Open SDV API
全体概要とAPIコンセプト

バージョン：202509α
発行日：2025年9月30日

Open SDV Initiative

ドキュメントの位置付け

ドキュメントの目的

- このドキュメントは，Open SDV Initiativeで作成を進めている Open SDV API の全体概要とAPI設計のコンセプトを説明するものである

ドキュメントの完成度

- このドキュメントの現バージョンは，完成度が低い部分が残っている。部分的に，大幅な改訂を行う可能性がある

変更履歴

バージョン	発行日	備考
202503α	2025年3月31日	初版
202509α	2025年9月30日	

目次

バージョン 202503aとの主な違い

データ型の規定

- データ型の詳細定義を行った
  - 数値型に値の範囲(特殊値を含む)の指定を追加した
  - (列挙型の)集合型を導入した
- データ型のプログラミング言語ライクな記述法を規定した
- データのYAMLでの記述法を規定した

車両の構成記述

- 車両の構成記述をYAMLで記述する方法を規定した

APIの要素

- 機能セット, リスクセットの概念を定義した
- サービスコールの戻り値を具体化した
- イベントのデータ型やイベントキューの操作方法(サービスコール)について規定した

リスク制御

- 「アクセス制御」を「リスク制御」と改名し，実現方法を大幅に見直した
  - OEMにより使用が承認されたAPIについても，使用を許すかどうかの最終判断をユーザに求めることに
  - 安全性確保のためのアクセス制御とプライバシ保護のためのアクセス制御を統合して，リスク制御に改名
  - 「安全性クラス」を「リスククラス」と改名し，車両の構成記述には，その集合を記述できるように変更
  - リスク制御情報に，安全性クラス毎にサービスコールを呼び出せる車両状態の条件を記述するのではなく，対応できているリスククラスの集合を記述するように変更

車両状態の定義

- 車両状態の定義の目的を，アクセス制御に使用することから，リスククラスが危険を生じる可能性がある車両状態の記述を容易にすることに変更

共通規定

- 共通データ型を定義した
- 各機能の規定を具体化した
  - パラメータとリターンパラメータの名称と型を規定した
- getConfig と getConfigAll を分離した
- getEvent，notify，unnotify の記述を追加した
- MovableObject の詳細を規定した

OPEN SDV INITIATIVE の活動と現状

オープンSDVとOpen SDV Initiative

オープンSDVとは？

- サードパーティ(自動車メーカやその委託先以外の組織や個人)が開発したアプリケーションをインストールすることができる車両を，オープンSDVと呼ぶ

Open SDV Initiative の活動

- Open SDV Initiative は，近い将来にオープンSDVが実現・普及することを想定して，オープンSDVのためのビークルAPIとして “Open SDV API” を策定する
  - 策定したAPIは，標準化団体に提案する
- また，Open SDV API を評価するためのシミュレータの開発や実車を用いたPoCも行う
- ! 以下では，単にAPIと言えば，ビークルAPIを指すものとする

Open SDV Initiative の活動経緯

これまでの活動経緯

- 2024年6月：立ち上げを発表，参加企業の募集を開始
- 2024年10月：本格的な活動を開始
  - APIコンセプトWG，ボディWG，AD/ADAS WG，HMI WG，開発環境/シミュレーションWG，実車製作WG，調査WGを設置し，検討を開始
  - 12月に，UXアイデア検討WGを追加設置
- 2025年3月：Open SDV API仕様の最初の版（バージョン：202503α）を公開
- 2025年9月：Open SDV API仕様の改訂版（バージョン：202509α）を公開
  - APIコンセプトWG，ボディWG，AD/ADAS WG，HMI WGのこの時点までの検討結果の取りまとめ

Open SDV API仕様

Open SDV API

- Open SDV Initiative で策定するビークルAPIを，“Open SDV API” と呼ぶ

現時点（バージョン：202509α）の完成度

- 4つのWGの現時点の検討成果を取りまとめたもの
  - 本格的な活動開始から1年で，コンセプトは固まってきたが，個々のAPIの完成度はまだ低い。WGの検討成果間の不整合も残っている
  - 知見のあるメンバがいなかったために，設置できなかったWGがあり，抜けも多い
  - 仕様の内容検討を優先したため，体裁も整っていない
- APIの策定が急がれる状況であり，他団体との協調のためにも，完成度が低くても現時点の仕様を公表すべきと判断

