順序は特に重要です。

9.6 ESDL でのモデリング

連続系基本ブロックの定義には、ESDL のあらゆる言語機能を用いることができます。さらに、セマンティックスチェック機能や、微分方程式を定義するための数々のライブラリ関数も用意されています。これらについて、以下の項で説明します。

9.6.1 ESDL の微分方程式

ESDL では、連続ステート変数が微分演算子 `ddt` をサポートしています。この `ddt` 演算子を用いて、微分方程式を定義することができます。

この例として、連続ステート変数 x および `xp`、入力 `in`、およびパラメータ `d`、`T`、`K` を用いる PT2 システムを挙げることができます。このシステムを数式で記述すると、以下のようになります。

```
x' = xp;
xp' = (K*in - (2.0*d*T*xp) - x) / (T*T);
```

この PT2 システムを ESDL でモデリングする場合、これらの微分は `ddt` メソッドを用いて以下のように定義されます。

```
x.ddt(xp);
xp.ddt( (K*in - (2.0*d*T*x.ddt()) - x) / (T*T) );
```

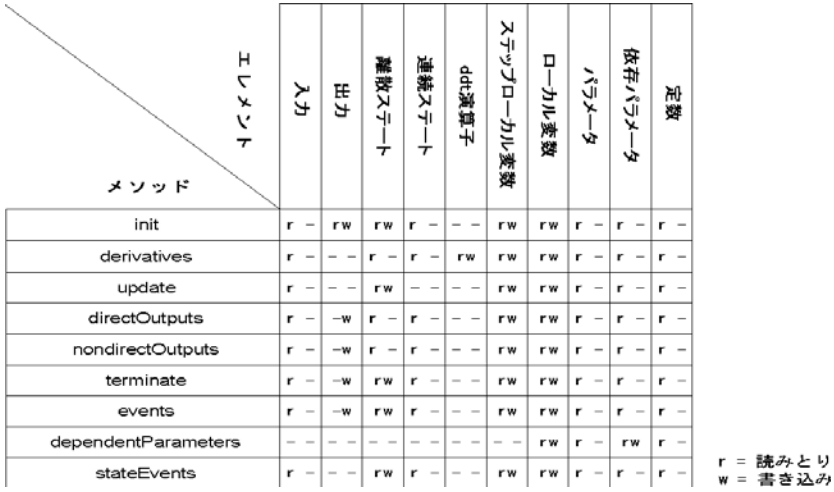
微分方程式の左辺の導関数（つまり、微分メソッドの引数）にアクセスすることはできません。そのようなアクセスが必要な場合には、システムを構成し直す必要があります。

`ddt` 演算子は、`derivatives()` メソッド内でしか使用できません。

9.6.2 ESDL におけるセマンティックスチェック

連続系メソッド内に ESDL を使用する場合、セマンティックスチェックを行うことができます。このときの確認アイテムにより、モデルが基本的な連続系シミュレーションの枠組みに適合していることを確認できます。たとえば、ステート変数の値を直接変更することは許されていません（`resetContinuousState()` 関数を使用して、積分アルゴリズムを内部的にリセットする必要があります）。図

9-5 に、さまざまなエレメントに対するアクセス権についての概要を示します。セマンティックスチェックでは、これらのアクセス権の違反がすべて検知されます。



メソッド ＼ エレメント	入力	出力	離散ステート	連続ステート	ddt演算子	ステップローカル変数	ローカル変数	パラメータ	依存パラメータ	定数
init	r –	rw	rw	r –	– –	rw	rw	r –	r –	r –
derivatives	r –	– –	r –	r –	rw	rw	rw	r –	r –	r –
update	r –	– –	rw	– –	– –	rw	rw	r –	r –	r –
directOutputs	r –	–w	r –	r –	– –	rw	rw	r –	r –	r –
nondirectOutputs	r –	–w	r –	r –	– –	rw	rw	r –	r –	r –
terminate	r –	–w	rw	r –	– –	rw	rw	r –	r –	r –
events	r –	–w	rw	r –	– –	rw	rw	r –	r –	r –
dependentParameters	– –	– –	– –	– –	– –	– –	rw	r –	rw	r –
stateEvents	r –	– –	rw	r –	– –	rw	rw	r –	r –	r –

r = 読みとり
w = 書き込み

図 9-5　各種エレメントに対するアクセス権

微分演算子 `ddt` は初回の導関数だけをサポートします。`nondirectOutputs()` メソッドの出力方程式が分析され、出力が入力に直接依存するかどうかが調べられます。直接依存性が認められた場合には、警告が発せられます。

9.6.3　汎用ライブラリ関数

高度の連続系モデリングを ESDL で行うために、以下のようなさまざまなライブラリ関数がシステムライブラリに用意されています。

- `getTime()`
- `getdT()`
- `getIntegrationStepsize()`
- `resetContinuousState()`
- `resetCTSolver()`

以下に、各ライブラリ関数の使用法について、詳しく説明します。各メソッドからこれらの関数へのアクセスを図 9-6 に示します。

メソッド ＼ ライブラリ関数	getTime	getdT	getIntegrationStepsize	resetContinuousState	resetCTSolver
init	+	+	+	+	+
derivatives	+	+	+	-	-
update	+	+	+	+	+
directOutputs	+	+	+	-	-
nondirectOutputs	+	+	+	-	-
terminate	+	+	+	-	-
events	+	+	+	-	-
dependentParameters	+	+	+	-	-
stateEvents	+	+	+	+	+

+ アクセス可

- アクセス不可

図 9-6　連続系ブロックのメソッドからの関数アクセス

getTime( )：場合によっては、シミュレーションの現在時間が重要になることがあります。オンライン実験の場合、これは実際の経過時間となります。この値を、getTime ライブラリ関数を用いて取得することができます。

```
t = getTime ( );
```

getTime( ) 関数は、どのメソッド内でも使用できます。

getdT( )：getdT ライブラリ関数は、外部通信用に現在のステップ幅を返します。

```
step = getdT ( );
```

getIntegrationStepsize( )：getIntegrationStepsize( ) ライブラリ関数は、現在の積分ステップ幅を返します。

```
h = getIntegrationStepsize ( );
```

resetContinuousState(state, new value)：時間あるいはステートに依存する非連続性をモデリングするには、多くの場合、連続ステート変数をリセットする必要があります。数値の評価が正しく行われるようにするために、積分メソッドを内部的に初期化し直す必要があります。この初期化には、resetContinuousState 関数を使用します。

```
resetContinuousState (x, 0.0 );
```

この場合、ステート x には 0.0 が設定され、必要な場合には、積分メソッドの再初期化が行われます。`resetContinuousState` ライブラリ関数は、`init` および `stateEvents` メソッド内でしか使用できません。`update` メソッド内でも使用できますが、このメソッドは連続ステートへの書き込み権がありません。`resetContinuousState(x,y)` の後には、自動的に `resetCTSolver()` が続きます。

`resetCTSolver( )`：`resetCTSolver` を用いれば、積分メソッドを明示的にリセットすることができます。

```
resetCTSolver ( );
```

`resetCTSolver` ライブラリ関数は、`init`、`update`、および `stateEvents` メソッド内でしか使用できません。`resetContinuousState(x,y)` の後には、自動的に `resetCTSolver()` が続きます。

9.7 C コードでのモデリング

C コードによるモデリングを行うと、C 言語の機能を活用できますが、セマンティックスチェックは行われません。C で記述された連続系基本ブロックはハードウェアに依存する可能性がありますが、C の中でも ANSI-C でプログラミングを行えば、ハードウェアに依存しないモデルを作成することができます。これは、ポインタや C のサブルーチンを使用する場合に必要です。また C の基本ブロックを ESDL の基本ブロックと同じ方法で使用して、ハードウェアに依存するブロックをモデリングすることができます。C の基本ブロックには、**Block Behavior** メニューアイテムから **direct** か **nondirect** を選択して、そのブロックが直接通過（出力が入力に直接依存する）か間接通過かを明示的に定義する必要があります。この定義により、自動的に決まる実行順序が変わります。

注記

C でモデリングすると、モデリングに矛盾がないことを確認するセマンティックスチェックは行われません（ESDL の場合には行われます）。矛盾がないことを、ユーザーが確認する必要があります。
連続系のモデリングでは、コントローラに依存するシステム部分をモデリングする場合や C のポインタやサブルーチンを使用しなければならない場合など、どうしても必要なときだけ C を使用するようにお勧めします。

9.7.1 C の微分方程式

C では、連続ステート変数ごとに内部微分変数が作成されます。この変数の名前はステート変数とプレフィックス `ddt` とで構成されます。

たとえば、連続ステート変数 x および xp について自動的に作成される微分変数は、`ddtx` および `ddtxp` です。これらの変数は、すべてのメソッドで見ることができます。

これを完全に示す例として、連続ステート変数 x および xp、入力 in、およびパラメータ d、T、K を用いる PT2 システムを挙げることができます。

```
x’ = xp;
xp’ = (K*in - (2.0*d*T*xp) - x) / (T*T);
```

この PT2 システムは、以下のように CT ブロック内に C コードで記述することができます。

```
ddtx = xp;
ddtxp = (K*in - (2.0*d*T*ddtx) - x) / (T*T);
```

9.7.2 汎用 C ルーチン

C でのモデリングには、以下の C ルーチンを使用することができます。これらのルーチンを汎用的に使用するためには、現在のブロックの内部データ構造体を、そのルーチンのインターフェースに定義する必要があります。`CTBlock` および `self` というメソッドは、各メソッドで見ることができます。

以下のルーチンを使用することができます。

- `getTime`
- `getdT`
- `getIntegrationStepsize`
- `resetCTSolver`
- `sizeU`
- `sizeY`
- `sizeV`
- `sizeX`
- `sizeXK`

`get` および `reset` ルーチンに加えて提供されている `size` ルーチンを使用すれば、インスタンス変数の数や配列サイズを変更する必要がある場合に、モデルを汎用的な設計にすることができます。

これらの C ルーチンについて、以下に詳しく説明します。セマンティックスチェックは行われず、これらのルーチンには使用上の制約はありません。これらを適切に使用できるかどうかは、ユーザーの責任となります。

`real64 getTime(CTSimExperiment *):`

`getTime` 関数は、現在のシミュレーション時間を返します。

```
t = getTime (CTBlock);
```

`real64 getdT ():`

`getdT` 関数は、現在の外部通信周期を返します。

```
step = getdT ();
```

`real64 getIntegrationStepsize(CTSimExperiment *):`

`getIntegrationStepsize` 関数は、現在の積分ステップ幅を返します。

```
h = getIntegrationStepsize (CTBlock);
```

void resetCTSolver(CTSimExperiment *):

resetCTSolver ルーチンを用いれば、積分アルゴリズムを明示的にリセットすることができます。用例としては、以下のような連続系ステートのリセットを挙げることができます。

```
x = 0.0;
resetCTSolver (CTBlock);
```

