大都市大震災軽減化特別プロジェクト
震災総合シミュレーションシステム (DDT-IS)
Simulator Module API Spec （案）

Simulation Kernel WG

version 0.00
(平成 15 年 2 月 10 日 29)

目 次

第1章　はじめに

1.1　目的

本ドキュメントは，文部科学省の大都市大震災軽減化特別プロジェクトで作成する分散シミュレーションシステム（震災総合シミュレーションシステム，以下 DDT-IS と略す）のシミュレータ開発者向けのアプリケーション・プログラムインタフェース（Simulator API，以下 DDT-IS S-API）の仕様を規定することを目的としている．

コア組織は，本仕様に基づいて作成されたソフトウエア群（ソフトウエア開発キット, 以下 SDK）を大都市大震災軽減化特別プロジェクトのシミュレータ開発者を含む一般に提供する予定である．SDK を利用することで，シミュレータ開発者は DDT-IS に組み込むことの可能なサブシミュレータを容易に開発することが可能となる．

1.2　特徴

シミュレーションシステム全体の概要を示すため，本項では，DDT-IS 全体の特徴を述べる．DDT-IS は以下の 6 つの特徴を持つ．

1. データ駆動型の分散協調シミュレーションシステム

   DDT-IS はデータ駆動型の分散シミュレーションシステムを作成するためのフレームワークを提供する．

   DDT-IS では，時系列情報を含む共有データを「共有時空間データ」(Shared Space-Time Data, 以下 SSTD) で扱う．すなわち，データを属性を持ったオブジェクトの時系列として管理する．

   SSTD で扱われるデータはシミュレーションシステム全体のシミュレーション結果を表し，すべてのサブシミュレータとカーネルで共有される．

2. サブシミュレータのプラグインが可能なモジュール構成

   DDT-IS は複数のサブシミュレータをシステム内に取り込むためのフレームワークを提供する．

   これまで扱わなかった自然現象を模擬するためのサブシミュレータが新たに開発された場合，既存のシミュレーションシステムに組み込んで利用することが可能となる．

3. 複数の異なる種類のサブシミュレータの協調動作

   組み込まれたサブシミュレータ同士が協調動作を行うことで，複合的な現象のシミュレーションが可能である．

例えば，地震シミュレータと交通シミュレータを協調動作されることで，災害時の交通状況を模擬することが可能となる．

4. 「エージェント」との協調動作

   自然現象を模擬するサブシミュレータに加え，人間の活動を模擬するエージェントを導入し，災害対応計画の評価等が行える．

5. 地域分散シミュレーション

   広域のシミュレーションを効率良く行うため，シミュレーション空間を地域に分割して，複数台の計算機上で分散シミュレーションを行うことが可能である．この場合，サブシミュレータは分割毎にそれぞれ実行され，サブシミュレータがシミュレーションを担当する分割に含まれるオブジェクトと，その周囲のシミュレーションの入力として必要な範囲にあるオブジェクトとが，SSTD としてサブシミュレータへ提供される．

6. ロールバック処理

   実世界の観測データや，行動決定のやり直しなどによるシミュレーションのロールバック処理（後戻り処理）をサポートする．この処理によって，何度もシミュレーションの条件を変更し，試行錯誤を繰り返すことが可能となる．

1.3　DDT-IS アーキテクチャ

図 1.1 に DDT-IS のアーキテクチャを示す．

DDT-IS はカーネルと呼ばれるモジュールを中心に，モジュール構成を取る．

- カーネル

  DDT-IS の主要コンポーネントで，時刻管理および分散データ管理等を行う．

- サブシミュレータ

  自然災害等の物理現象のシミュレーションを行うコンポーネント．

- エージェント

  人間の行動などのシミュレーションを行う．

- ビューア

  シミュレーション結果を視覚的に人間に提示する．

- 初期環境設定モジュール

  外部地理データを読み込んで初期化を行う．

DDT-IS S-API 仕様に従って作成されたシミュレータを，DDT-IS 準拠サブシミュレータ（以下，単にサブシミュレータ）と呼ぶ．シミュレータ作成者は SDK を利用し，S-API 仕様に従ってシミュレーションアルゴリズムを記述しプログラムを作成することで，他のサブシミュレータと協調動作するサブシミュレータを作成することが可能となる．

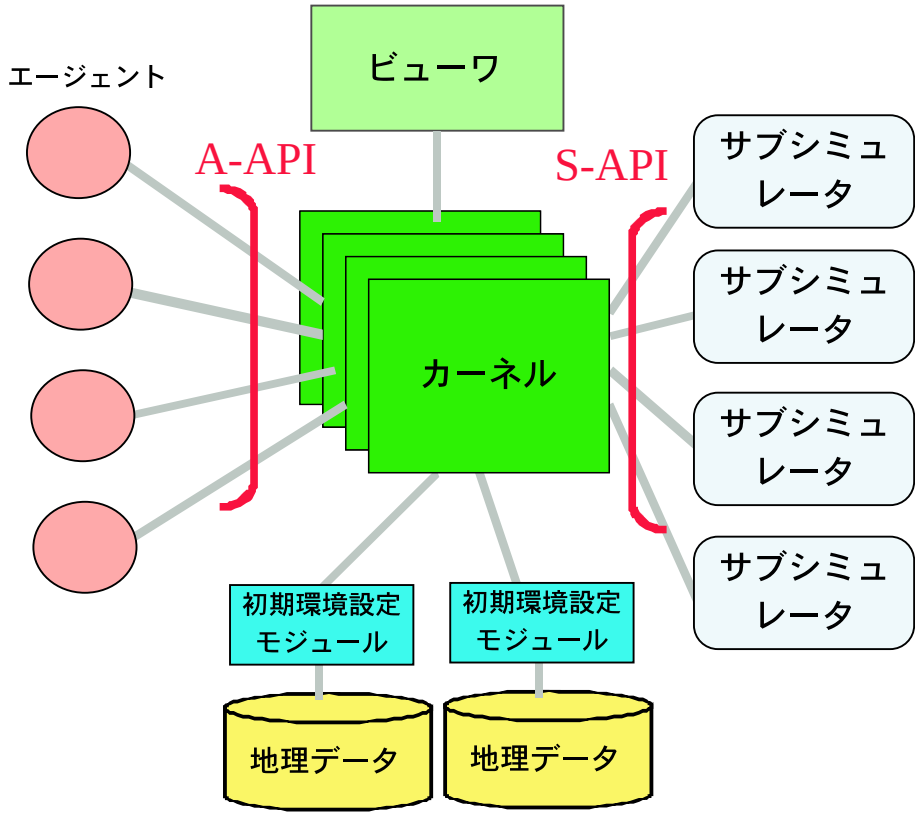

図 1.1: DDT-IS のアーキテクチャ

1.4　地理データ

1.4.1　外部地理データの利用方法

シミュレーションの基礎となる外部地理データは初期環境設定モジュールから与えられる．初期環境設定モジュールでは外部地理データを読み込み，SSTD の内部形式に展開しカーネルに提供する．

外部地理データの符号化方法の違いは，初期環境設定モジュールで吸収する．複数形式の地理データに対応するために，複数の初期環境設定モジュールを提供する予定である．

サブシミュレータから見た場合，SSTD はカーネルで一括管理され，S-API を経由して初期データとして提供される．S-API は共有時空間情報に関わる属性データの構造を規定しない．

1.4.2　地理情報に関わる演算処理

距離や面積の計算，地理近傍の検索等の地理情報に関する計算処理は S-API では提供しない．S-API とは別のライブラリ (GI-API) として提供する予定である．

1.4.3　SSTD における標準地理データモデル

シミュレータ作成者は他のサブシミュレータとデータを共有するため，SSTD において標準地理データモデルを利用する必要がある．SSTD の基礎となる標準地理データのモデルを付属資料 1 に示す．

本モデルは，RoboCupRescue version 0 シミュレータ向けに設計されたものを基にしており，交通流，火災延焼，建物倒壊，道路閉塞シミュレータに対応したモデルとなっている．

本モデルは初期ドラフトに位置づけられており，今後，大大特で作成されるシミュレータが増加するに従い，シミュレータ開発者との意見交換を通じて，適宜モデルの追加修正を行う予定である．

1.5　S-API の提供方法

S-API は Java のクラスライブラリおよびプリプロセッサ（spacegen）として提供される【/（図 1.2）．spacegen は，標準地理データモデルが記述されたスケルトン定義ファイルと，サブシミュレータがアクセスするオブジェクトや属性を宣言したスペース定義ファイルとを入力に取り，Java のクラスを出力する．サブシミュレータは，この出力されたクラスを利用して実装される■ [1] 小藤】．

互換性およびポータビリティの観点から，新規に開発されるサブシミュレータは Java 言語を利用することが推奨される．

[1] [Comment by 小藤]