各WGの取り組みと成果の概要

APIコンセプトWG

- API策定にあたっての考え方を整理
- 想定するシステム階層を定義
- 車両の構成記述の方法を定義
- APIの要素を規定，サービスコールの設計方針を設定
- 制御の調停とリスク制御の機能を定義

ボディWG

- キャビン，ボディまわりの各種のデバイスに対するAPIの案を提示
  - COVESA VSSをベースにしつつ，CAAM-SDVのAPI設計も参考にしながら，アプリケーションに使いやすいボディAPIを目指した

AD/ADAS WG

- AD（自動運転）/ADASを構成する認知・判断・操作の内，判断部をアプリケーションで，認知部と操作部をビークルOSで実現するための車両周辺モデル，自車位置・方位，ドライバ状態・意図，車両運動制御の4つのAPIの考え方と叩き台について，202503αで提示したものに対して，修正・拡張・具体化を行った
- 策定したAPIの動作を確認できるように，AD/ADASのOSSであるAutowareや周辺のシミュレータの実装を進め，ベースシステムの動作を確認した
- 今後は，Open SDV Initiative内外からの意見を聞き，APIのさらなる修正・拡張を行うとともに，物理APIまでの具体化を行い，実装した環境での評価・検証を行う

HMI WG

- サウンド，グラフィックス，汎用スイッチのAPIの案を提示
- 小さいリソースで実現できるグラフィックスAPIを策定し，広い範囲の車両で動作できるアプリケーションを実現可能に
  - リッチなグラフィックスを使いたい場合は，ベースOS（例えば，Android Automotive）の機能を使えば良い。どちらのAPIを使うかは，アプリケーション開発者が決める

残っている課題 (API関連)

API定義の詳細化・ブラッシュアップ

- APIの定義の詳細化，定義範囲の拡大
- UXアイデア検討WGの成果の反映

検討できていないAPIの領域

- 以下の領域のAPIは，検討に着手できておらず，今後の検討課題として残っている（他にも抜けがある可能性）
  - バッテリー制御，エネルギー管理
  - ナビゲーションや自動運転に対するAPI（アプリケーション間連携）

特定の言語および（コンピュータの）OS向けの物理API

- 特定のプログラミング環境上で Open SDV API を使用するための物理API（バインディング）の規程

今後の計画（API関連以外）

テスト実装と評価

- シミュレータの開発とそれを用いた評価
- 実車を用いたPoC

国（経産省）への打ち込み

- モビリティDX検討会で，ビークルOS開発を協調領域として実施すべきという意見が出ている

標準化団体への打ち込み

- 公開した仕様をベースに，JASPARやCOVESAなどの標準化団体と会話を継続する

API策定にあたっての考え方

策定するAPIの位置付けと考え方

APIの使用者

- サードパーティ（OEMやOEMから委託を受けたサプライヤ以外の組織や個人）とOEM（または委託先のサプライヤ）の両方が使うことを想定してAPIを策定する
- APIの使用により危険な動作を起こす可能性はあるが（これに関する考え方は後述），以下の理由から，APIをリスク（危険性）があるものとないものに分類することはしない
  - あるAPIにリスクがあるかどうかは，車両が持つ安全機能によって異なる
  - 現時点ではリスクがあるAPIが，技術の発展（車両の安全機能の進化）により，リスクをなくせる可能性がある
  - リスクがあるAPIをサードパーティに使わせて良いかどうかが，国や地域の規制によって異なる場合もある

どのようなアプリケーションを想定するか？

- 現時点で考えられるアプリケーションの例を後で示すが，想像しないようなキラーアプリケーションが出てくる可能性もあるため，汎用性・拡張性の高いAPIとすることを目指す
  - （逆に言うと）見えているアプリケーションだけに特化したAPIとはしない

どのような車両を想定するか？

- まずは，一般の乗用車（シェアリング用の車両やタクシーも含む）を想定する
- （可能な範囲で）サービスカー（バスやトラックなど）にも適用できるように配慮する

どのAPI(どの階層のAPI)を策定するか？

- Open SDV Initiative は，アプリケーションとビークルOSの間のインタフェースであるビークルAPIを策定対象とする
  - ビークルAPIには，以下の2つの大きい目標がある
    - ✓車両によらず，同じアプリケーションが動作できる
    - ✓アプリケーションの開発が容易になる
  - この2つの目標が一致しない場合があることから，ビークルAPIを，前者の目標を達成するためのコアビークルAPIと，後者の目標を達成するための拡張ビークルAPIの2階層で構成する(詳しくは後述)
  - 実際には，想定するアプリケーションに対して，アプリケーションとビークルOSの役割分担を検討し，APIを定義する階層を決めることが必要