1 つあるいは複数の連続系ステートが明示的に設定されている場合、終了時に内部の構造体をリセットする必要があります。resetCTSolver コマンドは、必ず連続系ステートに値が設定されてから実行するようにしてください。

int_32 sizeU (CTSimExperiment *):

sizeU 関数はブロック入力の数を返します。

```
sizeU = sizeU (CTBlock);
```

入力に配列が含まれている場合には、スカラエレメントの総数を返します。レコード、構造体あるいはクラスなど、より複雑な入力は、1 つのエレメントとしてカウントされます。

int_32 sizeY (CTSimExperiment *):

sizeY 関数はブロック出力の数を返します。

```
sizeY = sizeY (CTBlock);
```

出力に配列が含まれている場合には、スカラエレメントの総数を返します。レコード、構造体あるいはクラスなど、より複雑な出力は、1 つのエレメントとしてカウントされます。

int_32 sizeV (CTSimExperiment *):

sizeV 関数はブロックパラメータ（パラメータと依存パラメータ）の数を返します。

```
sizeV = sizeV (CTBlock);
```

一部のパラメータステートが配列の場合には、スカラエレメントの総数を返します。

int_32 sizeX (CTSimExperiment *):

sizeX 関数は連続ステートの数を返します。

```
sizeX = sizeX (CTBlock);
```

一部の連続ステートが配列の場合には、スカラエレメントの総数を返します。

int_32 sizeXK (CTSimExperiment *):

sizeXK 関数は離散ステートの数を返します。

```
nofX = sizeXK (CTBlock);
```

一部の離散ステートが配列の場合には、スカラエレメントの総数を返します。

10 連続系構造ブロックとグラフィック階層

連続系構造ブロック（CT 構造ブロック）を用いれば、他の CT 構造や CT 基本ブロックをグラフィカルなブロックダイアグラムで組み合わせ結びつけて、複雑なモデルを構築することができます。連続系構造ブロック用のブロックダイアグラムエディタ（BDE）は、これまでのものとはやや異なります（165 ページの図 8-2 も参照してください）。BDE では、対応する入力と出力がグラフィックで互いに結びつけられます。

連続系構造ブロックは、固定数のメソッドを伴うブロックダイアグラムとしてモデリングされます。原則として、CT 基本ブロックのメソッドは自動的に定義されるので、BDE で変更することはできません。この機能定義は 1 つのダイアグラムに割り当てられます。適切な実行順序も自動的に決定されるので、それを直接変更することはできません。

『ASCET 入門ガイド』の「連続系をモデリングする」の項に、CT 基本ブロック、そのメソッド、および CT 構造ブロックの使用法から、実験環境におけるシミュレーションまでを示すシンプルな例が説明されています。

10.1 構造ブロックの再利用

CT 構造ブロックは CT 基本ブロックと同様にデータベースに格納されるので、他の CT 構造ブロックに利用することができます。これにより、モジュール式で階層モデル構造用のモデルライブラリを構築することができます。データベース内の CT 基本ブロックあるいは CT 構造ブロックに変更が加わると、その変更はデータベース内のすべてのモデルに反映されるので、CT ブロックのメンテナンスは 1 カ所で行うだけで済みます。

完成したモデル内の CT ブロックを以降のバージョンに置き換えたくない場合には、そのモデルを別のデータベースに格納する必要があります。

10.2 連続系構造ブロックのエレメント

連続系構造ブロックには、基本ブロック内で使用されるすべての変数が必要なわけではありません。使用できるのは以下のエレメントです。

- 入力
- 出力
- グローバルパラメータ
- 定数
- 1D および 2D テーブルパラメータ

各タイプのエレメントにはさまざまなディメンション、スコープ、およびデータ型を持たせることができます。

加算および減算の演算子が用意されていて、その入力の数を個別に選択することができます。

10.3 ブロックインターフェース

以下の項では、連続系構造ブロックで用いることのできるエレメントについて説明します。

入力：ブロックへの入り口は入力（input）により定義します。各評価ステップで、すべての入力変数が読みとられます。

出力：ブロックからの出口は出力（output）により定義します。各評価ステップで、すべての出力変数の値が更新されます。

グローバルパラメータ：グローバルパラメータはモデル全体で見ることのできるパラメータの定義に使用されます。普通、グローバルパラメータは実際のシステムの特性プロパティに対応します。グローバルパラメータを効率的に利用して、複雑さを軽減し、モデルのメンテナンスを容易に行うことができるようになります。

定数：重力定数など、実験中に変化しない値は定数として定義されます。

ディメンション、スコープ、およびデータ型：エレメントのタイプごとに、さまざまなディメンション、スコープ、および型を定義することができます。定義可能な組み合わせは下の表のとおりです。

組み合わせ / エレメント	ディメンション			スコープ		データタイプ			
	スカラ	配列	レコード	ローカル	グローバル	logic	sdisc	udisc	cont
入力	○	○	○	○		○	○	○	○
出力	○	○	○	○		○	○	○	○
グローバルパラメータ	○	○			○	○	○	○	○
定数	○	○		○	○	○	○	○	○

図 10-1 エレメントのディメンション、スコープ、およびデータ型

10.4 演算子

このシステムの理論上、構造ブロックの定義に必要なのは線形演算子だけです。非線形エレメントは基本ブロック内にカプセル化されるので、加算と減算の演算子だけが用意されています。

10.5 代数ループ

以下の連立方程式の、f3 以外の方程式は他の方程式に依存しています。

$x = f_1(z)$

$y = f_2(x)$

$z = f_3(\text{input a})$ （input a は妥当な値と仮定する）

この連立方程式を上から下の順に正しく処理できるようにするには、以下のように並べ替える必要があります。

```
z = f3(input a)
x = f1(z)
y = f2(x)
```

この順序なら、従来の PC プログラムでも簡単に計算することができます。

以下の場合には、代数ループが存在します。

```
y=f1(x);
x=f2(y);
```

つまり、これら 2 つの関数は互いに依存し合っています。x の算出には y が必要で、y の算出には x が必要です。

10.6 直接出力と間接出力

ASCET は、接続されていて相互に直接依存しあう CT ブロック内の CT ブロックやメソッドを、自動的に適切な順序に並べ替えます（自動シーケンシング）。モデル内に代数ループが存在する場合には、ASCET は実行順序を決定する際に適切なエラーメッセージを出力して終了します。このような事態は、たとえば、直接出力を伴う 2 つ以上の CT ブロックがフィードバックループを形成している場合に発生します。

実行順序を自動的に決定しコントロールできるようにするために、出力のプロパティを指定する必要があります。入力に直接依存する出力は、`directOutputs` メソッドで記述つまり定義する必要があります。このような CT 基本ブロックは、直接出力を伴う、つまり「直接通過」のブロックです。入力に直接は依存しない出力は `nondirectOutputs` メソッドで定義されます。このような CT 基本ブロックは、間接出力を伴う、つまり「間接通過」のブロックです。

ESDL モデリング言語を使用した場合、出力の不適切な宣言（例、`nondirectOutputs` メソッド内に定義されている直接出力）は検知されます。C プログラミング言語で記述された CT ブロックでは、*nondirect* あるいは *direct* のプロパティはモデル設計者により決定されます。

nondirectOutputs および directOutputs メソッドにより、CT 基本ブロックの動作と CT 構造ブロック内の実行順序が基本的に決まります。このことをもう一度、下の例に示します。

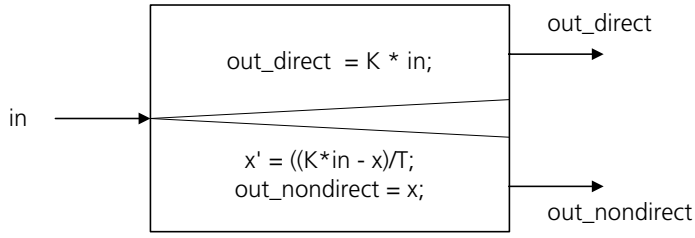


図 10-2 例：直接出力と間接出力

一般に、ある出力が入力の 1 つに直接的に依存する場合には、その出力は「直接通過」です。たとえば、増幅器ブロック（p 動作）は以下のように定義されます。

```
out = K * in
```

この出力は入力に直接依存するので、必ず入力を読みとってから出力を算出しなければなりません。したがって、この関数は directOutputs メソッドを用いて記述する必要があります。

入力に依存しない出力、たとえば、連続ステートやパラメータ条件に依存する出力は、直接通過ではありません。前のステップの値に基づいて、間接出力が算出されます。直接出力を伴う CT ブロックは、既存のループを終了させます。一例として、下のようないわゆる PT_1 動作が挙げられます。

```
x’ = ((K*in - x)/T);
out = x;
```

この微分方程式を解くために、前回の出力値と入力値 in を用いて今回の出力値 x を算出する積分メソッドを使用します。out=x という代入は、nondirectOutputs メソッドを用いて記述する必要があります（微分方程式は、derivatives メソッドで扱われます）。

2 つのシンプルな例を用いて、1 つの CT 構造ブロック内で直接出力と間接出力を使用して 2 つの CT 基本ブロックを組み合わせる際の正しい方法と誤った方法を示します。基本的に、直接出力には現在のタイムステップの入力データが必要であるということを理解する必要があります。間接出力は、現在のタイムステップの入力情報がなくても算出して出力することができます。そこで、直接出力を間接出力より後で算出するようにします。

図 10-3 では、直接通過の CT ブロックと間接通過の CT ブロックが結びつけられています。この場合、代数ループにはなりません。

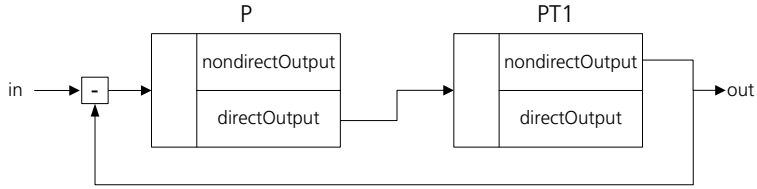


図 10-3 直接出力と間接出力により形成される CT ブロックの回路

P ブロックでの計算には妥当な入力値が必要です。そのため、最初に PT_1 ブロックの `nondirectOutputs` メソッドの計算を行ってから、P ブロックの `directOutputs` メソッドの計算を行う必要があります。

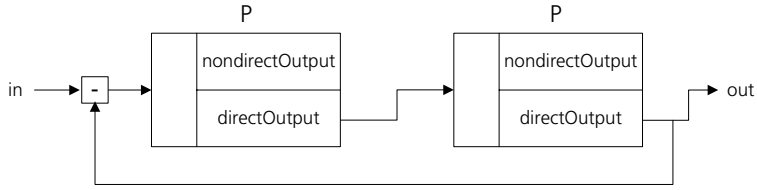


図 10-4 代数ループ

図 10-4 では、直接出力を伴う CT ブロックが 2 つ直列に接続されています。これでは、どちらのブロックにも相手のブロックからの現在の出力値が必要なため、算術ループになってしまいます。ASCET はこのエラーを報告します。

直接通過回路は回避しなければなりません。代数ループを暗黙的に解決するためには反復的なメソッドが必要で、リアルタイム環境ではそのようなメソッドを用いることはできないため、代数ループを自動的かつ暗黙的に解決するのは不可能です。

このしくみが代数ループの自動的解決より優れている点は、自分のモデルを把握しているユーザーが、直接出力を伴わないブロックを最適な位置に挿入し、次の反復ステップで後続のブロックの計算が行われるようにすることができるという点です。

原則として、代数ループを回避するには以下の 2 通りの方法があります。

1. 直接出力を伴わないブロックを挿入する方法。このブロックはストレージエレメントに相当するので、モデルのダイナミクスに悪影響が及ばないように、積分ステップ幅を小さくする必要があります。
2. モデル構造を修正して代数ループをなくす方法。CT 基本ブロックの方程式を修正し、構造を変えます。