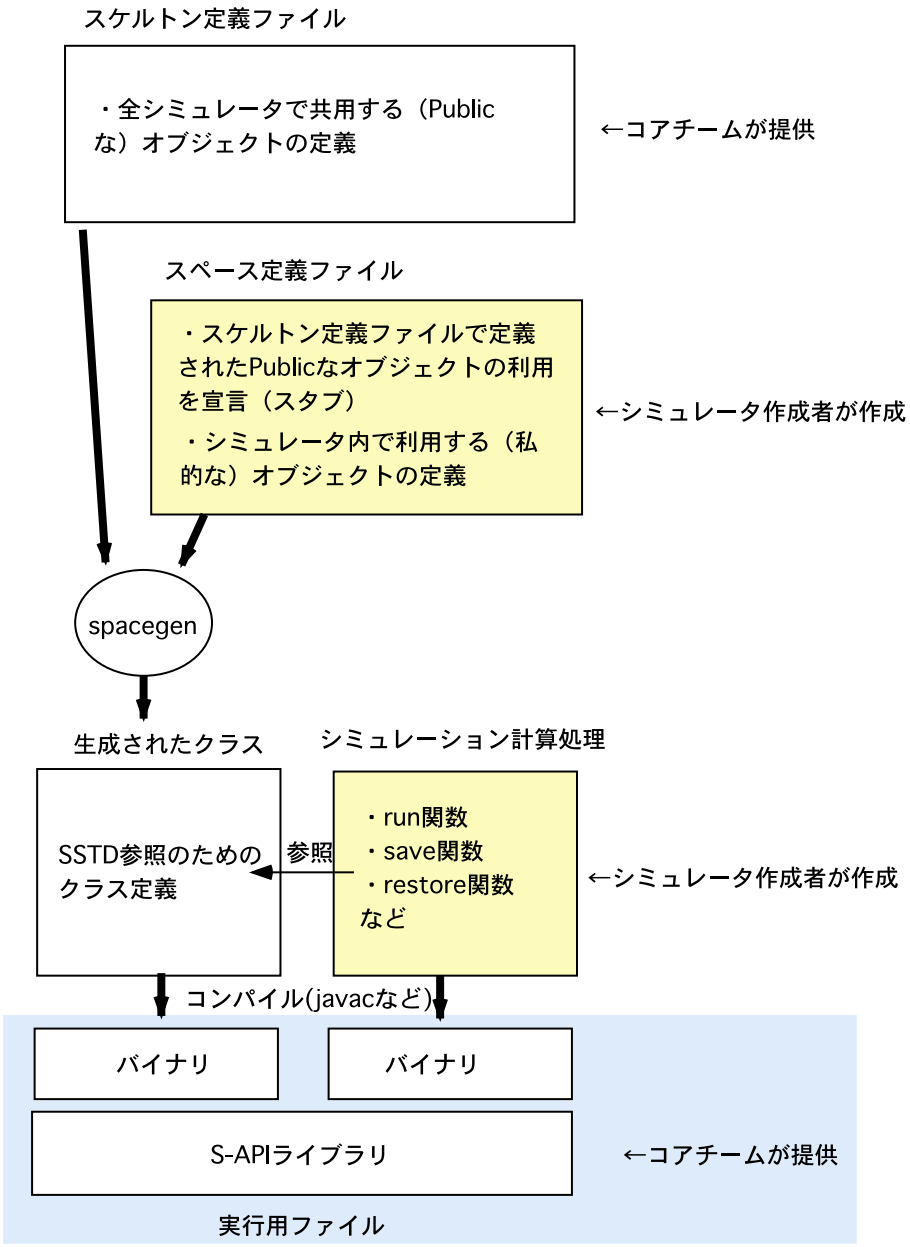
スケルトン定義ファイル
・全シミュレータで共用する（Public
な）オブジェクトの定義
←コアチームが提供
スペース定義ファイル
・スケルトン定義ファイルで定義
されたPublicなオブジェクトの利用
を宣言（スタブ）
・シミュレータ内で利用する（私
的な）オブジェクトの定義
←シミュレータ作成者が作成
spacegen
生成されたクラス
シミュレーション計算処理
SSTD参照のための
クラス定義
参照
・run関数
・save関数
・restore関数
など
←シミュレータ作成者が作成
コンパイル(javacなど)
バイナリ
バイナリ
S-APIライブラリ
←コアチームが提供
実行用ファイル

図 1.2: S-API

1.6　プログラム開発上の注意点

シミュレータ開発者は以下の点に注意してプログラムを開発する必要がある．

1.6.1　SSTD における標準地理データモデルの利用

すでに 1.4.3 でも述べた通り，シミュレータ作成者は他のサブシミュレータとデータを共有するために，SSTD において標準地理データモデルを利用する必要がある．

1.6.2　ロールバックのためのセーブ・リストア処理

DDT-IS は，実世界の観測データや，行動決定のやり直しなどによるシミュレーションのロールバック処理（後戻り処理）をサポートする予定である．DDT-IS に適合したシミュレータはシミュレーションシステムからの要求に応じて，以下の処理を行う必要がある．

- 状態のセーブ処理

  シミュレーション状態を保存し，要求に応じて再度その時の状態に復帰できるようにする．

- 状態のリストア処理

  セーブ処理で保存した状態に復帰し，シミュレーションを再開する．

1.6.3　プログラムからの SSTD の扱い

シミュレーションシステム全体のシミュレーション結果は，他のサブシミュレータや外部観測値などの値に基づいて決められる．従って，サブシミュレータのシミュレーション結果は，SSTD に保持されるシステム全体のシミュレーション結果にはならない場合がある．

1.6.4　ロールバック耐性の実現

ロールバック処理を実現するため，サブシミュレータは「ロールバック耐性」を持つことが必要とされる．すなわち，シミュレーション計算は任意の時刻において，それ以前の時系列情報に依存しないで計算を行うこともできるようになっていなければならない．

通常は時系列情報を利用してもよいが，ほかのサブシミュレータの結果や，外部の観測データ等によって，遡った任意の時刻を自分が計算したものとは別の状態にされてしまうという，ロールバックの可能性があるということである．もちろん，その場合でも，ロールバック時刻以前の時系列情報を利用することは (それが妥当であれば) 可能である．過去の時系列情報を使うためには，ロールバックに対応した履歴管理を行っておく必要がある．

1.7　用語

ここでは，本仕様書で用いる用語を改めて定義する．表 1.1 に用語の一覧と意味を示す．

表 1.1: 本仕様書で使われる用語

用語	意味
DDT-IS	震災総合シミュレーションシステム.
SDK	ソフトウエア開発キット
カーネル	DDT-IS の主要コンポーネントの一つで，時刻管理および分散データ管理を行う.
サブシミュレータ	DDT-IS の一部として，物理現象のシミュレーションを行うコンポーネント.
エージェント	DDT-IS の一部として，人間の行動などのシミュレーションを行うコンポーネント.
ビューア	DDT-IS のシミュレーション結果を視覚的に人間に提示するためのコンポーネント.
初期環境設定モジュール	外部地理データを読み込んで初期化を行うモジュール.
共有時空間データ (SSTD, Shared Space-Time Data)	サブシミュレータによって共有されるデータ. 【/カーネルによって管理される．■ [2] 小藤】
標準地理データモデル	DDT-IS で SSTD として利用するデータモデルのこと．サブシミュレータは必ず標準地理データモデルを利用する必要がある.
カーネルサービス	カーネルがサブシミュレータに対し提供するサービス．時間管理機能，共有データ管理機能，パラメータ管理機能など.
DDT-IS S-API	震災総合シミュレーションシステム向けのシミュレータ開発用アプリケーション・プログラムインタフェース．シミュレータは S-API を経由してカーネルサービスを利用する.
スペース定義ファイル	SSTD 上のオブジェクトや属性へ，Java のオブジェクトやメソッドによってアクセスできるようにするスタブを定義するファイル.
sync メソッド	カーネルサービスの中で，時間管理を行う関数．他のシミュレータと同期をとるために使われる．この関数を呼ぶことで SSTD のオブジェクトやその属性値が時間進行して確定する.
イベント	SSTD のオブジェクトの属性値の変化.
ロールバック処理	実世界の観測データをシミュレーションに反映したり，行動決定のやり直しを行うため，シミュレーション上の時刻を戻して，パラメータを変更後，再度シミュレーションを実行する処理.
ロールバック耐性	ロールバック処理を行うための必須条件．それ以前の時系列情報に依存せずにシミュレーション計算を行えるようにしておくこと．あるいは，履歴情報をどの時間にも戻れるような構造にしておくこと.
シミュレーション時刻	シミュレーションされる世界における時刻
シミュレーション時間	シミュレーションされる世界における時間

第2章　基本機能

本章では，DDT-IS で用いるサブシミュレータを作成するために必要な基礎的な知識を簡単に説明する．

DDT-IS で用いるサブシミュレータを作成するためには，以下の準備をする必要がある．

- スペース定義ファイル (2.3 節，3 節) の文法に従い，そのサブシミュレータが扱うデータの構造を記述する．このスペース定義ファイルはサブシミュレータ毎に用意する．
- スペース定義ファイルはスペースジェネレータ（**spacegen**）と呼ばれるコンパイラにより Java のプログラムに変換される．このプログラムには，サブシミュレータの基底クラスとなるべきクラス（*SubSimulatorBase* と書くことにする）が含まれる．また，SSTD を扱うための多くのクラスが含まれる．
- *SubSimulatorBase* のサブクラスとして，サブシミュレータを実装する (2.1 節)．

本節では，まず 2.1 節 において標準となるプログラムスタイルについて述べる．続いて 2.2 節においてサブシミュレータの同期機構である sync メソッドの動作について解説する．最期に 2.3 節ではサブシミュレータが参照するオブジェクトのカテゴリおよび属性を宣言するスペース定義ファイルと，サブシミュレータ内におけるデータ管理のためのメソッドの解説を行う．

2.1　標準プログラムスタイル

ここでは，DDT-IS で用いるサブシミュレータの標準的なプログラムスタイルを示す．S-API はこのスタイルを基準に設計されているが，すべてのサブシミュレータに同じスタイルを強制するものではない．

各サブシミュレータには，`run()` と `save()` の 2 つのメソッドが定義されなければならない．`run()` はシミュレーションを行う本体であり，DDT-IS が提供する同期メカニズム，データ共有アクセス等の機能を用いて担当するデータの時間変化を計算しなければならない．`save()` は 2.1.4 節で述べるセーブ機能を実現するもので，あるシミュレーション時刻のサブシミュレータ固有の状態を，各サブシミュレータにおいて再開が可能な形でファイルに保存する機能を提供する必要がある．

以下では，セーブされたデータを用いずにシミュレーションを開始することを**シミュレーションの新規開始**あるいは単純に**新規開始** と呼び，セーブによって保存された途中経過のデータからシミュレーションを開始することを**シミュレーションの再開**，あるいは単純に **再開** と呼ぶ．

標準的なサブシミュレータの `run()` は，

- 初期化部:
  サブシミュレータの初期化を行い，シミュレーションの新規開始の場合はデータの初期化を，再開の場合はセーブされたデータの復元を行う．さらに，他のサブシミュレータやシステムの準備ができるのを待つ．

- メインループ部:
  他のサブシミュレータと同期をとりつつ，シミュレーションの計算を行う．

- 後処理部:
  サブシミュレータの終了処理を行う．

からなる．以下にそのプログラムテンプレートを示す．

```
void run(RestoreInformation restoreInfo) {
  /* <<<初期化部>>> */
    :

  /* <<<メインループ部>>> */
    :

  /* <<<後処理部>>> */
    :
}
```

run() の第 1 引数の *restoreInfo* はシミュレーション再開時の情報を格納したオブジェクトである．これから行われるセッションがシミュレーションの新規開始である場合，*restoreInfo* には null が渡される．また，セッションがシミュレーションの再開の場合は，*restoreInfo* にはシミュレーションの途中経過を復元するための情報が格納されている．よって，新規開始と再開で処理が異る場合，*restoreInfo* が null かそうでないかによって条件分岐する必要がある．

以下の節ではこのプログラムテンプレートに従い，run メソッドにおける各部で実現すべき機能の詳細を説明する．また，save() で実現すべき機能については 2.1.4 節で詳述する．

2.1.1　run メソッドの初期化部

run メソッドの初期化部ではシミュレーションを新規開始，あるいは再開するための準備処理を行う．準備処理は，シミュレーションの新規開始と再開によって多少処理が異る．シミュレーションの新規開始，再開の区別は run() の第 1 引数 *restoreInfo* が null かどうかで判別する．