どのAPI(どの階層のAPI)を策定するか？ – 続き

- 必要であれば，ビークルデバイスインタフェース(定義は後述)も策定するが，策定の優先度は低くする
- (コンピュータの)OSのAPIは，既存のもので十分な場合はそれを使えば良いため，策定対象としない
  - 具体的には，AUTOSAR APやAndroid Automotiveが持っている機能のAPIは，それらで十分であれば，策定対象としない
  - ただし，既存のAPIではビークルAPIとして不十分な場合には，策定対象とする

APIの安全性に関する考え方

- ビークルAPIの中には，（対策をしないと）危険な動作を起こす可能性があるAPIが多数ある
- APIにより危険な動作を起こす可能性がある場合に，ビークルOSは，それを防止する機能を持っていることが望ましい（必須ではない）
  - 策定するAPIは，可能な範囲で，ビークルOS（特にコアビークルサービス）により危険を防止できるようなものとする
  - 同じAPI呼び出しに対しても，危険を防止できる車両と，できない車両がある
- ビークルOSで危険を防止できないAPIも残るため，そのようなAPIをアプリケーションが使用することを禁止できるようにするために，リスク制御と呼ぶ機能を設ける

APIのリスク制御

- どのアプリケーションにどのAPIの使用を許すかは，OEMとユーザの判断
- OEM（またはOEMから委託された者）は，アプリケーションを審査し，何らかのリスクがあるAPIの使用を禁止することができる
  - 審査する観点の例：危険な制御を行わないこと，ドライバデストラクションがないこと，など
  - 審査の結果として，APIの使用可否を制御する情報を作成する。これを「リスク制御情報」と呼ぶ
  - リスク制御情報は，審査機関による署名付きで，アプリケーションと一緒に配布する
- OEMの審査で使用が禁止されなかったAPIについては，ユーザにリスクを説明し，アプリケーションに使用を許可するかどうかの最終判断を求める

策定するAPIは，どのECUで使用するものか？（アプリケーションはどのECUで動作するか？）

- 基本方針：車両のE/Eアーキテクチャ（ECUとネットワークの構成）によらず使用できるAPIとする
- E/Eアーキテクチャは車両によって異なるため，APIを使用するECUは限定しない
  - 下位層に分散プラットフォーム（AUTOSAR APやROSなど）があることを想定すれば，別のECUで実現されたサービスを呼び出すことは可能
  - 実際には，IVI ECUやAD/ADAS ECUで使用することを想定する
  - アプリケーションによって異なるECUで動作させることや，1つのアプリケーションを複数のECUに分散して動作させることも想定する

策定するのは，論理APIか物理APIか？

- まずは論理API（セマンティクスレベルのAPI）を策定する
  - 物理API（シンタックスレベルのAPI）は，プログラミング言語や（コンピュータの）OSに依存する。変わったとしても，アプリケーションの作り直しは機械的
  - APIを介して受け渡しされる数値を表すデータの型については，単位まで規定する
    - ✓単位が異なると，アプリケーションの修正が大きくなるため
- 代表的なプログラミング環境（プログラミング言語やOS）に対する物理API（バインディング）も規定する
  - シミュレータやPoCの開発にあたって必要であるため
  - 異なるプログラミング環境向けの物理API間でのデータ変換等のオーバヘッドが大きくなることを防ぐため

クラウド上のAPI（クラウド上やスマホ上のアプリケーションから，クラウドを介して車両の機能を操作するためのAPI）は策定しないのか？

- クラウド上のAPIも重要な課題であると認識している
- 車両の機能を操作するAPIは，車両内で使用するAPIもクラウド上のAPIも，以下を除いては大きい違いはないと考えられるため，まずは車両上のAPIにフォーカスする
  - クラウドには，車両に関する過去のデータやそれを集計したものを蓄積するため，クラウド上では，それらの蓄積データにアクセスするAPIが必要になる可能性がある
- クラウド上のアプリケーションは，車両上のアプリケーションを介して，車両を操作する（＝車両上のAPIを使う）想定