10.7 グラフィック階層と CT 構造ブロックの違い

外見上、実行順序に関しては、CT 構造ブロックは CT 基本ブロックと同様に機能します。実行順序は CT 構造ブロック全体で決定されます。構造は、その構造ブロックの出力が入力に直接的に依存する場合は直接出力を伴う 1 つのブロックのように、あるいは間接的に依存する場合は間接出力を伴う 1 つのブロックのように機能します。

一方、階層は CT 構造ブロックをよりはっきりとレイアウトするために用いられる、純粋に表記的なものです。シミュレーションに影響を及ぼすことはありません。図 10-5（左の部分）は、CT 構造ブロックから CT 基本ブロックへの直接出力があり、その CT 基本ブロックから同じ CT 構造ブロックへの直接出力があるという例を示しています。これにより、1 つの構造ブロック内の 2 つのブロックが互いの直後に（事実上同時に）計算されるという、代数ループが発生します。構造ブロック内の第 2 の CT ブロックには、構造ブロック外の CT ブロックからの最新の出力が必要です。

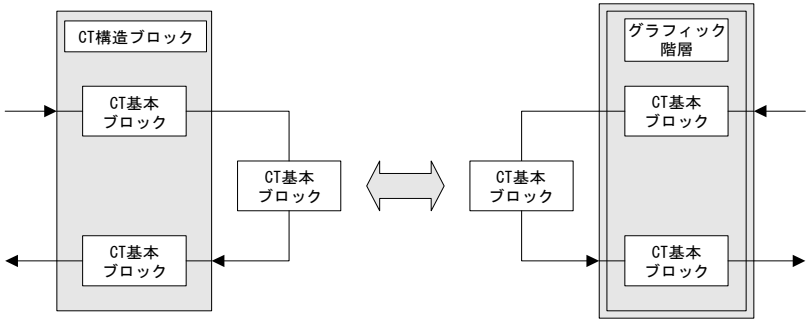


図 10-5 CT 構造ブロックを用いる構成とグラフィック階層を用いる構成

代数ループは、構造ブロックを分解し、代わりに明らかに相互に関係のあるモデル部分を結びつけて示すグラフィック階層を使用して回避できます（図 10-5 の右の部分）。各階層ごとに格納することはできないので、全階層が内包されている構造ブロックと共に格納するしかない、という点が、階層の短所です。

10.8 構造ブロック内のメソッドの実行順序

1 つの CT 構造ブロック内の実行順序は、基本的には 9.5 で説明した CT 基本ブロック内のメソッドの実行順序と同様に決まります（178 ページの図 9-4 を参照してください）。実行順序は、主として積分メソッドと選択されている時間間隔、あるいは通信周期次第です。

原則として、構造ブロック内のメソッドは、基本ブロック内のメソッドの場合と同じ順序（`init`、`nondirectOutputs`、`directOutputs`、...）で実行されます。1 つの構造ブロック内では、まず、すべての基本ブロック内の同じメソッド

が実行されてから、次のメソッドの実行に移ります。つまり、すべての基本ブロック内の `nondirectOutputs` メソッドが実行される前に、すべてのブロック内の `init` メソッドが実行されます。

どの CT 基本ブロック内でも `directOutputs` メソッドが使用されていなければ、CT 基本ブロック内の実行順序は、他のブロックからの影響を受けずに決まります。複数の CT 基本ブロックに同じメソッドが使用されている場合、それらのメソッドがどのような順序で実行されるかということはそれほど重要ではありません。つまり、まずすべての `init` メソッドが、次にすべての `nondirectOutputs` メソッドが、不定のブロック順序で実行されます。

`directOutputs` メソッドが複数のブロックで使用されている場合、`directOutputs` メソッドの入力の一部には他の出力からの最新の値が必要なので、実行順序は重要です。その入力が `nondirectOutputs` メソッドの出力に接続されている場合、`directOutputs` メソッドの前にすべての CT ブロック内の `nondirectOutputs` メソッドが実行されるので、その最新の出力値が常に確保されます。しかし、その `directOutputs` メソッドの入力が別の `directOutputs` メソッドからの出力に依存する場合には、後者のメソッドを先に実行する必要があります。

実行順序の例

図 10-6 は、結合された CT 基本ブロックからなる小さな CT 構造ブロックにおける実行順序を示します。`readInputs` は本来的な意味でのメソッドではなく、`directOutputs` に属していますが、`directOutputs` メソッド内の計算を実行するためには最新の値を最初に読みとる必要があることを強調するために、あえて示してあります。実行順序は、自動シーケンシングアルゴリズムで決められ、番号で示されます。番号が同じ場合には、その実行順序は不定です。

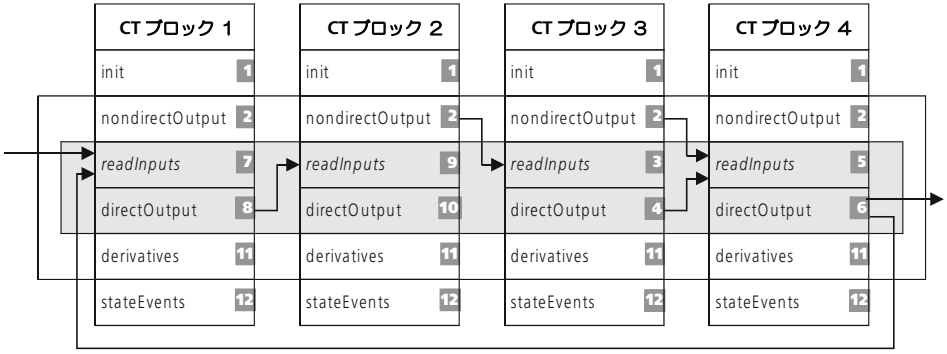


図 10-6 CT ブロックを結合した場合のメソッドの実行順序

直接出力（`directOutputs` メソッド）を伴う CT ブロックには、同じ反復周期の値を入力する必要があるため、実行順序が特に重要です（図 10-6 の陰影部分）。

CT ブロックは上から下の順に実行されます。しかも、各メソッドは 1 つずつ順に実行されます。`init` はシミュレーション開始時に一度だけ実行されます。

積分ループ内（nondirectOutputs メソッドから derivatives メソッドまで）では、すべての nondirectOutputs が必ず実行されます。それらの順序は固定ではありません。directOutputs メソッドは対応する入力に直接依存するので、ASCET は、他の directOutputs メソッドに依存しない readInput を持つ directOutputs メソッド（図 10-6 では CT 基本ブロック 3 の directOutputs メソッド）が見つかるまで、すべての directOutputs メソッド（図 10-6 の陰影部分）を検索します。その結果、入力の読みとりと directOutputs メソッドの実行は、以下の順序で行われます。

1. readIuputs（CT ブロック 3）、directOutputs（CT ブロック 3）
2. readIuputs（CT ブロック 4）、directOutputs（CT ブロック 4）
3. readIuputs（CT ブロック 1）、directOutputs（CT ブロック 1）
4. readIuputs（CT ブロック 2）、directOutputs（CT ブロック 2）

この後、derivatives メソッド 1 ～ 4 が不定の順序で実行されます。1 段階の積分メソッドの場合、次に CT ブロック 1 ～ 4 の stateEvents メソッドが不定の順序で実行されてから、nondirectOutputs に戻ります。

n 段階の積分メソッドでは、CT ブロック 1 ～ 4 の nondirectOutputs - directOutputs（前述のように適切な順序で）および derivatives が n 回実行されてから、stateEvents が実行されます（178 ページの図 9-4 も参照してください）。

つまり、CT 基本ブロックや CT 構造ブロックが 1 つの構成に結合されている場合には、積分メソッドの中間ステップでも通信が発生します。通信のたびに、nondirectOutputs メソッドから derivatives メソッドまでが実行されます（シングルラインフレーム）。

update メソッドは stateEvents の後で、通信周期 dT の粒度で実行され、terminate はシミュレーションの終わりにだけ実行されます。構造ブロック内でのこれらのメソッドの実行順序（どのブロックの update メソッドあるいは stateEvents メソッドから始めてどのようなブロック順序で実行するか）は不定です。

このように、1 つの構造ブロック内のメソッドの実行順序として、同等のシーケンスが何通りかあるのが普通です。ASCET のシーケンシングアルゴリズムは、考えられるシーケンスのうちの 1 つを自動的に選択します。

実行不可能な例：

代数ループが存在する場合、実行順序を自動的に決定することはできません。このような状況を図 10-7 に示します。どの `directOutputs` メソッドの入力も別の `directOutputs` に依存しているため、CT ブロック 4 から CT ブロック 1 までのループが閉じてしまっています。このような場合には、エラーメッセージが出力されます。

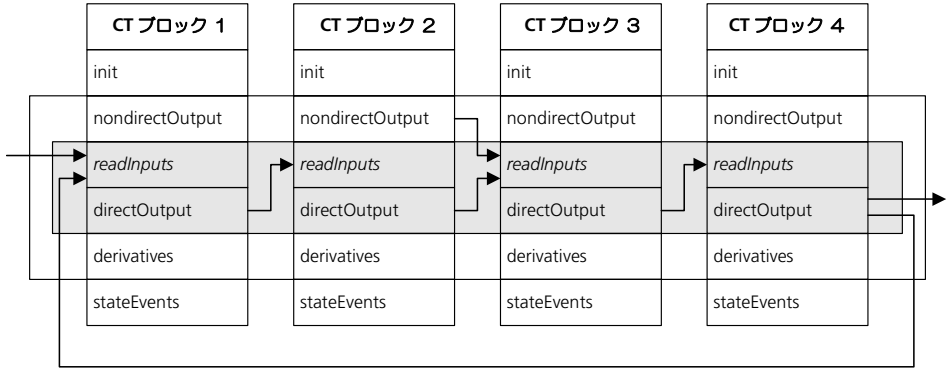


図 10-7 代数ループ

11 プロジェクトとハイブリッドプロジェクト

プロジェクトは、以下の目的で使用されます。

- CT ブロックと標準 ASCET ブロックのオンラインシミュレーション（hardware-in-the-loop シミュレーション）を行う
- 積分アルゴリズムやステップ幅が異なる複数の CT 構造を、1 つのプロジェクト内に連続系モデリングする
- CT ブロックのシミュレーションをリアルタイムに行う

1 つのプロジェクトに、標準や連続系のモジュールや構造を適宜に混在させることができます。ハイブリッドプロジェクトは、標準 ASCET ブロックと連続系コンポーネントの両方で構成されるプロジェクトです。たとえば、hardware-in-the-loop シミュレーションでは、実際のプロセスからのシグナル（連続時間的構造によりシミュレートされます）の送受信やその処理が、標準モジュールで行われるのが普通です。

たとえば油圧系コンポーネントと機械系コンポーネントのように極端に高速のコンポーネントと低速のコンポーネントを持つシステムをモデリングしシミュレーションを行う場合、互いに異なる積分メソッドや異なる積分ステップを用いて処理時間を短縮することができます。このためには、各モデル部分を、CT 基本ブロックか CT 構造ブロックに適宜に配置しなければなりません。

さまざまな CT モデル部分がプロジェクトにロードされ、ブロックダイアグラムエディタで相互に接続されます。プロジェクト内では、各 CT モデル部分（CT 基本ブロックあるいは CT 構造ブロック）を、独自の積分メソッドと積分ステップ幅を用いて独立のプロセスとして実行することができます。ただし、個々のブロックは、選択可能だが固定的な時間間隔 dT でしか相互通信しないさまざまなタスクに結びつけられていることに注意してください。