準備処理は waitKickOff() の呼び出しをはさんで以下の 2 つのフェーズに分かれる．

I.1 **初期化フェーズ:**
サブシミュレータ固有の初期化を行う．具体的には以下のような処理を行う．

- 実行時パラメータの取得:
  カーネルサービスの一つとして，実行時パラメータデータベースがある．これはシミュレーションのタイムステップなどシミュレーションにおける各種パラメータなどを一括管理するメカニズムである．getParameter(*key*) により，*key* に対応したパラメタを取得することができる．

- SSTD 初期値設定:
  サブシミュレータが SSTD の初期値代入の役割を担っている場合，SSTD の初期値代入 (初期オブジェクトの生成と各公開属性の初期化) を行う．各サブシミュレータはスペース定義ファイルで宣言したカテゴリのオブジェクトスタブ (*object*) を生成 (`new`) した後，`createObjectsAt(`*objects*`,`*createdTime*`)` により SSTD に時刻指定 (*createdTime*) でオブジェクトの生成を伝える．また，生成したオブジェクト (*object*) の各属性 (*property*) に対し，*object*`.set`*Property*`At(`*value*`,`*setTime*`)` により時間指定で値を与えることができる．ここで与えられた値は `waitKickOff()` の呼び出し時点で SSTD に反映される．

I.2 **初期状態取得フェース:**
SSTD からサブシミュレータが参照するデータを取得し，シミュレーションのメインループに入る準備をする．具体的には以下のような処理を行う．

- 初期化された SSTD のうち，サブシミュレータが必要とするデータの初期状態を取得し，内部で利用するデータ構造等に格納する．サブシミュレータは `getObjects()` により参照するカテゴリ[1]のオブジェクトのリストを取得できる．通常はこのオブジェクトリストに従い，内部参照用のデータベースを構築する．
- シミュレーションの再開の場合は，セーブされた状態を読み込み，セーブ時の状態を復元する．*restoreInfo*`.getInputStream()` はセーブ処理においてサブシミュレータが保存したデータを読み出す入力ストリームを返す．

これらの 2 つのフェーズは `waitKickOff()` というメソッドの呼び出しにより分けられる．このメソッドはサブシミュレータの初期化フェーズが終了したことをカーネルに通知し，ほかのサブシミュレータの初期化フェーズの終了を待つ．よって，`waitKickOff()` の後では全サブシミュレータが起動しており，各サブシミュレータにおける初期化フェーズが完了して SSTD の準備が整っていることが保証される．

[1] 各サブシミュレータは参照するオブジェクトのカテゴリをスペース定義ファイルにおいて宣言する (3 節節).

```
void run(RestoreInformation restoreInfo) {
  /* <<<初期化部>>> */

    /* <<初期化フェーズ>> */

    /* カーネルサービスオブジェクトの取得 */
    KernelService kernel = getKernelService() ;

    /* 実行時パラメータの取得 */
    Object someParameter = getParameter("some key");
    :
    /* SSTD の初期値代入 */
    if(restoreInfo == null) { /* シミュレーションの新規開始の場合 */
      /* SSTD に新規オブジェクトを生成 */
      Stub newObject = new Stub() ;
      newObject.setPropertyAt(initValue, initTime);
      :
      createObjectsAt(objectList,initTime) ;
      :
    }

    /* 初期化フェーズ完了 */
    waitKickOff();

    /* <<初期状態取得フェーズ>> */

    /*  SSTD の初期状態の取得 */
    List  objects = getObjects();

    /* セーブ時の状態の復元 */
    if(restoreInfo != null) { /* シミュレーションの再開の場合 */
      DataInputStream istr = restoreInfo.getInputStream() ;
      :
    }

    /* 必要に応じて  objects をサブシミュレータ内部で用いるデータ構造等に格納 */
    :

  /* <<<メインループ部>>> */
    :
  /* <<<後処理部>>> */
    :
}
```

2.1.2　run メソッドのメインループ部

メインループ部においては，シミュレーション計算を繰り返す．各繰り返しは以下のステップからなる．

M.1 sync メソッド (2.2 節) により，シミュレーションクロックを進める．すなわち，前回の sync メソッド以降に行ったシミュレーションの結果を SSTD に記録し，自分のシミュレーションがある程度のシミュレーション時刻まで進行したことをシステムへ通知して，他のサブシミュレータが同じ時刻までシミュレーションを終了するのを待つ．

M.2 現在のシミュレーションクロックにおける SSTD を読み込み，サブシミュレータ内部で用いるデータ構造へ反映する．

M.3 シミュレーションの計算を行う．

M.4 シミュレーション結果を各オブジェクト (*object*) の各属性 (*property*) に *object*.set*Property*At(*value*,*time*) を使って反映させる．ここで与えられた値は次回の sync メソッドにより SSTD に反映される．

DDT-IS では syntAt()，syntByEvent()，syntByEventUpto() の 3 種類の sync メソッドを用意している．syntAt() は時間指定あるいは定期更新型のシミュレーションを行う場合に用いられ，シミュレーションの実行の同期を時間指定で行う．syntByEvent() はイベント[2]の発生を待って同期を行う場合に用いられる．syntByEventUpto() は syntAt() と syntByEvent() の複合であり，イベントと指定時間のうち，早く生じた事象によりシミュレーションの同期を行う．

以下では時間指定型およびイベント待機型のサブシミュレータの標準プログラムスタイルを示す．

定期更新型および時間指定駆動型サブシミュレータ

以下に示す例は，一定のタイムステップ (*timeStep*) 毎にシミュレーションを行い，SSTD のあるオブジェクト *fooObj* の属性 *barProp* の値を *newValue* を変更する例である．*defaultTimeStep* はサブシミュレータが適当だと考える，シミュレーションのタイムステップである．

[2] イベントは属性の変化として定義される．2.2.5 節参照．

```
void run(RestoreInformation restoreInfo) {
  /* <<<初期化部>>> */
    KernelService kernel = getKernelService() ;
    :
    /* タイムステップを初期化する．実行時パラメータで指定されていれば，その値を使うのが
良い */
    Time timeStep = kernel.getTimeParameter("time_step", defaultTimeStep);

    /* ルックアヘッドの設定．実行時パラメータで指定されていれば，その値を使うのが良い */
    Time lookahead = kernel.getLongParameter("lookahead", defaultLookahead);

    /* 開始時刻の取得．再開時には，前回の開始時刻を取得することが望ましい．*/
    Time nextClock ;
    if(restoreInfo == null) { /* 新規開始時 */
       nextClock = kernel.getTimeParameter("start_time", defaultStartTime);
    } else { /* 再開時 */
       nextClock = restoreInfo.currentTime() ;
    }
    :
  /* <<<メインループ部>>> */
    while (true) {
        try { /* シミュレーションクロックを進める */
            kernel.syncAt(nextClock,lookahead);
        } catch (SessionFinishedException x) {
            /* シミュレーション終了
             * 後処理がなければ，catch せずにメソッドを終了しても良い
             */
            break;
        }

        /*  SSTD 中のオブジェクトの集合を得る */
        List objects = kernel.getObjects();
        /* あるいは，kernel.getCreatedObjects() や
         * kernel.getRemovedObjects() を用いることで，
         * 前回のシミュレーションステップからの増減分のみを得ることもできる．
         */

        Iterator it = objects.iterator();
        while (it.hasNext()) {
            SimulationObject o = (SimulationObject) it.next();
            /* 内部モデルへ反映 */
              :
        }

        /* シミュレーションしたい時刻 */
        nextClock = kernel.currentTime().add(timeStep);

        /* nextClock における値をシミュレーションする */
          :
        /* シミュレーション結果をシステムへ通知 */
        fooObj.setBarPropAt(newValue, nextClock);
    }

  /* <<<後処理部>>> */
    :
}
```

なお，この例において，*timeStep* は固定である必要はなく，値が正の範囲で自由な値を取るこ

とができる．

イベント 待機型サブシミュレータ

以下に示すコードは，指定したイベントを待ち，それによって駆動するシミュレータの例である．この例では，イベントは属性のリスト *propList* で指定され，SSTD のオブジェクト *fooObj* の属性 *barProp* を変更する．

```
void run(RestoreInformation restoreInfo) {
  /* <<<初期化部>>> */
    KernelService kernel = getKernelService() ;
    :
    /* ルックアヘッドの設定.
     * 実行時パラメータで指定されていれば，その値を使うのが良い
     */
    long lookahead = kernel.getLongParameter("lookahead", defaultLookahead);
    /* イベントリストの設定.
     * シミュレーションの再開の場合，前回の sync のイベントリストを取得する.
     */
    List propList ;
    if(restoreInfo == null) { /* 新規開始時 */
      propList = ... ; // シミュレータ独自のイベントリスト
    } else {
      propList = restoreInfo.savedSyncEventList() ;
    }
    :

  /* <<<メインループ部>>> */
    while (true) {
        EventSyncResult syncResult ;
        try {
          /* イベント（属性値の変化）が発生するまでシミュレーションクロックを進める */
          syncResult = kernel.syncByEvent(propList);
        } catch (SessionFinishedException x) {
            /* シミュレーション終了
             * 後処理がなければ，catch せずにメソッドを終了しても良い
             */
            break;
        }

        /* 属性値が変化したオブジェクトの集合を得る */
        List objects = syncResult.getChagedObjects();

        Iterator it = objects.iterator();
        while (it.hasNext()) {
            SimulationObject o = (SimulationObject) it.next();
            /* 内部モデルへ反映 */
              :
        }

        /* シミュレーションしたい時刻
         * kernel.currentTime() はすでにシミュレーションが終了した
         * 時刻なのでそれより未来の時刻を指定する.
         */
        Time nextClock = kernel.currentTime().add(timeStep);

        /* nextClock における値をシミュレーションする */
          :

        /* シミュレーション結果をシステムへ通知 */
        fooObj.setBarPropAt(newValue, nextClock);
    }

  /* <<<後処理部>>> */
    :
}
```

2.1.3　run メソッドの後処理部

後処理部においては，サブシミュレータで何か必要な終了処理があれば，それを行う．

2.1.4　save メソッドとシミュレーションの再開

save メソッドは，シミュレーションの状況を保存する必要がある場合に，sync メソッドの実行中に，カーネルから呼び出される (図 2.2 参照)．保存された状況を復元することで，一度実行したシミュレーションをやり直すことや，シミュレーションシステムを一度シャットダウンした後でも，その続きからシミュレーションを行うことができる．