API策定にあたって重視していること

アプリケーションの開発容易性

- サードパーティのアプリケーション開発者にとっては，一度の開発で，多くの車両に適用できるアプリケーションが開発できることが重要
- APIで車両の違いを完全に隠蔽できなくても，アプリケーション側で車両の違いが吸収できれば良い
  - 例えば，車両によってウィンドウの数が異なることに対しては，ウィンドウのリストをAPIを使って取得し，各ウィンドウに対して操作するAPIを呼び出す方法をとる
  - このようなケースで，車両の違いに依存せずに使えるAPI（例えば，すべてのウィンドウを操作するAPI）が欲しい場合には，ライブラリ等で用意する
- 車両（主にハードウェア）の機能不足により，アプリケーションが動作できない場合があるのはやむを得ない

多様な車両に適用できるAPIとする

- 日系OEMの特徴／強みを活かせるAPIとするために，多様な車両に適用できるAPIとすることを目指す
  - 日系OEMの特徴／強み（の1つ）は，多様な車両を開発・販売していること
  - 多様な車両：パワートレインの違い，高級車／大衆車，乗用車以外の車両
- 車両の構成（configuration）の記述方法も策定する

アプリケーションの例

- 以下に示すのは，現時点で思いついているアプリケーションの例であり，想像しないようなキラーアプリケーションが出てくることを期待したい

ボディ系

- セントリーモード，洗車モード

自動運転系

- 自動運転（例：テスラのFull Self-Driving）
- 自動駐車，バレーパーキング

HMI系

- メーターのパーソナライズ