CT 構造で CT ブロックを結合した場合と同様に、積分メソッドの中間ステップでの値が交換されることはありません。また、積分の実行順序を決める CT 構造については、セマンティックスの自動確認も行われません。このことは、プロジェクトレベルの CT ブロック／構成だけに当てはまります。CT 構造内の CT ブロックや CT 構造は、プロジェクト内でもやはり積分ステップ幅の粒度で通信しあいます。

数値的安定性を保証するためには、確実に結合させたいシステムはプロジェクトレベルではなく CT 構造内で結合させる方がよいでしょう。それほど結合性の強くないシステムは、プロジェクトレベルで構築できます。動的プロパティが非常に多種多様で結合性の弱い結合のシステムの利点は、積分メソッドと積分ステップ幅を個別に選択して処理時間を最適化できることです。

図 11-1 は、1 つの離散系標準ブロックと 2 つの異なる CT 構造ブロックからなるプロジェクトを図解したものです。

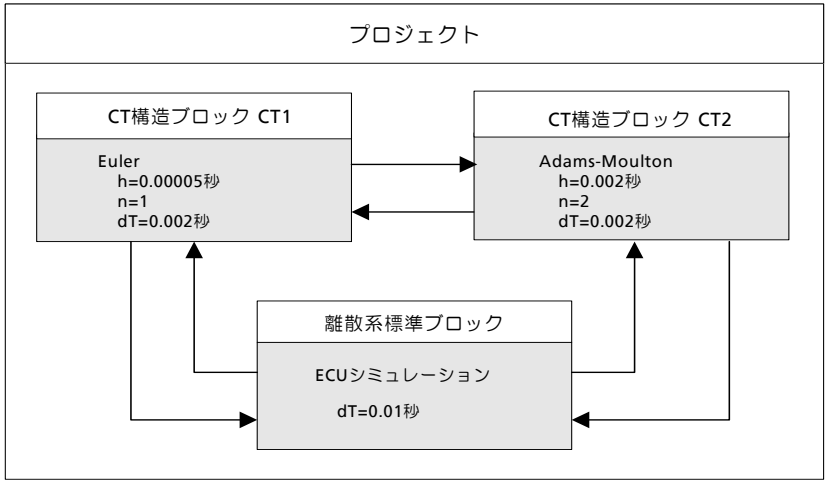


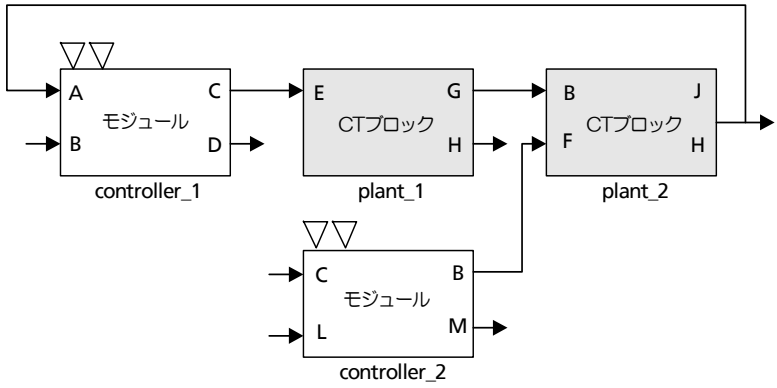
図 11-1 2 つの連続系ブロックと 1 つの離散系ブロックからなるプロジェクト

ハイブリッドプロジェクト（例、ECU テストの自動化用プロジェクト）は、コントローラモデル（標準 ASCET ブロック）と制御システムモデル（連続系の CT 構造ブロック）を結合します。連続系モデル部分自体は積分メソッドやステップ幅の異なる 2 つの CT 構造ブロックで構成されます。CT ブロックは、離散系標準ブロックと 10 ミリ秒間隔で通信しながら、他方の CT ブロックと 2 ミリ秒間隔で通信します。

11.1 連続系ブロックとモジュールの結合

プロジェクト内の離散型モジュールは、メッセージ（ASCET ブロック内のグローバル変数）を通じて通信を行います。送信メッセージと受信メッセージは明示的に（接続線で）接続されるのではなく、名前により相互に対応付けられます。

一方、連続系ブロックは、連続系ブロック同士やモジュールとの間で、グラフィックで定義された接続を通じて通信を行います。接続は、ブロックダイアグラムの場合と同じ方法を用いて確立します。



モジュール間の暗黙の接続：
controller_1 の B と controller_2 の B
controller_1 の C と controller_2 の C

モジュールとCTブロック間の暗黙の接続：
なし

CTブロック間の暗黙の接続：
なし

図 11-2 連続系ブロックとモジュールの結合

離散型モジュールについては、ユーザーはタスクを明示的に定義して、モジュールエディタで定義したプロセスを適切なタスクに割り当てる必要があります。

それに対して CT ブロックの場合は、タスクは必要に応じて自動的に定義されるので、ユーザーがタスクを明示的に定義する必要はありません。各 CT ブロックに `simulate` タスクと `event` タスクが作成されます。さらに、共通の `init` タスクと `terminate` タスクが、プロジェクト内のすべての CT ブロックに生成されます。上の例では、以下のタスクが自動生成されます。

- `simulate_CT1 (plant_1)`
- `simulate_CT2 (plant_2)`
- `event_CT1 (plant_1)`
- `event_CT2 (plant_2)`
- `initialize_CT (plant_1 ... plant_n)`
- `terminate_CT (plant_1 ... plant_n)`

これらの定義済みタスクは静的で、すべて協調タスクとして定義されます。これらのタスクの意味について、以下に詳しく説明します。

simulate_CTn タスク :

simulate_CTn タスクの場合、ステップ幅 dT の 1 シミュレーションステップが実行されます。各 simulate_CTn タスクごとに独自のステップ幅を定義することができるので、1 つのプロジェクト内にさまざまな CT 構造ブロック用の複数の積分メソッドを定義することができます。積分ステップ幅の設定は、実験中に対話形式で行うことができます。普通、simulate_CTn タスクのトリガモードは *Timer* です。

event_CTn タスク :

event_CTn タスクが呼び出されると、その基礎となる CT ブロックの event メソッドが実行されます。通常 event メソッドは非同期的に呼び出されるので、event_CTn タスクのトリガモードは *Software* か *Event* を使用します。

initialize_CT タスク :

initialize_CT タスクが呼び出されると、その基礎となる CT ブロックの init メソッドが実行されます。通常 init メソッドはシミュレーションの冒頭で実行されるので、initialize_CTn タスクのトリガモードは *Init* です。

terminate_CT タスク :

terminate_CT タスクが呼び出されると、その基礎となる CT ブロックの terminate メソッドが実行されます。terminate_CT タスクは、実験が終了するときに自動的に実行されます。

ASCET V5.2

リファレンスリスト

12 ASCET システムライブラリ

12.1 ビット演算子

12.1.1 and

2 つの引数（2 進数）の論理和を返します。

メソッド	引数	戻り値
and	bitArray1:: unsigned discrete bitArray2:: unsigned discrete	unsigned discrete

メソッドの処理

and： bitArray1 と bitArray2 の論理和（AND）を返します。

12.1.2 clearBit

引数の指定の位置のビットをリセットします。位置 0 は LSB[1] を示します。

メソッド	引数	戻り値
clearBit	bitArray:: unsigned discrete position:: unsigned discrete	unsigned discrete

メソッドの処理

clearBit： bitArray の position の位置のビットを 0 にします。

[1] **L**east **S**ignificant **B**it

12.1.3 getBit

引数の指定の位置のビットの値を論理値として戻します。

メソッド	引数	戻り値
getBit	bitArray:: unsigned discrete	logical
	position:: unsigned discrete	

メソッドの処理

getBit： bitArray の position の位置のビットが 1 なら TRUE、そうでなければ FALSE を返します。

12.1.4 or

2 つの引数（2 進数）の論理積を返します。

メソッド	引数	戻り値
or	bitArray1:: unsigned discrete	unsigned discrete
	bitArray2:: unsigned discrete	

メソッドの処理

or： bitArray1 と bitArray2 の論理積（OR）を返します。

12.1.5 rotate

引数を、指定のビット数だけ左にローテートします。

メソッド	引数	戻り値
rotate	bitArray:: unsigned discrete	unsigned discrete
	k:: unsigned discrete	

メソッドの処理

`rotate`： `bitArray` を k ビット分だけ左にローテートした値を返します。

12.1.6 setBit

引数の指定の位置のビットをセットします。位置 0 は LSB を表わします。

メソッド	引数	戻り値
`setBit`	`bitArray:: unsigned discrete` `position:: unsigned discrete`	`unsigned discrete`

メソッドの処理

`setBit`： `bitArray` の `position` の位置のビットを 1 にして返します。

12.1.7 shiftLeft

引数の全ビットを左にシフトします。シフトされた右側のビットはすべて 0（ゼロ）になります。

メソッド	引数	戻り値
`shiftLeft`	`bitArray:: unsigned discrete` `k:: unsigned discrete`	`unsigned discrete`

メソッドの処理

`shiftLeft`： `bitArray` を k ビット分だけ左にシフトした値を返します。`k=1` の場合、2 倍された値が戻ります。

12.1.8 shiftRight

引数の全ビットを右にシフトします。シフトされた左側のビットにはすべて 0（ゼロ）になります。

メソッド	引数	戻り値
shiftRight	bitArray:: unsigned discrete	unsigned discrete
	k:: unsigned discrete	

メソッドの処理

shiftRight： bitArray を k ビット分だけ右にシフトした値を返します。

12.1.9 toggleBit

引数の指定の位置のビットを反転します。

メソッド	引数	戻り値
toggleBit	bitArray:: unsigned discrete	unsigned discrete
	position:: unsigned discrete	

メソッドの処理

toggleBit： bitArray の position の位置のビットを反転して返します。

12.1.10 writeBit

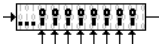
符号なし離散型の引数の指定された位置のビットに、論理型引数の値を書き込みます。

メソッド	引数	戻り値
writeBit	bitArray:: unsigned discrete	unsigned discrete
	aBool::logical	
	position:: unsigned discrete	

メソッドの処理

writeBit：　aBool = FALSE の場合、bitArray の position の位置のビットに 0 を書き込んで返します。aBool = TRUE の場合は、同じ位置のビットに 1 を書き込んで返します。

12.1.11 writeByte



8 個の論理入力の値を引数の下位のビット位置に書き込みます。

メソッド	引数	戻り値
writeByte	bitArray:: unsigned discrete	unsigned discrete
	b0::logical	
	b1::logical	
	b2::logical	
	b3::logical	
	b4::logical	
	b5::logical	
	b6::logical	
	b7::logical	

メソッドの処理

writeByte： bitArray のビット 0 ～ 7 に b0 ～ b7 の値を書き込んで返します。ビット 0 は LSB です。論理値 TRUE および FALSE は、1 と 0 に相当します。

12.1.12 xor

2 つの引数（2 進数）の排他的論理和を返します。

メソッド	引数	戻り値
xor	bitArray1:: unsigned discrete bitArray2:: unsigned discrete	unsigned discrete