保存すべきデータは，SSTD 以外にそのサブシミュレータがローカルに使用しているデータに限られる．SSTD はシステムによって保存・復元されるため，サブシミュレータが内部的に使用しているデータ構造が SSTD から復元可能であれば，サブシミュレータは save メソッドを実装する必要はない．そうでない場合，save で復元に必要な情報を記録しなければならない．

```
public void save(DataOutputStream output) throws IOException {
  /* <<<セーブ処理>>> */
    output.writeInt(localData) ;
    :
}
```

output は，保存する情報を出力するストリームである．このストリームに保存されたデータは，再開時に run() に渡される *restoreInfo* 引数の getInputStream() メソッドで渡されるストリームから読み出すことができる．

2.1.5　run および save におけるカーネルとサブシミュレータのタイムチャート

以上で述べた run の標準プログラムスタイルにおけるカーネル・サブシミュレータ間の相互作用をタイムチャートで示すと，図 2.1，図 2.2，図 2.3，図 2.4 となる．

図 2.1 は新規開始時のタイムチャートであり，サブシミュレータは run による起動後，初期化フェーズ，初期状態取得フェーズを経たあとメインループに入り，sync の呼び出しとシミュレーション計算からなるシミュレーションサイクルを繰り返す．シミュレーションの終了時には，メインループの繰り返しの最中に終了の要求が発生した場合，図 2.2 に示すように全サブシミュレータが sync メソッドの呼び出し中になるのを待って終了の例外 (SessionFinishedException) を発生させ，各サブシミュレータの終了処理を行わせる．全サブシミュレータが終了するのを待ってカーネルが終了処理を行う．

メインループの繰り返しの最中にセーブの要求が入った場合には図 2.3 に示すように，全サブシミュレータが sync メソッドを呼び出している最中にまずカーネルにおいて SSTD の保存が行われ，続いて各サブシミュレータの状態のセーブ処理が行われる．全サブシミュレータのセーブ処理が完了すると通常のシミュレーションのサイクルに戻る．ここで保存されたデータを用いてシミュレーションを再開する場合，図 2.4 に従ってセーブ時の状態を復元し，メインループで計算を再開する．この場合，各サブシミュレータでメインループに入る時点は，図 2.3 において「再開時の復帰時点」で示された時点とみなされるものとする．よって再開時の run の初期化部は図 2.3 の

「再開時の復帰時点」の状態を復元するようプログラムされなければならない

2.2　時間管理

各サブシミュレーターは，時間指定型 (*kernel*.syncAt)，イベント待機型 (*kernel*.syncByEvent)，タイムアウト付イベント待機型 (*kernel*.syncByEventUpto) の 3 種類のメソッドで他のサブシミュレータと同期を取り，時間を進めることができる．この同期を取る 3 つのメソッドをまとめて sync メソッドと呼ぶ．

以下では，各関数を用いた時間の進め方およびイベントの定義に関して説明する．

2.2.1　現在時刻と固着時刻

各サブシミュレータにおいて，直前の同期 (sync メソッドの呼び出し) が完了した時点のシミュレーション時刻のことを**現在時刻** と呼ぶ．各サブシミュレータは，現在時刻の SSTD の情報しか得ることができない．また，SSTD 上においてデータの値を変更できる最も早い時刻を**固着時刻** と呼ぶ．言い換えれば，固着時刻より古い時刻 (< 固着時刻) のデータを書き換えることはできない．これら現在時刻および固着時刻は各々，*kernel*.currentTime() および *kernel*.anchorTime() で取得することができる．

同期時刻に対し各サブシミュレータがデータに変更を加えないことを保証する時間幅のことを**先行時間 (lookahead)** と呼ぶ．先行時間は sync メソッドの引数として指定され，Time.TIC と等しいかまたは大きい値でなければならない．

run のメインループ部において sync メソッドの完了後，次の sync メソッドの呼び出しの間は，固着時刻，現在時刻，先行時間の間に以下の関係が成り立つ．

$$\textbf{固着時刻} \quad = \quad \textbf{現在時刻} + \textbf{先行時間}$$

また，初期化部においては，現在時刻と固着時刻の変化は以下のように変化する．シミュレーションの新規開始時の場合，初期化フェーズにおいては現在時刻と固着時刻共に時刻の最小値 (Time.MIN_VALUE) をとる．初期状態取得フェーズに入ると固着時刻はシミュレーションの開始時刻[3]に，現在時刻は 開始時刻 – Time.TIC [4]に変化する．一方，シミュレーションの再開時の場合は，初期化フェーズ・初期状態取得フェーズを通して，再開データのセーブを行う前に完了した sync メソッド直後のシミュレーション時刻 (直前の sync 完了後の現在時刻) が現在時刻となり，セーブを行ったシミュレーション時刻が固着時刻となる．

現在時刻と固着時刻の変化をまとめると，以下のようになる．

[3] シミュレーションの開始時刻は，一般に *kernel*.getTimeParameter("start_time") で取得できる値と一致する．
[4] Time.TIC は時間の最小単位である．