情報提供系

- ナビゲーション
- POI情報提供，旅行情報提供

エンターテイメント系

- ドライブ記録動画/アルバム作成
- 車両を活用したゲーム
- 自動運転中の車内エンターテイメント

プローブ系

- ドライブレコーダ
- テレマティクス保険，安全運転診断，運転能力診断
- リアルタイムストリートビュー
- 地図作成支援

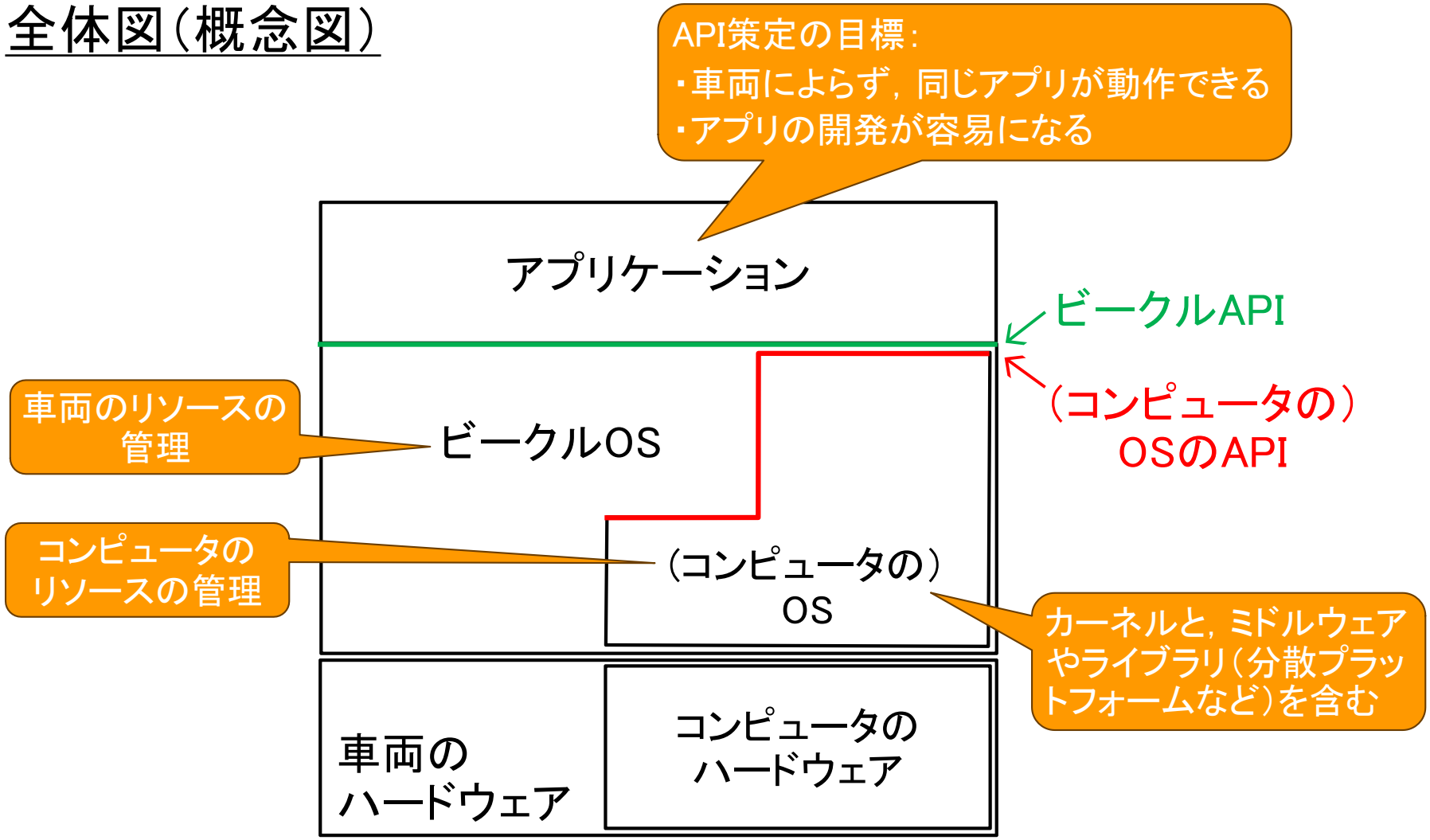
その他

- シェアリングカーの管理
- V2H（Vehicle to Home），V2G（Vehicle to Grid）

想定するシステム階層

想定するシステム階層（概念図）

全体図（概念図）

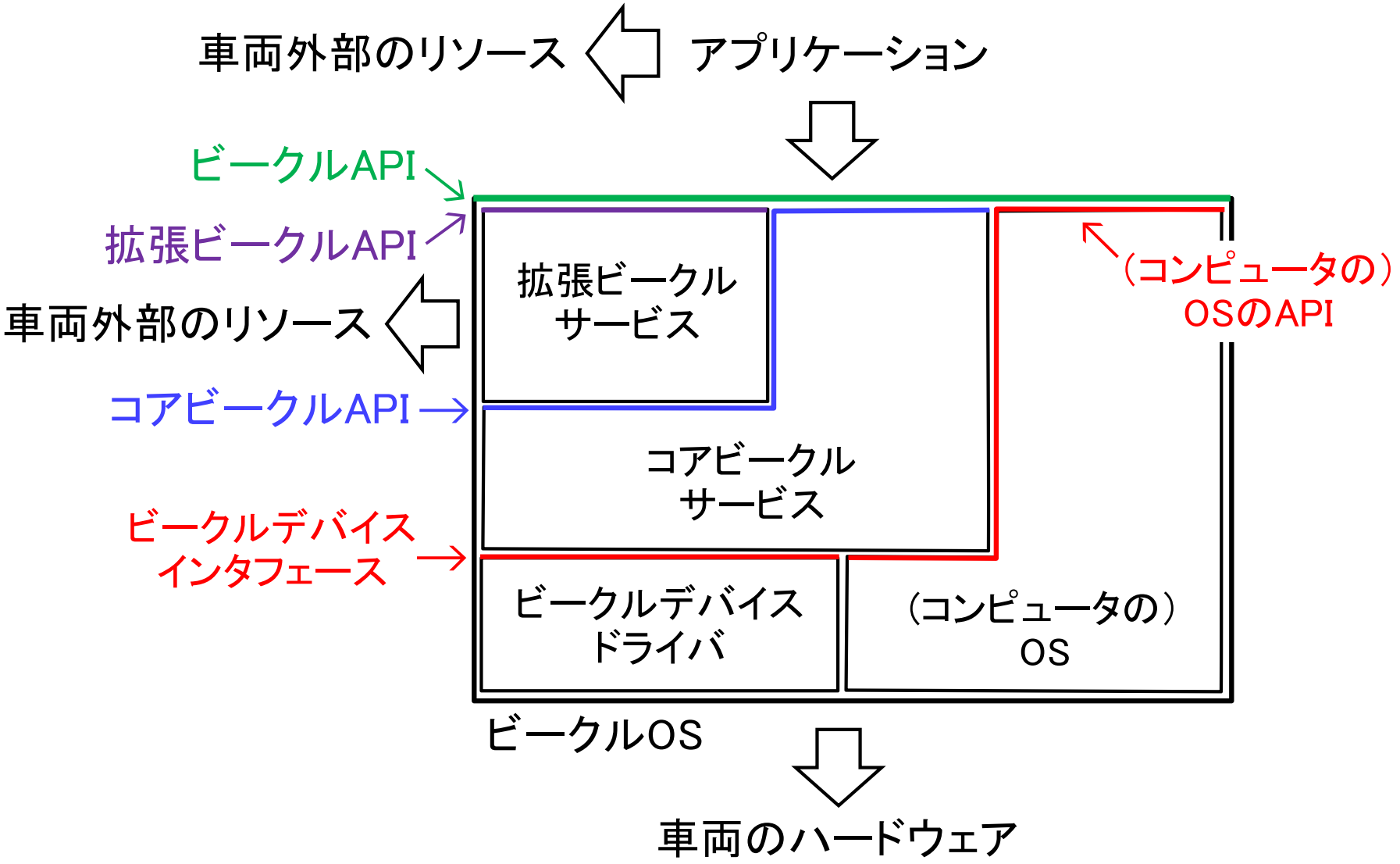


※ 実装構造を表現する図ではない

用語に関する補足説明

- ここでは，OSという用語を，ソフトウェアプラットフォーム（SPF）と同義で用いている（広義のOS）
- 「ビークルOS」は，「（コンピュータの）OS」も含む
  - コンピュータ（のハードウェア）も車両（のハードウェア）の一部分であるため
- 「（コンピュータの）OS」には，カーネルに加えて，ミドルウェアやライブラリを含んでいる
  - 分散プラットフォーム（AUTOSAR APのARA::COMやROSなど）を含んでいることを想定
  - AUTOSAR APのPlatform Foundation FCsに含まれるサービスは，ここに含まれるものと位置付ける

ビークルOS内部の構造（概念図）



コアビークルサービス