メソッドの処理

xor： bitArray1 と bitArray2 の排他的論理和（XOR）を返します。

12.2 コンパレータ

12.2.1 ClosedInterval

x の値が A 以上 B 以下である場合に TRUE を返します。

メソッド	引数	戻り値
out	x::continuous A::continuous B::continuous	logical

各メソッドの処理

out： A <= x <= B なら TRUE を、そうでなければ FALSE を返します。

12.2.2 LeftOpenInterval

x の値が A より大きく B 以下である場合に TRUE を返します。

メソッド	引数	戻り値
out	x::continuous	logical
	A::continuous	
	B::continuous	

各メソッドの処理

out： A < x <= B なら TRUE を、そうでなければ FALSE を返します。

12.2.3 OpenInterval

x の値が A より大きく B より小さい場合に TRUE を返します。

メソッド	引数	戻り値
out	x::continuous	logical
	A::continuous	
	B::continuous	

各メソッドの処理

out： A < x < B なら TRUE を、そうでなければ FALSE を返します。

12.2.4 RightOpenInterval

x の値が A 以上 B 未満の値である場合に TRUE を返します。

メソッド	引数	戻り値
out	x::continuous	logical
	A::continuous	
	B::continuous	

各メソッドの処理

`out`： A <= x < B なら TRUE を、そうでなければ FALSE を返します。

12.2.5 GreaterZero

x の値がゼロより大きい場合に TRUE を返します。

メソッド	引数	戻り値
`out`	`x::continuous`	`logical`

各メソッドの処理

`out`： x > 0.0 なら TRUE を、そうでなければ FALSE を返します。

12.3 カウンタとタイマ

12.3.1 CountDown

タイムカウンタを 1 だけデクリメントし、ゼロになった場合はその旨を通知します。

メソッド	引数	戻り値
`start`	`startValue:: unsigned discrete`	`none`
`compute`	`none`	`none`
`out`	`none`	`logical`

各メソッドの処理

`start`： カウンタに `startValue` をセットします。

`compute`： カウンタを 1 だけデクリメントします。

`out`： 現在のカウンタ値が 0 より大きければ TRUE を、そうでなければ FALSE を返します。

12.3.2 CountDownEnabled

タイムカウンタを 1 だけデクリメントし、ゼロになった場合はその旨を通知します。カウント動作は、明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
start	startValue:: unsigned discrete	none
compute	enable::logical	none
out	none	logical

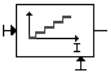
各メソッドの処理

start：　カウンタに startValue をセットします。

compute：　enable が TRUE なら、カウンタを 1 だけデクリメントします。

out：　現在のカウンタ値がゼロより大きければ TRUE を、そうでなければ FALSE を返します。

12.3.3 Counter



タイムカウンタを 1 だけインクリメントします。

メソッド	引数	戻り値
reset	none	none
compute	none	none
out	none	unsigned discrete

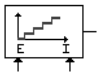
各メソッドの処理

reset：　カウンタ値をゼロにします。

compute：　カウンタを 1 だけインクリメントします。

out：　現在のカウンタ値を返します。

12.3.4 CounterEnabled



タイムカウンタを 1 だけインクリメントします。カウント動作は、明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initEnable::logical	none
compute	enable::logical	none
out	none	unsigned discrete

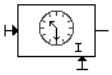
各メソッドの処理

reset：　initEnable が TRUE なら、カウンタ値をゼロにします。

compute：　enable が TRUE なら、カウンタを 1 だけインクリメントします。

out：　現在のカウンタ値を返します。

12.3.5 StopWatch



タイムカウンタを 1dT 分だけインクリメントします。

メソッド	引数	戻り値
reset	none	none
compute	none	none
out	none	continuous

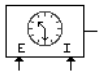
各メソッドの処理

reset：　カウンタ値をゼロにします。

compute：　カウンタを dT だけインクリメントします。

out：　現在のカウンタ値、つまり、カウント開始時からの経過時間を返します。

12.3.6 StopWatchEnabled



タイムカウンタを 1dT 分だけインクリメントします。このカウント動作は、明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initEnable::logical	none
compute	enable::logical	none
out	none	continuous

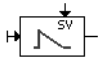
各メソッドの処理

reset： initEnable が TRUE なら、カウンタ値を 0 にします。

compute： enable が TRUE なら、カウンタを dT だけインクリメントします。

out： 現在のカウンタ値、つまり、カウント開始以来 enabled が TRUE だった時間の合計を返します。

12.3.7 Timer



タイムカウンタを 1dT 分だけデクリメントし、ゼロになった場合はその旨を通知します。カウント中の再起動はできません。

メソッド	引数	戻り値
start	startTime::continuous	none
compute	none	none
out	none	logical

各メソッドの処理

start： タイムカウンタ値がゼロ以下なら、それを startTime にセットします。

compute： タイムカウンタを dT だけデクリメントします。

out： タイムカウンタ値がゼロより大きければ TRUE を、そうでなければ FALSE を返します。

12.3.8 TimerEnabled

タイムカウンタを 1dT 分だけデクリメントし、ゼロになった場合はその旨を通知します。カウント動作は、明示的にイネーブルにする必要があります。

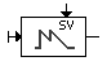
メソッド	引数	戻り値
compute	enable::logical in::logical startTime:: continuous	none
out	none	logical

各メソッドの処理

compute：enable が TRUE の場合、タイムカウンタ値がゼロ以下の時に in が立ち上がると、タイマを起動（タイマカウンタに startTime をセット）し、それ以外はタイムカウンタを dT だけデクリメントします。enable が FALSE の場合には何も行いません。

out：タイムカウンタ値がゼロより大きい場合は TRUE を、それ以外は FALSE を返します。

12.3.9 TimerRetrigger



タイムカウンタを 1dT 分だけデクリメントし、ゼロになった場合はその旨を通知します。カウント中の再起動が可能です。

メソッド	引数	戻り値
start	startTime::continuous	none
compute	none	none
out	none	logical

各メソッドの処理

start：タイムカウンタに startTime をセットします。

compute：タイムカウンタを dT だけデクリメントします。

out：タイムカウンタ値がゼロより大きい場合は TRUE を、それ以外は FALSE を返します。

12.3.10 TimerRetriggerEnabled

タイムカウンタを 1dT 分だけデクリメントし、ゼロになった場合はその旨を通知します。カウント中の再起動が可能です。カウント動作は、明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
compute	enable::logical in::logical startValue::continuous	none
out	none	logical

各メソッドの処理

compute：enable が TRUE の場合、in が立ち上がるとタイマを起動（タイマカウンタに startValue をセット）し、それ以外はタイムカウンタを dT（タイムフレーム）だけデクリメントします。enable が FALSE の場合には何も行いません。

out：タイムカウンタ値がゼロより大きい場合は TRUE を、それ以外は FALSE を返します。

12.4 遅延

12.4.1 DelaySignal

入力シグナルを 1 ステップ遅らせて出力します。

メソッド	引数	戻り値
compute	signal::logical	none
out	none	logical

各メソッドの処理

compute：入力シグナルをバッファに格納します。

out：バッファに格納された前回のシグナルを返します。この結果、入力シグナルが 1 ステップ分遅れて出力されることになります。

12.4.2 DelaySignalEnabled

入力シグナルを 1 ステップ遅らせて出力します。遅延処理は明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initEnable::logical initValue::logical	none
compute	signal::logical enable::logical	none
out	none	logical

各メソッドの処理

reset：　initEnable が TRUE の場合、initValue をバッファに格納します。

compute：　enable が TRUE の場合、入力シグナルをバッファに格納します。

out：　バッファに格納されたシグナルを返します。この結果、入力シグナルが 1 ステップ分遅れて出力されることになります。

12.4.3 DelayValue

入力値を 1 ステップ遅らせて出力します。

メソッド	引数	戻り値
compute	value::continuous	none
out	none	continuous

各メソッドの処理

compute：　入力した value をバッファに格納します。

out：　バッファに格納された値を返します。この結果、入力値が 1 ステップ分遅れて出力されることになります。

12.4.4 DelayValueEnabled

入力値を 1 ステップだけ遅らせて出力します。遅延処理は明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initEnable::logical initValue::continuous	none
compute	value::continuous enable::logical	none
out	none	logical

各メソッドの処理

reset：　initEnable が TRUE の場合、initValue をバッファに格納します。

compute：　enable が TRUE の場合、入力値をバッファに格納します。

out：　バッファに格納された値を返します。この結果、入力値が 1 ステップ分遅れて出力されることになります。

12.4.5 TurnOffDelay

入力シグナルの立ち下がりを遅らせます。

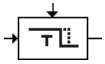
メソッド	引数	戻り値
compute	signal::logical delayTime:: continuous	none
out	none	logical

各メソッドの処理

compute：　入力シグナルの立ち下がりを遅らせます。シグナルが TRUE から FALSE に変わるとタイマを起動します。シグナルが FALSE である間は、タイマ値を dT だけインクリメントしてから delayTime と比較します。入力シグナルが TRUE の場合は、タイマをリセットします。

out：　入力シグナルが TRUE の場合、あるいはタイマ値が delayTime を超えていない場合、TRUE を返します。それ以外は、FALSE を返します。

12.4.6 TurnOffDelayVariable



入力シグナルの立ち下がりを遅らせます。Time 変数により、実行時に遅延時間を変更することができます。

メソッド	引数	戻り値
compute	signal::logical delayTime::continuous	none
out	none	logical

各メソッドの処理

compute：　入力シグナルの立ち下がりを遅らせます。シグナルが TRUE から FALSE に変わるとタイマを起動します。シグナルが FALSE である間は、タイマ値を dT だけインクリメントしてから delayTime と比較します。入力シグナルが TRUE の場合は、タイマをリセットします。

out：　入力シグナルが TRUE の場合、あるいはタイマ値が delayTime を超えていない場合、TRUE を返します。それ以外は、FALSE を返します。

12.4.7 TurnOnDelay

入力シグナルの立ち上がりを遅らせます。

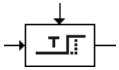
メソッド	引数	戻り値
compute	signal::logical delayTime::continuous	none
out	none	logical

各メソッドの処理

compute：　入力シグナルの立ち上がりを遅らせます。シグナルが FALSE から TRUE に変わるとタイマを起動します。シグナルが TRUE である間は、タイマ値を dT だけインクリメントしてから delayTime と比較します。入力シグナルが FALSE の場合は、タイマをリセットします。

out：　入力シグナルが FALSE の場合、あるいはタイマ値が delayTime を超えていない場合、FALSE を返します。それ以外は、TRUE を返します。

12.4.8 TurnOnDelayVariable



入力シグナルの立ち上がりを遅らせます。Time 変数により、実行時に遅延時間を変更することができます。

メソッド	引数	戻り値
compute	signal::logical delayTime::continuous	none
out	none	logical

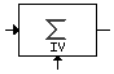
各メソッドの処理

compute：　入力シグナルの立ち上がりを遅らせます。シグナルが FALSE から TRUE に変わるとタイマを起動します。シグナルが TRUE である間は、タイマ値を dT だけインクリメントしてから delayTime と比較します。入力シグナルが FALSE の場合は、タイマをリセットします。

out：　入力シグナルが FALSE の場合、あるいはタイマ値が delayTime を超えていない場合、FALSE を返します。それ以外は、TRUE を返します。

12.5 メモリ

12.5.1 Accumulator



入力値を積算します。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	value::continuous	none
out	none	continuous