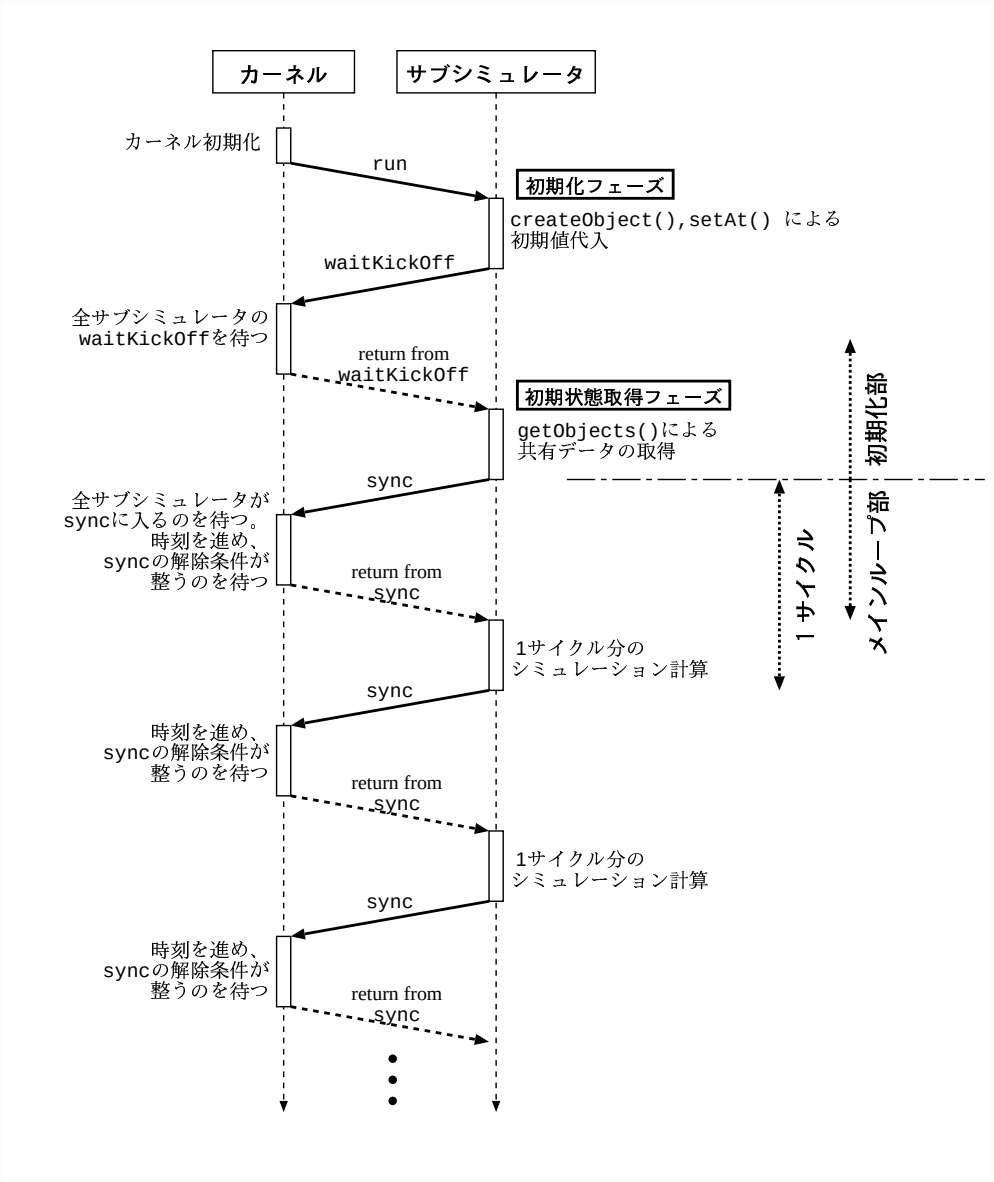

図 2.1: 新規開始時のカーネル・サブシミュレータ間の相互作用のタイミング図

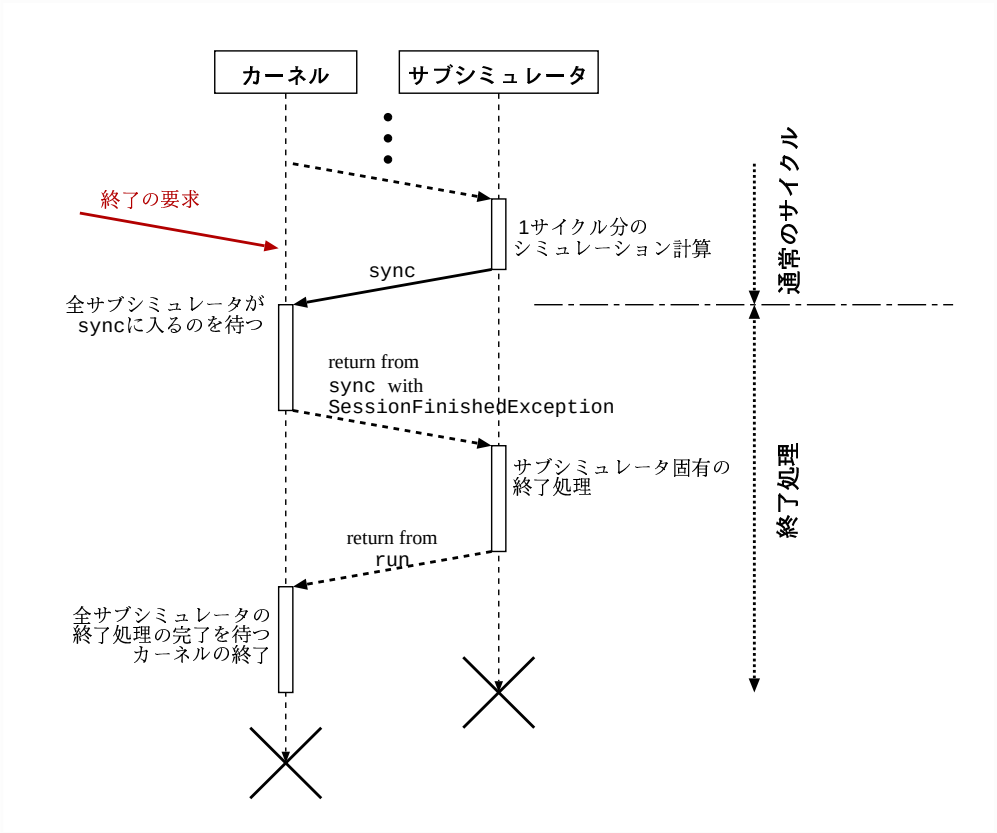

図 2.2: 終了処理部のカーネル・サブシミュレータ間の相互作用のタイミング図

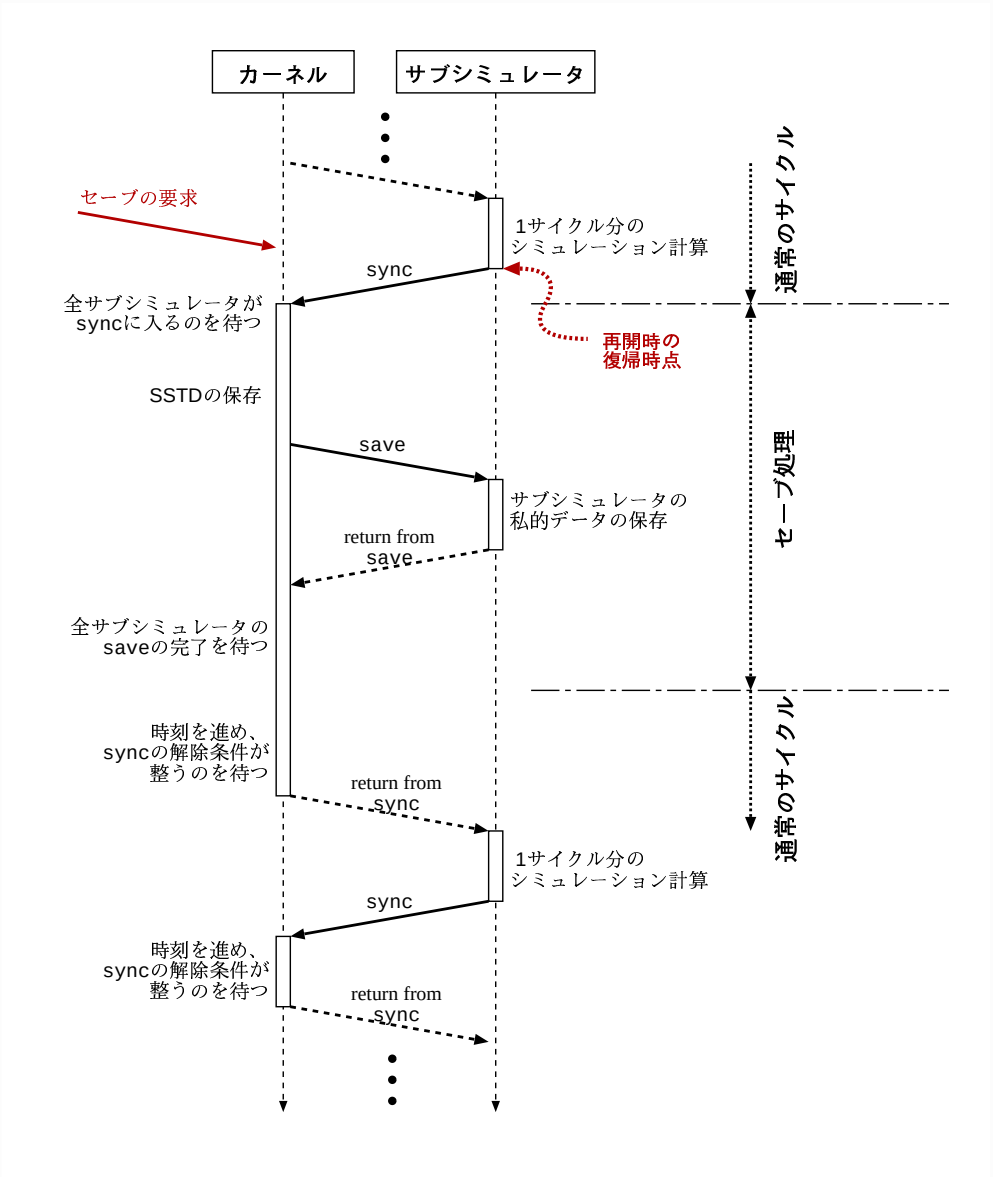

図 2.3: セーブ時のカーネル・サブシミュレータ間の相互作用のタイミング図

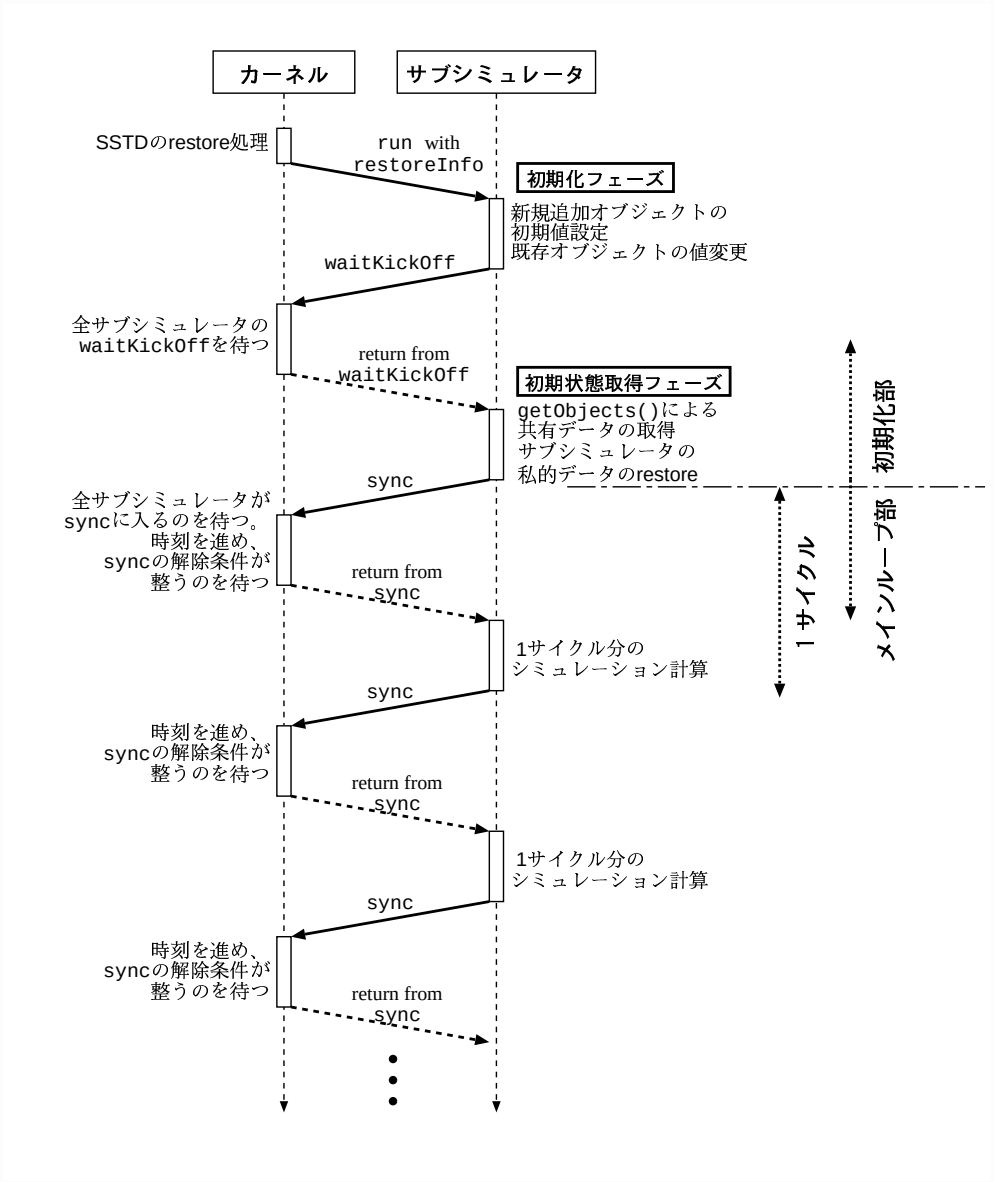

図 2.4: 再開時のカーネル・サブシミュレータ間の相互作用のタイミング図

新規開始/再開	フェーズ	現在時刻 `currentTime()`	固着時刻 `anchorTime()`
シミュレーションの新規開始時	初期化フェーズ	`Time.MIN_VALUE`	`Time.MIN_VALUE`
	初期状態取得フェーズ	開始時刻 − `Time.TIC`	開始時刻
	メインループ部	直前の sync メソッドが完了した時刻	現在時刻 + 先行時間
シミュレーションの再開時	初期化フェーズ	セーブの前に完了した sync 直後の現在時刻	再開時刻
	初期状態取得フェーズ	セーブの前に完了した sync 直後の現在時刻	再開時刻
	メインループ部	直前の sync メソッドが完了した時刻	現在時刻 + 先行時間

2.2.2　時間指定型同期

kernel.`syncAt`(*time*,*lookahead*) は，指定した時間に駆動するタイプのシミュレータの同期に用いられる．SSTD を定期的に更新するタイプもこの方式を用いる．具体的には以下の 2 つの処理を行う．

- 時刻 *time* 以前 (*time* を含む) について，そのサブシミュレータがシミュレーションを終了しており，その結果はすべて S-API を通じてシステムに通知済みであることを宣言する．
- `$in` または `$io` に指定した属性に関して，*time* 以前のシミュレーション時間において，他のサブシミュレータのシミュレーションが終了し，SSTD における値が決定されるまで，サブシミュレータの実行をブロックする．

2.2.3　イベント 待機型同期

kernel.`syncByEvent`(*propertyList*,*lookahead*) は，ほかのサブシミュレータにより引き起こされるイベントを待って駆動するタイプのシミュレータの同期に用いられる．具体的には，以下の 3 つの処理を行う．

- イベント (指定した属性の集合 *propertyList* の中の任意の属性の値の変化．2.2.5 節参照) が起きるまでの間に関して，そのサブシミュレータがシミュレーションを終了しており，その結果はすべて S-API を通じてシステムに通知済みであることを宣言する．
- 他のサブシミュレータのシミュレーションを，イベントが生じたかどうか判定するのに十分なだけ待つ．
- `$in` または `$io` に指定した属性に関して，イベントが発生した時刻までのシミュレーション