- 以下の3つの役割を持つソフトウェア
  - 車両のハードウェアの違いの吸収
  - 車両のハードウェアへのアクセス制御・調停
  - (可能な範囲での)安全性の確保
- 上の3つの役割を果たせる範囲で，コアビークルサービスの機能は最小限にする
  - 車両のハードウェア以外のリソースを管理する機能は，原則として持たせない

コアビークルAPI

- アプリケーションや他のビークルサービスが，コアビークルサービスを使用するためのインタフェース
- 車両によらず同じアプリケーションを動作させるために鍵となるAPIであり，標準化・継続性が重要

拡張ビークルサービス

- アプリケーションの開発を容易にするために，コアビークルサービス上に実装されるソフトウェア
  - 車両のハードウェア以外のリソースを管理する機能は，拡張ビークルサービスで実現する
  - 例えば，地図を管理する機能は，拡張ビークルサービスで実現する。ダイナミックマップは，拡張ビークルサービスの位置付けとなる
- アプリケーションが拡張ビークルサービスの位置付けとなる場合もある
  - 具体的には，アプリケーションが他のアプリケーションから要求を受け付ける場合
  - 例えば，自動運転ソフトウェアやナビゲーションソフトウェアが典型例

拡張ビークルAPI

- アプリケーションや他のビークルサービスが，拡張ビークルサービスを使用するためのインタフェース
- 類似の機能を持つ拡張ビークルサービスが，複数あっても良い
  - 車両外部のリソースを管理する機能に対して，1つのAPIに標準化することは難しいため
- アプリケーション開発者の視点で使いやすいものとすることが重要