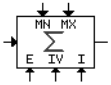
各メソッドの処理

reset：　アキュムレータに initValue をセットします。

compute：　以下のように、アキュムレータ値を入力値だけインクリメントします。
`accumulator (new) = accumulator (old) + input value`

out：　アキュムレータ値を返します。

12.5.2 AccumulatorEnabled



入力値を積算します。アキュムレータは明示的にイネーブルにする必要があります。このアキュムレータ値には上下限値を設けることができます。

メソッド	引数	戻り値
reset	initValue::continuous initEnable::logical	none
compute	value::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

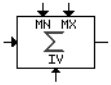
各メソッドの処理

reset：　initEnable が TRUE の場合、アキュムレータに initValue をセットします。

compute：　enable が TRUE の場合、以下のようにアキュムレータ値を入力値だけインクリメントします。
`accumulator (new) = accumulator (old) + input value`
ただし、アキュムレータ値は mn から mx までの値に限定されます。

out：　アキュムレータ値を返します。

12.5.3 AccumulatorLimited



入力値を積算します。このアキュムレータ値には上下限値を設けることができます。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	value::continuous mn::continuous mx::continuous	none
out	none	continuous

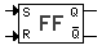
各メソッドの処理

reset：　アキュムレータ値を initValue にセットします。

compute：　以下のようにアキュムレータ値を入力値だけインクリメントします。
accumulator (new) = accumulator (old) + input value
ただし、アキュムレータ値は mn から mx までの値に限定されます。

out：　アキュムレータ値を返します。

12.5.4 RSFlipFlop



リセットとセットの入力を持つフリップフロップで、リセット入力の方がセット入力よりも優先されます。

メソッド	引数	戻り値
compute	r::logical s::logical	none
q	none	logical
nq	none	logical

各メソッドの処理

compute：　r が TRUE の場合、フリップフロップの状態を FALSE にします。r が FALSE で s が TRUE の場合は、状態を TRUE にします。r も s も FALSE の場合は、状態を変更しません。

q： フリップフロップの状態を返します。
nq： 状態の反対の値を返します。

12.6 その他

12.6.1 DeltaOneStep

今回と前回の入力値の差を返します。

メソッド	引数	戻り値
compute	value::continuous	none
out	none	continuous

各メソッドの処理

compute： 今回の入力値から前回の入力値を引きます。
out： 差を返します。

12.6.2 DifferenceQuotient

前回と今回の入力値の差分商を算出します。

メソッド	引数	戻り値
compute	value::continuous	none
out	none	continuous

各メソッドの処理

compute： (value - previous value)/dT を計算して差分商を求めます。
out： 差分商を返します。

12.6.3 EdgeBi

論理入力シグナルの立ち上がりと立ち下がりを検知します。

メソッド	引数	戻り値
compute	signal::logical	none
out	none	logical

各メソッドの処理

compute：　今回と前回の入力シグナルを比較します。

out：　今回と前回の入力シグナルが異なる場合は TRUE を、それ以外は FALSE を返します。

12.6.4 EdgeFalling

論理入力シグナルの立ち下がりを検知します。

メソッド	引数	戻り値
compute	signal::logical	none
out	none	logical

各メソッドの処理

compute：　今回と前回の入力シグナルを比較します。

out：　今回の入力シグナルが LOW で前回の入力シグナルが HIGH の場合は TRUE を、それ以外は FALSE を返します。

12.6.5 EdgeRising

論理入力シグナルの立ち上がりを検知します。

メソッド	引数	戻り値
compute	signal::logical	none
out	none	logical

各メソッドの処理

`compute`： 今回と前回の入力シグナルを比較します。

`out`： 今回の入力シグナルが HIGH で前回の入力シグナルが LOW の場合は TRUE を、それ以外は FALSE を返します。

12.6.6 Mux1of4

4 個の入力値 `s0, ..., s3` のうち、2 進数のインデックス値で示されるものを出力します。

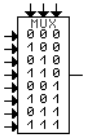
メソッド	引数	戻り値

`out`	`b0::logical`
	`b1::logical`
	`s0::continuous`
	`s1::continuous`
	`s2::continuous`
	`s3::continuous`
		`continuous`

各メソッドの処理

`out`： 入力値 `si`（`index i、i = b0 + 2*b1`）を返します。FALSE は 0、TRUE は 1 と解釈されます。

12.6.7 Mux1of8



8 個の入力値 s0, ..., s7 のうち、2 進数のインデックス値で示されるものを出力します。

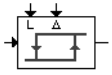
メソッド	引数	戻り値
out	b0::logical b1::logical b2::logical s0::continuous s1::continuous s2::continuous s3::continuous s4::continuous s5::continuous s6::continuous s7::continuous	continuous

各メソッドの処理

out： 入力値 si（index i、i = b0 + 2*b1 + 4*b2）を返します。FALSE は 0、TRUE は 1 と解釈されます。

12.7 非線型変換

12.7.1 Hysteresis-Delta-RSP



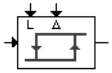
右側の切り替え点とオフセット delta を持つヒステリシスです。

メソッド	引数	戻り値
out	x::continuous delta::continuous rsp::continuous	logical

各メソッドの処理

out：　　x > rsp の場合は TRUE を、x < (rsp - delta) の場合は FALSE を返します。x の値が開区間](rsp - delta), rsp[内である場合は、戻り値は前回のままです。

12.7.2 Hysteresis-LSP-Delta



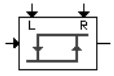
左側の切り替え点とオフセット delta を持つヒステリシスです。

メソッド	引数	戻り値
out	x::continuous lsp::continuous delta::continuous	logical

各メソッドの処理

out：　　x > (lsp + delta) の場合は TRUE を、x < lsp の場合は FALSE を返します。x の値が開区間]lsp,(lsp + delta)[内である場合は、戻り値は前回のままです。

12.7.3 Hysteresis-LSP-RSP



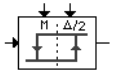
左右の切り替え点を持つヒステリシスです。

メソッド	引数	戻り値
out	x::continuous lsp::continuous rsp::continuous	logical

各メソッドの処理

out：　　x > rsp の場合は TRUE を、x < lsp の場合は FALSE を返します。x の値が開区間]lsp, rsp[内である場合は、戻り値は前回のままです。

12.7.4 Hysteresis-MSP-DeltaHalf



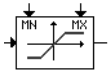
中間の切り替え点とオフセット delta/2 を持つヒステリシスです。

メソッド	引数	戻り値
out	x::continuous msp::continuous deltahalf:: continuous	logical

各メソッドの処理

out：　x > (msp + deltahalf) の場合は TRUE を、x < (msp - deltahalf) の場合は FALSE を返します。x の値が開区間](msp - deltahalf), (msp + deltahalf)[内である場合は、戻り値は前回のままです。

12.7.5 Limiter



入力値の上下限をリミッタ mn および mx によって制限します。

メソッド	引数	戻り値
out	x::continuous mn::continuous mx::continuous	continuous

各メソッドの処理

out：　以下のように入力値を mn から mx までの値にして返します。
max(min(x, mx), mn)
なお mn <= mx かどうかのチェックは行いません。

12.7.6 Signum

入力値の符号を返します。

メソッド	引数	戻り値
out	x::continuous	continuous

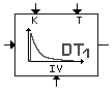
各メソッドの処理

out：　`x > 0.0` の場合は `1.0`、`x = 0.0` の場合は `0.0`、それ以外は `-1.0` を返します。

12.8 伝達関数

12.8.1 制御

dT1



時定数が T でゲインが K の、時間離散型の微分伝達関数です。

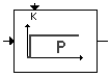
メソッド	引数	戻り値
compute	in::continuous T::continuous K::continuous	none
out	none	continuous

各メソッドの処理

compute：　P（比例）関数、および逆結合された I（積分）関数により、微分値を算出します。

out：　微分値を返します。

P



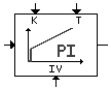
ゲインが K の、時間離散型の正比例伝達関数です。

メソッド	引数	戻り値
out	in::continuous K::continuous	continuous

各メソッドの処理

out：　戻り値 `out = in * K` を算出します。

PI



時定数が T、ゲインが K の時間離散型の正比例積分器です。

メソッド	引数	戻り値
reset	initValue:: continuous	none
compute	in::continuous T::continuous K::continuous	none
out	none	continuous

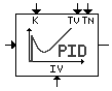
各メソッドの処理

reset：　積分器に initValue の値をセットします。

compute：　P 関数と I 関数の合計を算出して PI 関数の値とします。

out：　PI 関数の値を返します。

PID



時定数が Tv と Tn でゲインが K の、微分部分のある時間離散型の正比例積分器です。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous Tv::continuous Tn::continuous K::continuous	none
out	none	continuous

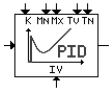
各メソッドの処理

reset：　積分器に initValue の値をセットします。

compute：　P（比例）関数、D（微分）関数および I（積分）関数の合計を算出して PID 関数の値とします。

out：　PID 関数の値を返します。

PIDLimited



時定数が Tv と Tn でゲインが K の、微分部分のある時間離散型の正比例積分器です。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous Tv::continuous Tn::continuous K::continuous mn::continuous mx::continuous	none
out	none	continuous

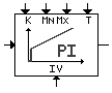
各メソッドの処理

reset：　積分器に initValue の値をセットします。

compute：　P 関数、D 関数、および I 関数の合計を算出して PID 関数の値とします。ただし I 関数の積分器の値は、mn から mx までの範囲に制限されます。

out：　PID 関数の値を返します。

PILimited



時定数が T でゲインが K の時間離散型の正比例積分器です。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous T::continuous K::continuous mn::continuous mx::continuous	none
out	none	continuous

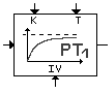
各メソッドの処理

`reset`： 積分器に `initValue` の値をセットします。

`compute`： P 関数と I 関数の合計を算出して PI 関数の値とします。ただし、積分関数の積分器の値は、`mn` から `mx` までの範囲に制限されます。

`out`： PI 関数の値を返します。

PT1



時定数が T でゲインが K の、時間離散型ローパスフィルタです。

メソッド	引数	戻り値
`reset`	`initValue::continuous`	`none`

`compute`	`in::continuous`
	`T::continuous`
	`K::continuous`
		`none`

`out`	`none`	`continuous`

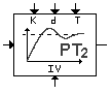
各メソッドの処理

`reset`： 積分器に `initValue` の値をセットします。

`compute`： I 関数、および逆結合された P 関数により、PT1 関数の値を算出します。

`out`： PT1 関数の値を返します。

PT2



時定数が T でゲインが K、および減衰が d の、時間離散型遅延関数です。

メソッド	引数	戻り値
`reset`	`initValue::continuous`	`none`

`compute`	`in::continuous`
	`T::continuous`
	`K::continuous`
	`d::continuous`
		`none`

`out`	`none`	`continuous`

各メソッドの処理

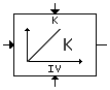
`reset`： 2 つの積分器に `initValue` の値をセットします。

`compute`： カスケーディングされている 2 つの P 関数で逆結合されている 2 つの I 関数を続けて実行することにより、PT2 関数の値を算出します。

`out`： PT2 関数の値を返します。

12.8.2 積分器

`IntegratorK`



ゲインが K の、時間離散型積分器です。

メソッド	引数	戻り値
`reset`	`initValue::continuous`	`none`
`compute`	`in::continuous` `K::continuous`	`none`
`out`	`none`	`continuous`

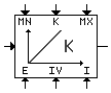
各メソッドの処理

`reset`： 積分器に `initValue` の値をセットします。

`compute`： `integrator (new) = integrator (old) + in * dT * K` を計算することにより、積分器の値を算出します。