時間において，他のサブシミュレータのシミュレーションが終了し，SSTD における値が決定されるまで，サブシミュレータの実行をブロックする．

2.2.4　タイムアウト 付イベント 待機型同期

kernel.syncByEventUpto(*propertyList*，*deadline*,*lookahead*) は，*kernel*.syncAt() と *kernel*.syncByEvent() を合わせた機能を提供する．すなわち，指定した時刻とイベントのうち，先に生じた事象と同期する．具体的には，以下の 3 つの処理を行う．

- *deadline* 以前かつイベントが起きるまでの間に関して，そのサブシミュレータがシミュレーションを終了しており，その結果はすべて S-API を通じてシステムに通知済みであることを宣言する．
- 他のサブシミュレータのシミュレーションを，*deadline* 以前の間にイベントが生じたかどうか判定するのに十分なだけ待つ．
- $in または $io に指定した属性に関して，イベントが発生した時刻か *deadline* の小さいほうまでのシミュレーション時間において，他のサブシミュレータのシミュレーションが終了し，SSTD における値が決定されるまで，サブシミュレータの実行をブロックする．

2.2.5　イベント

イベントは属性値の変化として定義される．各シミュレータは，*kernel*.syncByEvent(*propertyList*) あるいは *kernel*.syncByEventUpto(*propertyList*,*deadline*) により監視する属性の集合を指定してイベントを待つことができる．*propertyList* は監視する属性の名前のリストである．

どのオブジェクトやどの属性が変化したかは，メソッドの戻り値 (*syncResult*) に対する次のメソッド呼び出しによって知ることができる．

- *syncResult*.getChangedObjects(): 監視している属性が変化したオブジェクトのリストを返す．
- *syncResult*.getChangedProperties(): 監視している属性のうち，変化した属性のリストを返す．

2.3　データ管理

2.3.1　SSTD の宣言

各サブシミュレータ開発者は【共有する時空間データを /SSTD のどのデータに対してアクセスするかを，■ [5] 小藤】スペース定義ファイル（3 節参照）に指定の形式で記述する必要がある．ス

[5] [Comment by 小藤]

ペース定義ファイルは **spacegen** により Java のクラスファイルに変換される[6]．このクラスファイルには，次のようなクラスが定義される．

- サブシミュレータのベースクラス:
  サブシミュレータの実装の基盤となるクラス．クラスの名前は スペース定義ファイルの *JavaFQN* により指定され，サブシミュレータ固有のものとなる．
- SSTD のオブジェクトへアクセスするためのスタブ:
  【SSTD の構造を表し/■[7] 小藤】，SSTD へのアクセスする手段を提供するクラス．クラス名は `$stub` に続く *Identifier* が用いられる．

SSTD 上ではデータはオブジェクトの集合として表され，各オブジェクトは属性の集合からなる．属性は，時間的に変化する値として表現される．

各オブジェクトはまた，何らかのカテゴリに分類される．スペース定義ファイルは，そのサブシミュレータが SSTD 上のデータのどのカテゴリと，そのカテゴリ内のどの属性を参照するか宣言する．これらのカテゴリおよび属性を「参照カテゴリ」「参照属性」と呼ぶことにする．

spacegen はスペース定義ファイルから，参照カテゴリのオブジェクトおよびその参照属性へのアクセスを管理するスタブを生成する．また，カーネルサービスは，SSTD 上で参照カテゴリに属するオブジェクトに関して，全リストを取得するメソッド (`getObjects()`)，新たに生成されたオブジェクトのリストを取得するメソッド (`getCreatedObjects()`)，削除されたオブジェクトのリストを取得するメソッド (`getRemovedObjects()`) を提供する．これらのメソッドにより，サブシミュレータはそのカテゴリに属する全オブジェクトを他のサブシミュレータと共有できる．

2.3.2　SSTD の登録と消去

SSTD に新しいオブジェクトを追加するには，そのオブジェクトをあらわすスタブクラスのインスタンスを `new` し，`createObjectsAt` メソッドを呼び出す．例えば，次のようになる．

```
Stub newObject = new Stub();
newObject.setPropertyAt(value, time);
:
kernel.createObjectsAt(Collections.singleton(newObject), time);
```

オブジェクトを削除するには，`removeObjectsAt` メソッドを呼び出す．例えば，次のようになる．

```
SimulationObject objectToRemove;
kernel.removeObjectsAt(Collections.singleton(objectToRemove), time);
```

2.3.3　SSTD の参照と変更

SSTD に含まれるオブジェクトを，`getObjects()` により取得することができる．`getObjects()` の戻り値は，シミュレーションクロックにおいて SSTD に含まれるオブジェクトをあらわすスタブのリストである．【/このリストには，スペース定義ファイルでアクセスすることを宣言した種

[6] Java 以外に対応した spacegen についても, コアチームでサポートすることを考えているが, 予定は確定していない.
[7] [Comment by 小藤]

類のオブジェクトだけが含まれる．■[8] 小藤】サブシミュレータは，このスタブを通じて，SSTD にアクセスする．

値の参照

属性値の参照は *object*.*property*() により行う．これは，シミュレーションクロックにおける属性の値を返す．*object* はスタブであり，*property* は属性の名前である．

値の変更

SSTD の変更は *object*.set*Property*At(*newValue*，*newTime*) により行う．*object* はスタブであり，*Property* は属性の名前の先頭の文字を大文字にしたものである．その属性の時刻 *newTime* における値が *newValue* であるとシミュレーションされたことを宣言する．

set*Property*At() を複数回呼び出すことで，複数の時刻の値を指定することができる．たとえば，以下のコードを考える．

```
kernel.syncAt(time) ;
obj.setPropAt(value1, time + 1) ;
obj.setPropAt(value2, time + 2) ;
```

これは，オブジェクト obj の属性 prop のシミュレーション結果が，時刻 time + 1 において value1, 時刻 time + 2 において value2 であることを示している．この例のように，set*Property*At で指定する時刻は，同じオブジェクト・同じ属性については，前回の呼び出しにおける時刻より大きい値でなければならない．

[8] [Comment by 小藤]

第3章　API仕様

S-API の仕様は，本章と，添付される API リファレンスによって定められる．API リファレンスではクラスやメソッド毎に仕様を説明し，本章ではクラス毎メソッド毎では説明しにくい仕様や，特に重要な仕様を説明する．

S-API は，決まったクラス階層やメソッドをもつクラスライブラリと，**スペースジェネレータ**（**spacegen**）と呼ばれるコンパイラによって出力される，入力に応じて異なるクラスで構成される．**spacegen** によって出力されるクラスの主な役割は，SSTD におけるオブジェクトへ，Java のネイティブのオブジェクトを介してアクセスすることを可能にすることである．

spacegen は，**スケルトン定義ファイル**と**スペース定義ファイル**を入力に取る．スケルトン定義ファイルはすべての種類のサブシミュレータで共通のファイルであり，異種サブシミュレータによって共有される地理データモデルを定義する．スペース定義ファイルは,【同種サブシミュレータ■[1] 小藤:イントロで用語定義すること】の間では共通だが【異種サブシミュレータ■[2] 小藤:同様】の間では異なるファイルであり，そのサブシミュレータが，スケルトン定義ファイルで定義される地理データモデルへどのようにアクセスするかを記述する．また，スペース定義ファイルは，サブシミュレータに固有の情報を表現できるように，地理データモデルの拡張を定義する．

スケルトン定義ファイルの文法は，図 3.1 の拡張 BNF で示される．スペース定義ファイルの文法は，図 3.2 の拡張 BNF で示される．

3.1　公的と私的

スケルトン定義ファイルやスペース定義ファイルで扱われる要素には，**公的**（**Public**）なものと**私的**（**Private**）なものがある．ある要素が公的である場合，その要素はすべてのサブシミュレータで共有される．ある要素が私的である場合，その要素は同種サブシミュレータの間では共有されるが，異種サブシミュレータの間では共有されない．スケルトン定義ファイルで定義される要素はすべて公的であり，スペース定義ファイルで定義される要素はすべて私的である．スケルトン定義ファイル，スペース定義ファイルのいずれにおいても，公的な要素を自由に参照することができる．スケルトン定義ファイルでは私的な要素を参照することはできないが，スペース定義ファイルではそのスペース定義ファイルで定義された私的な要素であれば参照することができる．

[1] [Comment by 小藤] イントロで用語定義すること
[2] [Comment by 小藤] 同様

3.2　オブジェクトと属性

サブシミュレータ間で共有される全データは SSTD に格納される．SSTD は**属性**を持った**オブジェクト**の集合としてモデル化される．各オブジェクトは 1 つの**カテゴリ**に属し，同じカテゴリに属するオブジェクトはカテゴリによって定まる一連の属性を持つ．カテゴリと属性には，公的なものと私的なものがある．公的なカテゴリに属するオブジェクトは，異種サブシミュレータによって共有することができる．私的なカテゴリに属するオブジェクトは，同種のサブシミュレータのみによって共有される．公的な属性は異種サブシミュレータによって共有することができ，私的な属性は同種のサブシミュレータのみによって共有される．私的なカテゴリに属するオブジェクトが，公的な属性を持つことは無い．スケルトン定義ファイルは，どのような公的なカテゴリが存在し，そのカテゴリに属するオブジェクトがどのような公的な属性を持つかを定義する．スペース定義ファイルは，どのような私的なカテゴリが存在し，そのカテゴリに属するオブジェクトがどのような私的な属性を持つかを定義する．スペース定義ファイルはまた，公的なカテゴリに属するオブジェクトが，どのような私的な属性を持つかも定義する．

SSTD 上のオブジェクトへのアクセスは，S-API ではスタブクラス（3.6 節）のインスタンスによって行われる．このインスタンスは，SSTD 上のオブジェクト毎に用意され，属性にアクセスするためのメソッドを提供する．スタブクラスのインスタンスは，`getObjects` メソッドによって得ることができる．このメソッドは，サブシミュレータの同期時刻（3.7 節）において SSTD 上に存在するオブジェクトのうち，スペース定義ファイルで *PublicCategoryName* あるいは *PrivateCategoryName* に記述されたカテゴリに属するオブジェクトに対応するスタブクラスのインスタンスのリストを返す．

属性の値は，シミュレーション時間に対する時系列として保持される．属性の値の時系列はシミュレーションの進行に伴って，徐々に確定していく．あるシミュレーション時刻における属性の値が確定しているとき，その属性の値はそのシミュレーション時刻において**固定されている**と表現する．あるシミュレーション時刻における属性値が固定されたならば，同じセッション（3.9 節）の間はその時刻における属性値は変化せず固定されたままである．固定された属性の値は，**未定義**であるか，3.3 節で説明される値を持つかのいずれかである．通常，サブシミュレータや初期状態設定モジュールによって属性の値が一度も設定されなかった場合に，その属性の値は未定義であり，最初に設定された以降のシミュレーション時刻において，その属性の値は未定義ではなくなる．