ビークルサービス

- コアビークルサービスと拡張ビークルサービスを総称して，ビークルサービスと呼ぶ

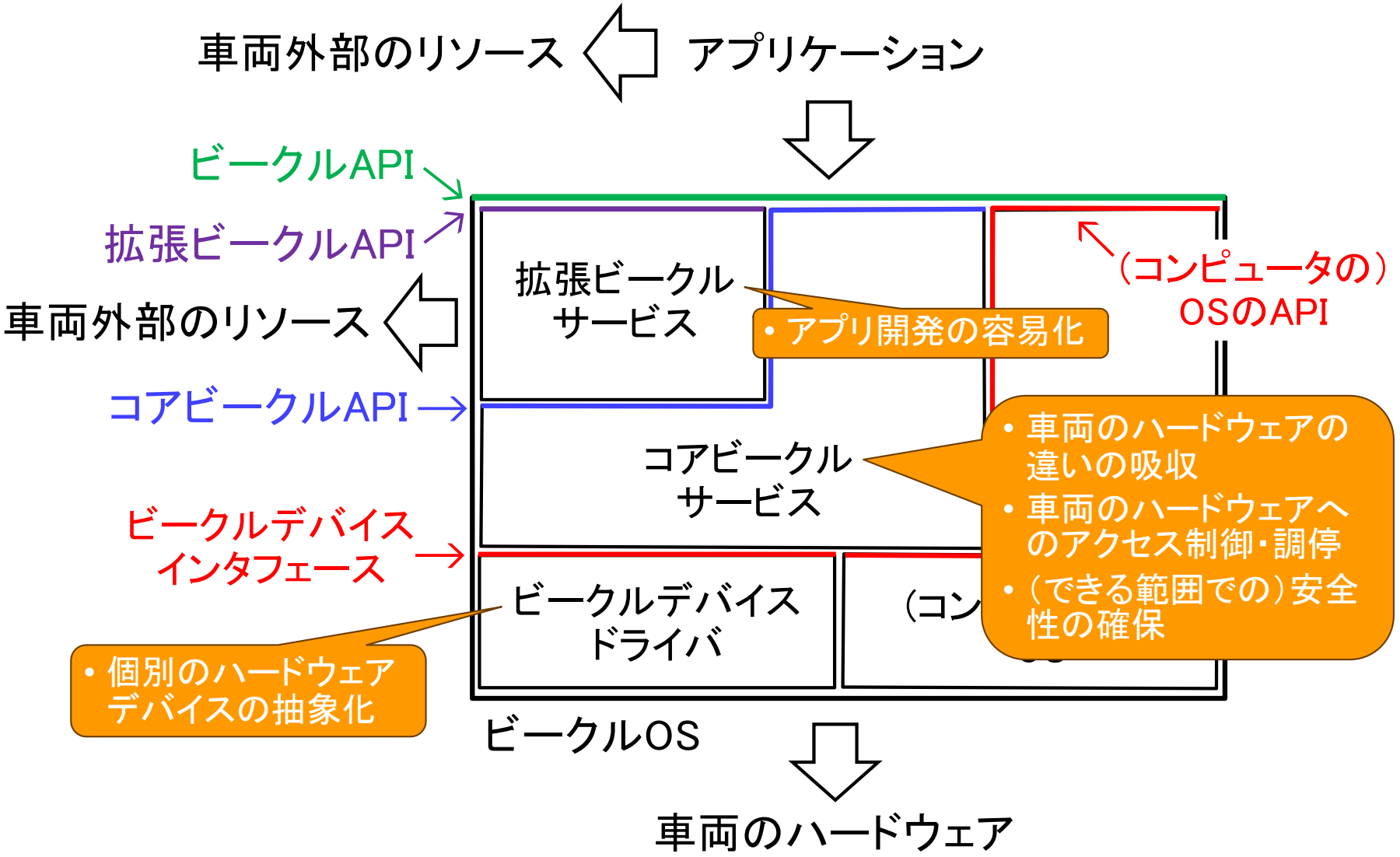
ビークルAPI

- コアビークルAPI，拡張ビークルAPI，（コンピュータの）OSのAPIで構成される

ビークルデバイスドライバ

- 車両のハードウェアデバイスを，それぞれ単独で抽象化するための階層
- ただし，何を1つのハードウェアデバイスと扱うか，どの程度抽象化するかは，一通りには定まらない
  - 例えば，HEVにおいて，「パワートレイン」を1つで扱う方法と，「エンジン」「モータ」「トランスミッション」を別々のハードウェアデバイスと扱う方法がある
  - 例えば，カメラに対する抽象化層は，カメラの生画像を提供する方法と，カメラ画像を認識した結果を提供する方法がある。なお，複数センサーの認識結果のフュージョン処理は，ビークルサービスの役割