`out`： 積分器の値を返します。

IntegratorKEnabled



ゲインが K の、時間離散型積分器です。この積分器の動作は明示的にイネーブルにする必要があります。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous initEnable::logical	none
compute	in::continuous K::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

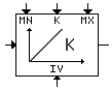
各メソッドの処理

reset：　initEnable が TRUE の場合は、積分器に initValue の値をセットします。

compute：　enable が TRUE の場合は、integrator (new) = integrator (old) + in * dT * K を計算することにより、積分器の値を算出します。ただし、この積分器の値は mn から mx までの範囲に制限されます。

out：　積分器の値を返します。

IntegratorKLimited



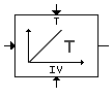
ゲインが K の、時間離散型積分器です。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous K::continuous mn::continuous mx::continuous	none
out	none	continuous

各メソッドの処理

reset：	積分器に initValue の値をセットします。
compute：	integrator (new) = integrator (old) + in * dT * K を計算することにより、積分器の値を算出します。ただし、この積分器の値は mn から mx までの範囲に制限されます。
out：	積分器の値を返します。

IntegratorT



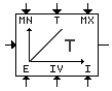
時定数が T の時間離散型積分器です。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous T::continuous	none
out	none	continuous

各メソッドの処理

reset：	積分器に initValue の値をセットします。
compute：	integrator (new) = integrator (old) + in * dT / T を計算することにより、積分器の値を算出します。
out：	積分器の値を返します。

IntegratorTEnabled



時定数が T の、時間離散型積分器です。この積分器の動作は明示的にイネーブルにする必要があります。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous initEnable::logical	none
compute	in::continuous T::continuous mn::continuous mx::continuous enable::logical	none
out	none	continuous

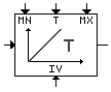
各メソッドの処理

reset：　initEnable が TRUE の場合は、積分器に initValue の値をセットします。

compute：　enable が TRUE の場合は、integrator (new) = integrator (old) + in * dT / T を計算することにより、積分器の値を算出します。ただし、この積分器の値は mn から mx までの範囲に制限されます。

out：　積分器の値を返します。

IntegratorTLimited



時定数が T の、時間離散型積分器です。この積分器は値の範囲を制限することができます。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous T::continuous mn::continuous mx::continuous	none
out	none	continuous

各メソッドの処理

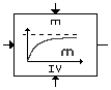
reset：　積分器に initValue の値をセットします。

compute：　integrator (new) = integrator (old) + in * dT / T を計算することにより、積分器の値を算出します。ただし、この積分器の値は mn から mx までの範囲に制限されます。

out：　積分器の値を返します。

12.8.3 ローパス

DigitalLowpass



入力値の中間値を再帰的に算出します。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous m::continuous	none
out	none	continuous

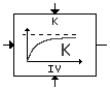
各メソッドの処理

reset：　中間値に initValue の値をセットします。

compute：　mean value (new) = mean value (old) + m * (in - mean value (old)) を計算することにより、中間値を算出します。

out：　中間値を返します。

LowpassK



ゲインが K の、単純化された PT1 関数（ローパスフィルタ）です。

メソッド	引数	戻り値
reset	initValue:: continuous	none
compute	in::continuous K::continuous	none
out	none	continuous

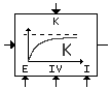
各メソッドの処理

reset：　ローパスに initValue の値をセットします。

compute：　lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT * K を計算することにより、ローパス値を算出します。

out：　ローパス値を返します。

LowpassKEnabled



ゲインが K の、単純化された PT1 関数（ローパスフィルタ）です。この関数の動作は明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initValue::continuous initEnable::logical	none
compute	in::continuous K::continuous enable::logical	none
out	none	continuous

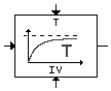
各メソッドの処理

reset：　initEnable が TRUE の場合は、ローパスに initValue の値をセットします。

compute：　enable が TRUE の場合は、lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT * K を計算することにより、ローパス値を算出します。

out：　ローパス値を返します。

LowpassT



時定数が T の、単純化された PT1 関数（ローパスフィルタ）です。

メソッド	引数	戻り値
reset	initValue::continuous	none
compute	in::continuous T::continuous	none
out	none	continuous

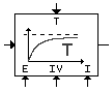
各メソッドの処理

reset：　ローパスに initValue の値をセットします。

compute：　lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT / T を計算することにより、ローパス値を算出します。

out：　ローパス値を返します。

LowpassTEnabled



時定数が T の、単純化された PT1 関数（ローパスフィルタ）です。この関数の動作は明示的にイネーブルにする必要があります。

メソッド	引数	戻り値
reset	initValue::continuous initEnable::logical	none
compute	in::continuous T::continuous enable::logical	none
out	none	continuous

各メソッドの処理

reset：　initEnable が TRUE の場合は、ローパスに initValue の値をセットします。

compute：　enable が TRUE の場合は、lowpass (new) = lowpass (old) + (in - lowpass (old)) * dT / T を計算することにより、ローパス値を算出します。

out：　　ローパス値を返します。

13 トラブルシューティング

本章では、ASCET の作業中に発生する可能性のある問題について説明し、それらを解決するためのヒントを紹介します。本章に記載されていない問題が発生した場合には、ETAS にご連絡ください。本章の改訂に役立てさせていただきます。

一般に、ASCET が知らせるシステムエラーはどれも重大なものです。システムエラーが発生した時は、すべてのデータをデータベースに格納することをお勧めします。システムエラーが発生した後にシステムが予想外の動作を始めた場合は、システムエラーが原因で実行中のシステムに不整合が生じています。このような場合には、ASCET を終了してから Windows を再起動してください。

13.1 一般的なヒント

データベースサイズの限界：ASCET のデータベースのサイズは最大 128M バイトです。この限界を超えるとデータベースが壊れてしまうので、大きなデータベースを扱う場合にはサイズが限界に達しないように注意してください。データベースツールを利用し、必要に応じて、データベースを圧縮してください。

データベースの変換：ASCET-SD V4.1 または V4.2、または ASCET V5.0 で開発されたデータベースは、自動的に ASCET 5.2 用のものに変換されます。変換後のデータベースは旧バージョンの ASCET では使用できません。変換時に、旧バージョンのデータベースのバックアップコピーが自動的に作成されます。

ASCET V4.0 またはそれ以前のデータベースは、ASCET 5.2 で直接開くことはできません。

ASCET は ANSI C に準拠する名前しかサポートしません。互換性を維持するために、内蔵変換ツールを使用して、データベース内のすべてのアイテムの名前を調整する必要があります。コンポーネントマネージャで **Tools → Database → Convert → All Names To ANSI C** を選択し、すべてのアイテムの名前を変換してください。

グラフィックカードの問題：ASCET ウィンドウの表示に関する問題が発生した場合には、ASCET、グラフィックカード、およびグラフィックカードドライバの間に不適合が生じていると考えられます。このような場合、グラフィックカード用の最新のドライバ（カードのメーカーからインターネットで入手可能）を試してみるか、別の解像度を試してみてください。通常は、すべての標準 VGA および SVGA モードを使用できるはずです。

オフライン実験のタイムアウト：オフライン実験用の時間（dT）は、およそ 3 日間以内（dT 単位）でなければならないので、dT の値が大きくなると（例、1000 秒）、
オフライン実験は数分後にクラッシュします。

複合代入が行われた場合の予想外の影響：モデル内で複合代入（複合エレメント同士の代入）が使用されていると、複合エレメントの測定時に予想外の影響が出ます。複合代入は、両複合エレメントのポインタの代入により表現されます。つ

まり、両方のオブジェクトは代入後は同一のものとなるため、一方のオブジェクトは失われてしまいます。たとえば、A=B という代入では、エレメント A はエレメント B になります。しかし、その後も測定／適合システムは A と B を別のオブジェ
クトとして扱います。このような失われたオブジェクト（この場合はオブジェクト A）の測定や適合を行うことはできますが、代入後の複合エレメントを表すオブジェクト（オブジェクト B）が考慮されないため、実際には無意味な作業となってしまいます。

フォントが正しく表示されない：Arial フォントファミリは Win 97/Win NT では正しく表示されないので、一部のアイテムが読み取りにくくなります。代わりにサイズ 10 の Microsoft SansSerif Font を使用してください。このフォントにすれば、表示上の問題はありません。

外部実験ターゲットの問題：Centronics リンクケーブルを使用する場合に考えられるエラー原因は、パラレルポートの速度が Centronics リンクケーブルには速すぎることです（特に、Pentium 200 以上のプロセッサを使用している場合）。このような場合には、コンピュータの BIOS セットアップでパラレルポートを設定し直すことをお勧めします。

ASCET の過負荷状態：ASCET がビジー状態の時（たとえば、コード生成中やデータベースへの保存中）には、ASCET の他の操作は行わず、現在のアクションが完了するまで待ってください。そうしないと、システムエラーなどの予想できないエラーを招く可能性があります。

13.2 ASCET 使用時に発生する可能性のあるトラブル

ASCET のある特定の実験が終了しないか、正しく実行できない：ASCET モデルに組み込まれた C コードに問題があることが多いようです。C コード中に、パラメータの渡し方の誤り（ASCET の `continuous` 型を変換する場合には、C の `double float` 型を選択しなければなりません）や無限ループなどがあると考えられます。無限ループは、再帰的なオブジェクト構造でも発生する可能性があります。このような問題箇所を見つけだすために、問題のありそうな C コードコンポーネントを除外して実行してみるとよいでしょう。

また、生成されたコードがスケジュールされたタイムフレームで実行されない、つまり、実行時間が長すぎる場合があります。このような場合には、仕様を変更するか、インターバルの長いタイムフレームを割り当て直す必要があります。

他に考えられる原因としては、シーケンスコールが割り当てられていなかったり、正しく設定されていないことも考えられます。

コンパイルで予期しないエラーメッセージが返ってくるか、コンパイルが終わらない：コンパイル中に別のウィンドウをクリックすると、コンパイルが行われている DOS ウィンドウの優先順位が大幅に下がってしまうので、コンパイル処理が停止してしまいます。このような場合には、DOS ウィンドウを、そのアイコンをダブルクリックしてアクティブにしてください。

また、以下のキーワードはエラー管理システムがコンパイラエラーを ASCET モデルに表示する際に使用するので、プログラム内で以下のキーワードを使用しないでください。
Error、ERROR、Serious、Fatal、illegal、Failed、failed、warning、known format

テンポラリ変数を使用すると、ASCET の計算が正しく行われない：1 つの変換式の結果がいくつかの分岐先で使用される場合には、自動的に作成されるテンポラリ変数を使用することができます。このようなテンポラリ変数は一度だけ、最初の分岐を評価する時に計算されます。このテンポラリ変数を使用する分岐が一定の条件下でしか実行されない場合（例えば、それらの分岐がスイッチや MUX 演算子への入力の場合）には、テンポラリ変数の値が正しく計算されないことがあります。そのため、テンポラリ変数から続く分岐が条件演算子への入力になる場合には、自動的に作成されるテンポラリ変数を使用しないでください。

オンライン実験中に L1 通信エラーが頻発する：このような場合、通信プロセスの優先度が低すぎる可能性があります。このプロセスの優先度を上げるには、ASCET のターゲットディレクトリにあるコンパイラ設定ファイル（`es1130cp.inv`、`es1130cp_gnu.inv`、`es1135cp_gnu.inv` のいずれか）を編集します。使用されるファイルは、ターゲットとコンパイラの組み合わせに応じて決まります。