SSTD 上のオブジェクトは，シミュレーション時間上の**存在期間**を持つ．存在期間も属性の時系列と同様に，シミュレーションの進行に応じて徐々に確定していく．オブジェクトの存在期間は，そのオブジェクトがシミュレーションされる空間に現れてから，無くなるまでの間である．オブジェクトがシミュレーションされる空間の外に移動して，再び戻ってきた場合，そのオブジェクトの存在期間は，連続した 1 つの期間ではなくなる．

SSTD 上のオブジェクトはまた，**ID** を持つ．ID はオブジェクトに固有の値である．

3.3　型

属性は**型**を持つ．型は，その属性がどのような値を取りうるかを規定する．スケルトン定義ファイルおよびスペース定義ファイルにおいて，*Type* は型を表す．型は，以下で説明されるように，

Java における型に対応付けられる．

Type	Java の型	取りうる値
`$boolean`	`boolean`	ブーリアン値
`$byte`	`byte`	8 ビットバイナリ
`$int`	`int`	32 ビット符号付整数
`$float`	`float`	32 ビット浮動小数点数
`$string`	`java.lang.String`	16 ビット Unicode 文字列
ArrayType	*Type* の配列	可変長配列
PublicStructName	（下記）	（下記）
StubOrStructName	（下記）	（下記）

スケルトン定義ファイル中の *PublicStructName* は，公的な構造体（3.4 節）の名前でなければならない．スペース定義ファイル中の *StubOrStructName* は，公的な構造体（3.4 節）の名前か，同じスペース定義ファイルで定義される，スタブクラス（3.6 節）か私的な構造体（3.4 節）の名前でなければならない．

PublicStructName や，*StubOrStructName* が構造体の名前の場合，これらが対応する Java の型は，構造体に対して出力されるクラス（3.4 節）の型である．このとき，属性の値はその構造体の値である．

StubOrStructName がスタブクラスの名前の場合，*StubOrStructName* に対応する Java の型は，スタブクラスの型である．属性の値は，スタブクラスのカテゴリへ属するオブジェクトの ID（3.2 節）であり，Java のプログラム上ではその ID を持つオブジェクトに対応するスタブ（3.6 節）で表現される．ただし，そのオブジェクトが存在（3.2 節）しない場合，`null` である．

3.4　構造体

スケルトン定義ファイル中の *StructDecl* およびスペース定義ファイル中の *PrivateStructDecl* は**構造体**を定義する．構造体は，C 言語の構造体に似た，ユーザ定義の型である．構造体は *StructName* あるいは *PrivateStructName* で定義される名前を持つ．構造体に対して，構造体と同じ名前のクラスが出力される．スケルトン定義ファイル中の *StructDecl* に対しては，このクラスはトップレベル・クラス（top level class, Java 言語仕様 §8）であり，スペース定義ファイル中の *PrivateStructDecl* に対しては，このクラスはサブシミュレータ基底クラス（3.6 節）の内部クラス（inner class, Java 言語仕様 §8.1.2）である．

SuperStruct が指定されていた場合，出力されるクラスは，*SuperStruct* で示される名前を持つ構造体に対して出力されるクラスのサブクラスになる．*SuperStruct* を名前に持つ構造体の判定は，*StubOrStructName* と同様に行われる．

構造体の値は，*FieldDecl* で定義される変数を持つ．*FieldDecl* の *FieldName* は変数の名前を表し，*Type* は変数の型を表す．また，*SuperStruct* が指定されている場合，*SuperStruct* で示される構造体が持つ変数も持つ．出力されるクラスは，変数の値を返す，変数と同名で引数のないメソッドを持つ．また，変数の値を設定する，文字列 `set` と変数の名前の先頭の文字を大文字に変換して得られる文字列とをつなげてできる文字列を名前に持ち，1 つの引数をとるメソッドを持つ．引数の型は変数の型であり，戻り値の型は `void` である．構造体を型に持つ属性の値を取得した場合，その値をこのメソッドによって変更しても，SSTD 上の値は変更されない．

SkeletonDefFile	::=	(*StructDecl* | *SkeletonDecl*)*
StructDecl	::=	`$struct` *PublicStructName* (`$extends` *SuperStruct*)?
		{ *FieldDecl** }
PublicStructName	::=	*JavaFQN*
JavaFQN	::=	*Identifier* (`.` *Identifier*)*
SuperStruct	::=	*PublicStructName*
FieldDecl	::=	*Type FieldName* `;`
FieldName	::=	*Identifier*
SkeletonDecl	::=	`$skeleton` *PublicCategoryName*
		{ *PropDecl** }
PublicCategoryName	::=	*Identifier* (`.` *Identifier*)*
PropDecl	::=	*Type PublicPropName* `;`
PublicPropName	::=	*Identifier* (`.` *Identifier*)*
Identifier	::=	⟨アルファベットで始まり，アルファベット，数字，およびアンダースコア (_) からなる文字列⟩
Type	::=	*PrimitiveType* | *ArrayType* | *PublicStructName*
PrimitiveType	::=	`$boolean` \| `$byte` | `$int` \| `$float` | `$string`
ArrayType	::=	*Type* `[ ]`

図 3.1: スケルトン定義ファイルの文法
アンダーラインは終端記号をあらわす.

Java ではクラスのインスタンスはすべて参照値で表されるため，構造体に対して出力されるクラスのインスタンスに対して，任意の連結グラフ構造を構成するように変数の値を設定できてしまう．しかし，サブシミュレータは，木構造以外のグラフ構造を構成させてはならない.

3.5　スケルトン定義ファイル

スケルトン定義ファイルは，公的なカテゴリ（3.2 節）と公的な属性（3.2 節）を定義する．スケルトン定義ファイルの文法は，図 3.1 の拡張 BNF で示される.

StructDecl は，構造体を定義する（3.4 節）.

SkeletonDecl は，**スケルトン**を定義する．スケルトンは，*PublicCategoryName* を名前に持つカテゴリに属するオブジェクトが，*PropDecl* で示される属性を持つことをあらわす．*PropDecl* では，*PublicPropName* によってその属性の名前が与えられ，*Type* によってその属性の型（3.3 節）が与えられる．１つのカテゴリに，同じ名前を持つ公的な属性が複数存在してはならない.

3.6　スペース定義ファイル

spacegen は，スペース定義ファイルに対して，サブシミュレータの基底クラスとなるクラス（**サブシミュレータ基底クラス**）を出力する．異種サブシミュレータは，異なるサブシミュレータ基底クラスのサブクラスとして実装されなければならない.

SpaceDefFile	::=	*SpaceDecl*
SpaceDecl	::=	`$space` *SpaceName* `;` (*PrivateStructDecl* | *StubDecl* | *PrivateStubDecl*)*
SpaceName	::=	*JavaFQN*
JavaFQN	::=	*Identifier* (`.` *Identifier*)*
PrivateStructDecl	::=	`$struct` *PrivateStructName* (`$extends` *SuperStruct*)? `{` *FieldDecl** `}`
PrivateStructName	::=	*JavaInnerClassName*
SuperStruct	::=	*PrivateOrPublicStructName*
PrivateOrPublicStructName	::=	*Identifier* (`.` *Identifier*)*
FieldDecl	::=	*Type FieldName* `;`
FieldName	::=	*Identifier*
StubDecl	::=	`$stub` *StubName* `=` *PublicCategoryName* (`$extends` *SuperStub*)? `{` (*PublicPropDecl* | *PrivatePropDecl*)* `}`
PrivateStubDecl	::=	`$stub` *StubName* `=` `$private` *PrivateCategoryName* (`$extends` *SuperStub*)? `{` *PrivatePropDecl** `}`
StubName	::=	*JavaInnerClassName*
PublicCategoryName	::=	*Identifier* (`.` *Identifier*)*
PrivateCategoryName	::=	*Identifier*
SuperStub	::=	*StubName*
PublicPropDecl	::=	*IODirection PrivatePropName* `=` *PublicPropName* `;`
PrivatePropDecl	::=	`$private` *IODirection Type PrivatePropName* `;`
IODirection	::=	`$in` \| `$out` | `$io`
PrivatePropName	::=	*Identifier*
PublicPropName	::=	*Identifier* (`.` *Identifier*)*
Identifier	::=	⟨ アルファベットで始まり，アルファベット， 数字，およびアンダースコア (_) からなる文字列 ⟩
JavaInnerClassName	::=	*Identifier*
Type	::=	*PrimitiveType* | *ArrayType* | *StubOrStructName*
PrimitiveType	::=	`$boolean` \| `$byte` | `$int` \| `$float` | `$string`
ArrayType	::=	*Type* `[ ]`
StubOrStructName	::=	*Identifier* (`.` *Identifier*)*

図 3.2: スペース定義ファイルおよび構造体定義ファイルの文法
アンダーラインは終端記号をあらわす.

スペース定義ファイルの文法は，図 3.2 の拡張 BNF で示される．*SpaceName* は，出力されるサブシミュレータ基底クラスの完全限定名（fully qualified name, Java 言語仕様 §6.7）である．*SpaceName* はサブシミュレータごとに異なる名前で無ければならない．名前の衝突を回避するために，*SpaceName* は Java の命名規約に従って付けられなければならない（Java 言語仕様 §6.8）．

PrivateStructDecl は，構造体を定義する（3.4 節）．

StubDecl および *PrivateStubDecl* は，**スタブクラス**を定義する．スタブクラスは SSTD 上のオブジェクトへのアクセスを容易にする Java のクラスである．スタブクラスのインスタンスを**スタブ**と呼ぶ．スタブクラスは *StubName* を名前に持つ，サブシミュレータ基底クラスの内部クラス（inner class, Java 言語仕様 §8.1.2）である．SSTD 上のオブジェクト 1 つに対して，スタブが 1 つ割り当てられる．同じオブジェクトに対しては，スタブがガーベジコレクションによって失われない限り，同じスタブが割り当てられ続ける．この割り当てられたスタブを，そのオブジェクトに**対応する**スタブと呼ぶ．

PublicCategoryName は，スケルトン定義ファイルで定義される公的なカテゴリ（3.2 節）の名前でなければならない．*PrivateCategoryName* は，私的なカテゴリ（3.2 節）の名前であり，このカテゴリはこの *PrivateStubDecl* によって定義される．*PublicCategoryName* や *PrivateStubDecl* で与えられるカテゴリを，そのスタブクラスのカテゴリと呼ぶ．1 つのスペース定義ファイル中に，同じカテゴリのスタブクラスが複数存在してはならない．SSTD 上のオブジェクトへのアクセスは，そのオブジェクトが属するカテゴリのスタブクラスのインスタンスを通じて行われる．

PublicPropDecl は，サブシミュレータが公的な属性（3.2 節）へアクセスすることを宣言する．*PublicPropName* は公的な属性の名前で無ければならない．*PrivatePropName* は，スタブクラスで提供される，この属性へアクセスするメソッドの名前を決定するのに使われる文字列である．

PrivatePropDecl は，サブシミュレータが私的な属性（3.2 節）へアクセスすることを宣言する．*PrivatePropName* は，属性の名前であると同時に，この属性へアクセスするメソッドの名前を決定するのに使われる文字列である．この属性の型（3.3 節）は，*Type* で与えられる．

IODirection は，アクセスの種類を表し，`$in` の場合読み込みのみ，`$out` の場合書き込みのみ[3]，`$io` の場合読み書きを行う．

読み込みアクセスを行う場合，スタブクラスには，*PrivatePropName* で与えられる名前を持ち，引数を取らないメソッドが含まれる．このメソッドは，サブシミュレータの同期時刻（3.7 節）における属性の値を返す．

書き込みアクセスを行う場合，スタブクラスには，文字列 `set`，*PrivatePropName* の先頭の文字を大文字に変換した文字列，文字列 `At` の 3 つ をつなげてできる文字列を名前に持つメソッドが含まれる．このメソッドは，属性の型に対応する Java の型の値と，シミュレーション時刻を引数に取る．このメソッドは，引数のシミュレーション時刻において属性が引数の値を持つこと が妥当である，とサブシミュレータが判断したことをシミュレーションシステムに対して宣言する．**実際のシミュレーション結果が，このメソッドで宣言された値になることは保証されない．**シミュレーション時刻は，そのサブシミュレータの現在時刻よりも大きく，かつ，同じオブジェクト同じメソッドの直前の呼び出しにおける引数より大きくなければならない．

[3] この書き込みはシミュレーションシステム全体としてのシミュレーション結果として採用されるとは限らず，SSTD には書き込みと異なる値が保持される可能性がある．サブシミュレータはシミュレーションシステム全体としてのシミュレーション結果に基づいて次のシミュレーションを行わなければならないので，属性に`$out` ではなく`$io` でアクセスすることが推奨される．

3.7　シミュレーションの進行

サブシミュレータは，それぞれ，**現在時刻**と呼ばれる値を持つ．現在時刻は，getClock() メソッドによって得ることができる．現在時刻は，そのサブシミュレータがシミュレーションを終了しているシミュレーション時間の上限をあらわす．現在時刻は，セッション（3.9 節）の間は減少しない．

サブシミュレータは，それぞれ，**同期時刻**と呼ばれる値を持つ．サブシミュレータの同期時刻は，getSyncTime() メソッドによって得ることができる．同期時刻は，現在サブシミュレータがシミュレーションの入力としているシミュレーション時刻を表す．現在時刻も，セッションの間は減少しない．同期時刻は 現在時刻と同じかぞれより以前の時刻である．

同期時刻や 現在時刻は，サブシミュレータが以下で説明されるメソッド（sync **メソッド**）を呼び出すことで変更される．sync メソッドは，サブシミュレータの現在時刻を増加させる．また，sync メソッドは，下のように同期時刻を増加させる．いずれも，同期時刻における SSTD の値が決定されるまで，メソッドは終了しない．

- syncAt(*time*，*lookahead*)
  現在時刻を *time* に *lookahead* の時間を加えた時刻に設定し，同期時刻を *time* に変更する．*time* は呼び出し時における同期時刻より大きくなければならない．*lookahead* は 0 か正でなければならない．*time* に *lookahead* の時間を加えた時刻が，呼び出し前の現在時刻以前であれば，現在時刻は変わらない．

- syncByEvent(*propertyList*，*lookahead*)
  このメソッドの呼び出しは，syncByEventUpto(*propertyList*，Time.MAX_VALUE，*lookahead*) の呼び出しと等価である．

- syncByEventUpto(*propertyList*，*time*，*lookahead*)
  propertyList は属性の名前のリストを表す．指定された属性の値が，syncByEventUpto の呼び出し直前の同期時刻より未来，*time* 以前の時刻において変化する場合，同期時刻をそれらの時刻で最小のものに変更する．そうでない場合，同期時刻を *time* へ変更する．いずれの場合も，現在時刻を，syncByEventUpto 終了時の同期時刻に *lookahead* を加えた時刻に設定する．*time* は呼び出し時における同期時刻より大きくなければならない．*lookahead* は 0 か正でなければならない．呼び出し終了時の同期時刻に *lookahead* の時間を加えた時刻が，呼び出し前の現在時刻以前であれば，現在時刻は変わらない．

これらのメソッドは，サブシミュレータがセッション（3.9 節）を終了させなければならない場合，SessionFinishedException 例外を発生させる．この例外が発生した場合，サブシミュレータはセッションを終了させなければならない．

3.8　オブジェクトの増減

シミュレーション空間へのオブジェクトの導入は createObjectsAt メソッドによって行われる．オブジェクトの取り除きは removeObjectsAt メソッドによって行われる．これらのメソッドは，

サブシミュレータがオブジェクトの導入や取り除きが発生することが妥当だと判断したことをシステムに宣言するだけであり，実際にオブジェクトが導入されることや，取り除かれることは保証されない．これらのメソッドの引数は，導入あるいは削除されるオブジェクトに対応するスタブのリストと，導入あるいは削除が実行されるシミュレーション時刻である．サブシミュレータは，スタブを new し，そのインスタンスを createObjectsAt に渡すことで新しいオブジェクトを SSTD へ導入する．

サブシミュレータは，導入されたり取り除かれたオブジェクトを，getCreatedObjects メソッドや getRemovedObjects メソッドによって取得できる．これらのメソッドは引数を取らず，直前の sync メソッドを呼び出す前の同期時刻より未来で，sync メソッド終了後の同期時刻以前の時間において増加あるいは減少したオブジェクトのリストを返す．

3.9　セッションとリストア

１つのシミュレーションを**セッション**と呼ぶ．セッションの間は，同期時刻は減少することは無く，決定された属性の値が変更されることは無い．あるシミュレーションの実行を途中で止めて，過去のシミュレーション時刻に時間を戻してシミュレーションを再開した場合，時間を戻す前のシミュレーションと，時間を戻した後の新しいシミュレーションは，別のセッションである．例えば，シミュレーション時刻 0 から 100 までシミュレーションを行い，シミュレーション時刻を 50 まで戻してシミュレーションを再開した場合，最初の時刻 0 から 100 までのシミュレーションが１つのセッションであり，時刻 50 以降の２度目のシミュレーションが２つめのセッションである．また，あるシミュレーションの実行を途中で停止し，システムを再起動してシミュレーションを再開した場合，システムを再起動する前と後のシミュレーションは，別のセッションである．サブシミュレータにとって，セッションは，run メソッドの呼び出しで始まり，これらのメソッドの終了で終わる．

サブシミュレータは，以下で説明されるようにセーブとリストアの機能を実装しなければならない．

シミュレーションシステムは，シミュレーションをセーブしなければならない場合，サブシミュレータの save メソッドを呼び出す．サブシミュレータがリストアのために情報を記録する必要がある場合，このメソッドをオーバーライドしなければらないない（SSTD から得られる情報のみでリストアが可能な場合，しなくてもよい）．save メソッドは，サブシミュレータが sync メソッドを呼び出している最中に呼び出される．save メソッドが呼び出される際に呼び出し中である sync メソッド呼び出しを，その save **メソッド 呼び出しに対応する** sync **メソッド 呼び出し**と呼ぶ．save メソッド中では，save メソッド実行中に呼び出せると明記されていない S-API を呼び出すことはできない．save メソッドは，java.io.DataOutputStream オブジェクトを引数にとる．サブシミュレータは，この出力ストリームに対して，リストアに必要な情報を出力する．このストリームを close してはならない．

リストアとは，過去に実行されたセッションの，save メソッドの呼び出しによってセーブされたある時点におけるシミュレーションの状況を復元し，その状況を初期状態とした新しいセッションを開始することである．シミュレーションのリストアは，次のようにして実現される．

サブシミュレータがすでに何らかのシミュレーションを行っている場合，sync メソッドが

SessionFinishedException 例外を返すことによって，sync メソッドを終了させる．サブシミュレータは SessionFinishedException 例外が発生した場合セッションをなるべく早く終了しなければならない．システムは，サブシミュレータがこれらのメソッドが終了したことを確認し，続いてサブシミュレータの run メソッドを呼び出す．これによって新しいセッションが開始される．

サブシミュレータがなんらかのシミュレーションを実行していない場合（run メソッドが一度も呼び出されていないか，呼び出されたが終了している場合），サブシミュレータの run メソッドを呼び出し，新しいセッションが開始される．

リストアによって run メソッドが呼び出されるとき，リストアで復元すべきシミュレーション状況を保存した save メソッドの呼び出しが存在し，その save メソッドの呼び出しに対応する sync メソッドの呼び出しが存在する．リストアによって run メソッドが呼び出されるとき，同期時刻の初期値は，その sync メソッドの呼び出し前における同期時刻である．getObjects をはじめとした同期時刻における SSTD 上の要素を返すメソッドは，その sync メソッドの呼び出し前における SSTD 上の要素を復元して返す．一方，現在時刻は，その sync メソッドの呼び出し前の現在時刻と同じかそれより未来の時刻であり，かつ，その sync メソッドが終了したときに取りうる最大の現在時刻と同じかより以前の時刻である．**再現性のために，サブシミュレータは，run メソッドの最初の sync メソッドの呼び出しを，シミュレーション状況を保存した save メソッド呼び出しに対応する sync メソッドと同じ引数で呼び出さなければならない．**

3.10　エラー処理

サブシミュレータは，何らかのエラーが発生した場合，次のように処理を行わなければならない．

その例外をメソッドの呼び出し元へ throw することが可能ならば，サブシミュレータはその例外を throw することでメソッドの呼び出しを終了してよい．メソッドの呼び出しを実行する場合，サブシミュレータは reportError メソッドによってその例外ををシステムに報告しなければならない．

そうでない場合，サブシミュレータは reportError メソッドによってその例外ををシステムに報告しなければならない．その後サブシミュレータは，処理を続行しても良いし，メソッドを終了しても良い．

付 録A　サンプルプログラム

(to be filled)

A.1　Version 0 骨骨モデル データ構造

(to be filled)