これらのファイル内には通信プロセスの優先度を指定するパラメータ `__L1_Prio =` があり、デフォルト値として最低レベルの値（0 など）が設定されていますので、これを適切な優先度に変更してください。

`.rtf` フォーマットのドキュメントを正しく作成できない：Windows 用の Word で `.rtf` ファイルを表示すると、ファイルに組み込まれているビットマップイメージファイルを表示できない場合があります。そのような場合には、外部の `*.gif` ファイルへのすべてのリンクを更新して、イメージを表示できるようにしてください。

14 コード生成時に出力されるメッセージ

本章では、ASCET でのコード生成時に表示される可能性のあるワーニングおよびエラーメッセージと、そのエラーの原因となった誤りを訂正するためのヒントや方法を紹介します。エラーメッセージは、記述上の誤りのうちコード生成プロセスを終了させてしまうほど重大なものを指摘するもので、ワーニングはそれほど重大ではない誤りを通知するものです。ワーニングが出力された場合、コード生成プロセスは問題なく終了しますが、生成されるコードは意図したとおりに機能しない可能性があります。

14.1 コンポーネント

14.1.1 エラーメッセージ

`method <method_name> must be defined; need a return value`

説明：

戻り値を出力するメソッドがコンポーネント内で宣言されていますが、その戻り値にシーケンスコールが割り当てられていません。このメソッドは他のコンポーネントから呼び出される可能性があるので、シーケンスコールを指定する必要があります。

解決策：

シーケンスコールを編集し、その戻り値を出力するメソッドをシーケンス名として選択してください。シーケンス番号はそのメソッドの中で最大の番号でなければなりません。

`<method_name> has no argument <argument_name>`

説明：

メソッド `method_name` に定義されている演算が、別のメソッドに属する引数を使用しています。メソッドで使用できるのはローカルエレメントとグローバルエレメント、および自身の引数だけで、他のメソッドの引数を使用することはできません。

解決策：

シーケンスコールを変更するか、引数を別のエレメントにしてください。

`missing argument connection for method <method_name> at block <block_name>`

説明：

ブロック `block_name` でメソッド `method_name` が呼び出されていますが、すべての引数が接続されていません。つまり、引数の 1 つが指定されていません。演算子の場合には、メソッド名が空白のままになっています。

解決策：

引数をすべて接続してください。演算子の場合には、適切な数の引数を取る演算子を選択してください。

double sequence number <sequence_number> **for** <name>

説明：

プロセス、メソッド、アクション、あるいはコンディション name に、2 つのシーケンスコールが同じシーケンス番号 sequence_number で割り当てられています。

解決策：

どちらかのシーケンス番号を、name で使用していないシーケンス番号に変更してください。

return value does not belong to <name>

説明：

あるメソッドあるいはコンディションの戻り値が、戻り値を出力しない name というメソッドあるいはアクションのシーケンスコールに接続されています。戻り値のシーケンスコールは、その戻り値が定義されているメソッドあるいはコンディションに接続されなければなりません。

解決策：

戻り値のシーケンスコールのシーケンス名を、その戻り値が定義されているコンディションあるいはメソッドの名前に変更してください。

delay-free loop detected at <block_name> **block**

説明：

内部で何も処理が行わないループが作成されています。たとえば、ある演算子の戻り値が、その演算子への入力として直接与えられています。

解決策：

そのループ内にエレメントを挿入してください。

type mismatch: expected <type_A>, **got** <type_B>

説明：

type_A 型の引数が必要な箇所に type_B 型の引数が使用されていますが、type_B を type_A に型変換することはできません。たとえば、cont 型の引数が論理演算子に入力されています。この接続はおそらく間違っています。

解決策：

正しい型の引数を使用してください。

type mismatch: expected <type_A> [<name_A>}, **got** <type_B> [<name_B>]

説明：

type_B 型の name_B というエレメントが type_A 型の name_A という変数に割り当てられていますが、type_B を type_A に型変換することはできません。たとえば、cont 型のエレメントが logical 型の変数に割り当てられています。この接続はおそらく間違っています。

解決策：

エレメントの型を変更するか、接続を訂正してください。

return must be the last operation of <name>

説明：

name という戻り値を伴うメソッドあるいはコンディションに return 文が定義されていますが、この文のシーケンスコールのシーケンス番号が、そのメソッドあるいはコンディションに割り当てられている中で最大の番号になっていません。

解決策：

このシーケンスコールのシーケンス番号を、name というメソッドあるいはコンディションに属するすべてのシーケンスコールのシーケンス番号の中で最大の番号にしてください。

<then> **part of IF block must be specified**

説明：

IF ブロックが使用されていますが、THEN の部分がありません。

解決策：

THEN の部分を定義してください。THEN 部分には、コネクタを伴うシーケンスコールが少なくとも 1 つは定義されていなければなりません。

state machine needs start state

説明：

ステートマシンに開始ステートがありません。

解決策：

ステートマシンのステートの 1 つを開始ステートとして指定してください。

multiple prio <priority_number> **for trigger** <trigger_name> **in state** <state_name>

説明：

ステートマシンのステート state_name からの 2 つのトランジションが、同じトリガ trigger_name に同じ優先度 priority_number で定義されています。これではトランジションがユニークにならないので、このような記述は認められません。

解決策：

どちらかの優先度を変更して、同じステートから同じトリガで発生するトランジションの優先度がすべて互いに異なるようにしてください。

unbalanced number of start/stop atomic in <name>

説明：

name というメソッド、プロセス、コンディション、あるいはアクションには、アトミックシーケンスコール（割り込み不可のシーケンス）があります。しかし、開始マークと停止マークの数が一致していません。

解決策：

開始マークか停止マークを適宜に挿入／削除して、数のバランスをとってください。

Expected consistent datamodel for <element_name> **in** <Class_name>**. Element needs GET/SET direct access - please change attributes OR restore diagram**

（または、ESDL ／ C コードの場合は以下のメッセージ）

method "<Element_name>**"/"**<function_name>**" not defined as public in class "**<Class_name>**"**

説明：

クラス <Class_name> 内のエレメント／関数が、直接アクセス／パブリックに設定されていません。

解決策：

エレメントの場合は直接アクセス（Set ／ Get 機能）を有効にし、関数の場合はパブリックに設定してください。

14.1.2 ワーニング

<name> **not defined**

説明：

name というメソッド、プロセス、あるいはアクションが宣言されましたが、実際に定義されていません。name というシーケンス名のシーケンスコールがありません。これは、戻り値を出力しないメソッドだけに関係します。

解決策：

そのメソッド、プロセスあるいはアクションを定義するか、その宣言をコンポーネントのインターフェースから削除してください。

type mismatch with casting from <type_B> [<name_B>], **got** <type_A> [<name_A>]

説明：

type_B 型の name_B というエレメントが、type_A 型の name_A という変数に割り当てられ、type_B から type_A への型変換が行われました。たとえば、cont 型のエレメントが sdisc 型の変数に割り当てられました。

解決策：

エレメントの型を変更するか、接続を訂正してください。

argument <argument_name> **of method** <method_name> **not used**

説明：

メソッド method_name の定義の中で、そのメソッドの引数 argument_name が使われていません。

解決策：

引数 argument_name をメソッド定義の中で使用するか、メソッド定義から削除してください。

unreachable state <state_name>

説明：

ステートマシンの中に、開始ステートから行き着くことのできない state_name というステートがあります。つまり、このステートに至るトランジションがありません。

解決策：

このステートを削除するか、開始ステートから行き着くことができるようにしてください。

literal value <value> **does not fit type** <type> **- limited to** <range_value>

説明：

リテラルの値が、代入先の type 型の変数には大きすぎます。この変数に代入されるリテラルの値は自動的に値 range_value に制限されます。リテラルだけで構成される式の場合は、このようなことはありません。type 型は udisc あるいは sdisc で、それぞれ符号なしあるいは符号付きの 32 ビット整数を扱うことができます。

14.2 プロジェクト

14.2.1 エラーメッセージ

need binding for imported element <element_name>

説明：

インポートされたエレメントあるいはメッセージ element_name は、グローバルなエレメントあるいはメッセージに結びつけられていません。

解決策：

バインディングを（自動的にあるいは手操作で）調整してください。

application modes missing for task <task_name>

説明：

タスク task_name のアプリケーションモードが定義されていません。

解決策：

アプリケーションモードを定義するか、タスク task_name を削除してください。タスクを実行対象から除外するには、unused というアプリケーションモードを定義し、除外するタスクにそのアプリケーションモードを割り当てます。

14.2.2 ワーニング

no start application mode specified - using <opmode_name>

説明：

開始モードとして定義されているアプリケーションモードがありません。自動的に、アプリケーションモード opmode_name が開始モードとして定義されました。

解決策：

自動的に選ばれたモードが開始モードとして適切でない場合には、適切なモードを開始モードとして定義してください。

missing trigger event

説明：

オペレーティングシステムで指定されているイベントタスクの中に、トリガイベントが定義されていないものが 1 つあります。

解決策：

そのタスクのモードを変更するか、適切なトリガイベントをそのタスクに割り当ててください。

14.3 固定小数点コード生成

14.3.1 エラーメッセージ

`Integer interval [a,b] of variable <name> too large for implementation type`

説明：

モデルのインターバルから導き出された整数のインターバル [a,b] は、選択された実装データ型には大きすぎます。おそらく、このエレメントのインプリメンテーションは編集されていないか、実装データ型が整数型になっていません。

解決策：

エレメント name のインプリメンテーションを編集してください。

`Cannot generate fixed point code for the non-linear formula <formula_name> of variable <name>`

説明：

非線形の変換式 formula_name が name に割り当てられました。固定小数点コード生成がサポートするのは線形の変換式だけです。

解決策：

name に割り当てる変換式を変更するか、変換式 formula_name を線形の式になるように変更してください。

`Physical interval [a,b] of divisor contains zero`

説明：

ゼロによる除算が発生する可能性があるので、固定小数点コードを生成できません。ゼロによる除算は、インプリメンテーションのインターバルを無限大にしてしまいます。

解決策：

除数用の変数を挿入し、それに有意義なインプリメンテーションを定義してください（ここでは物理インターバルにゼロは使用できません）。

14.3.2 ワーニング

`formula in implementation for <name> not known in current project - using default`

説明：

エレメント name のインプリメンテーションについて、現在のプロジェクト定義内では変換式が認識されていません。変換式が割り当てられていません。変換式の代わりに恒等写像が使用されます。

解決策：

エレメント name のインプリメンテーションについて、現在のプロジェクトに妥当な変換式を使用してください。

Interval mismatch in assignment of <variable_name>: [a,b] := [c,d] **(will be limited)**

説明：

固定小数点コードジェネレータは、変数 variable_name の代入で矛盾が起こりうることを発見しました。この変数に代入される式の値はインターバル [c,d] 内の値で、このインターバルは、その変換式内のエレメント用に定義されているインターバルに基づいて、インターバル演算を通じて算出されます。しかし、インターバル [c,d] は変数 variable_name 用のインターバル [a,b] に含まれないので、オーバーフローが発生する可能性があります。このオーバーフローを回避するために、代入を実行する前に式の値が自動的に変数 variable_name のインターバル内の値に制限されます。ただし算術ループがある場合には、このワーニングを回避することはできません。

索引

す

せ

そ

た

ち

つ