ビークルOS内部の構造（役割注釈付き）



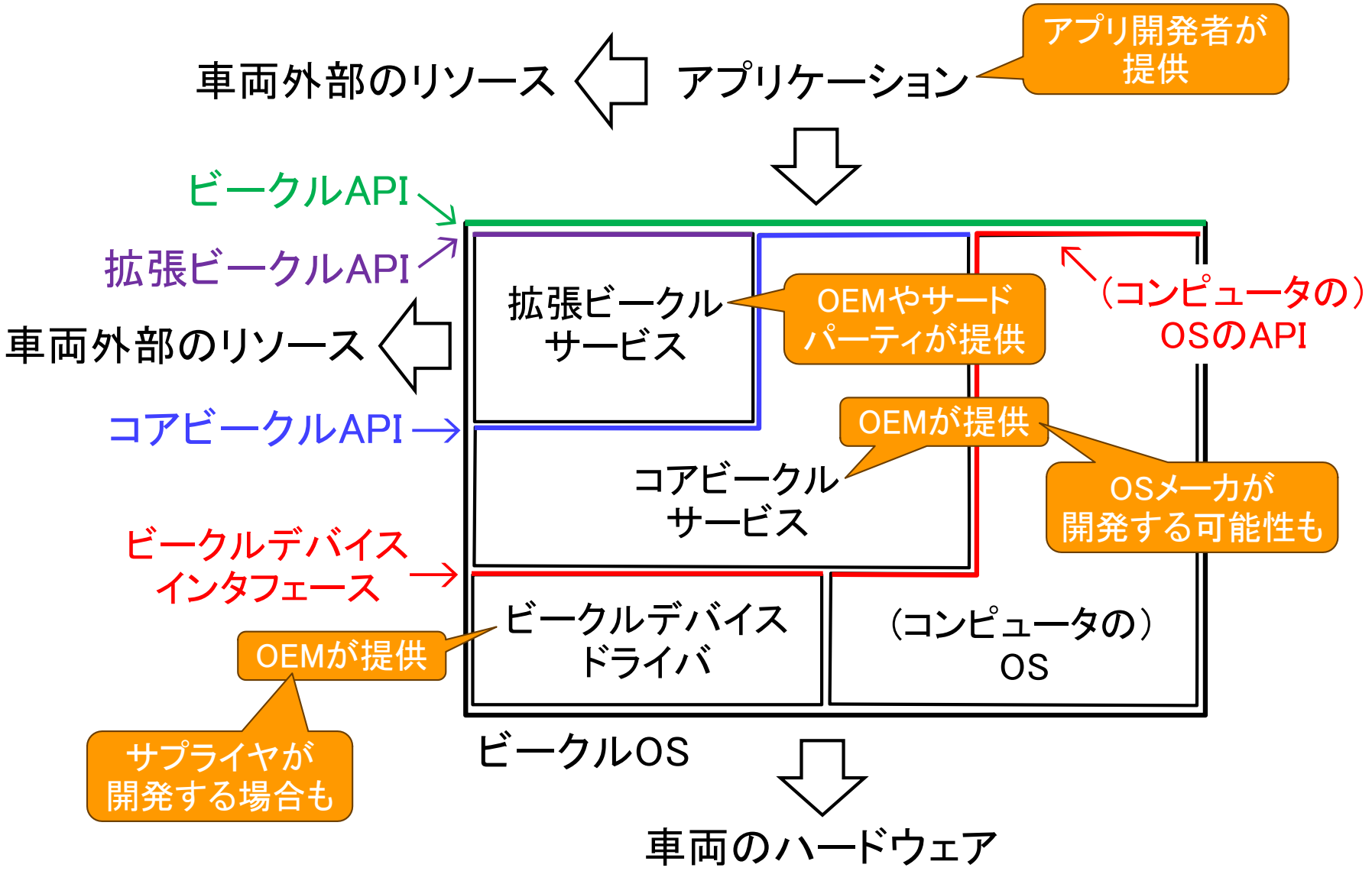
ソフトウェアのモジュール化の意義と階層分割の方針

- ソフトウェアのモジュール化（階層分割を含む）の目標の1つは，ハードウェアや周辺環境（交通規則，外部との通信仕様）の変更があった場合に，1つのモジュールのみの変更で対応できるようにすることである
  - 車両のハードウェアに変更があった場合には，ビークルOSのみの変更で対応できるのが望ましい
  - 交通規則（国・地域によって異なる），外部との通信仕様，地図の仕様の違いに対しては，1つの拡張ビークルサービスの差し替えのみで対応できるのが望ましい
- この目標に近づくように，階層分割を行う方針とする

各モジュールの提供者

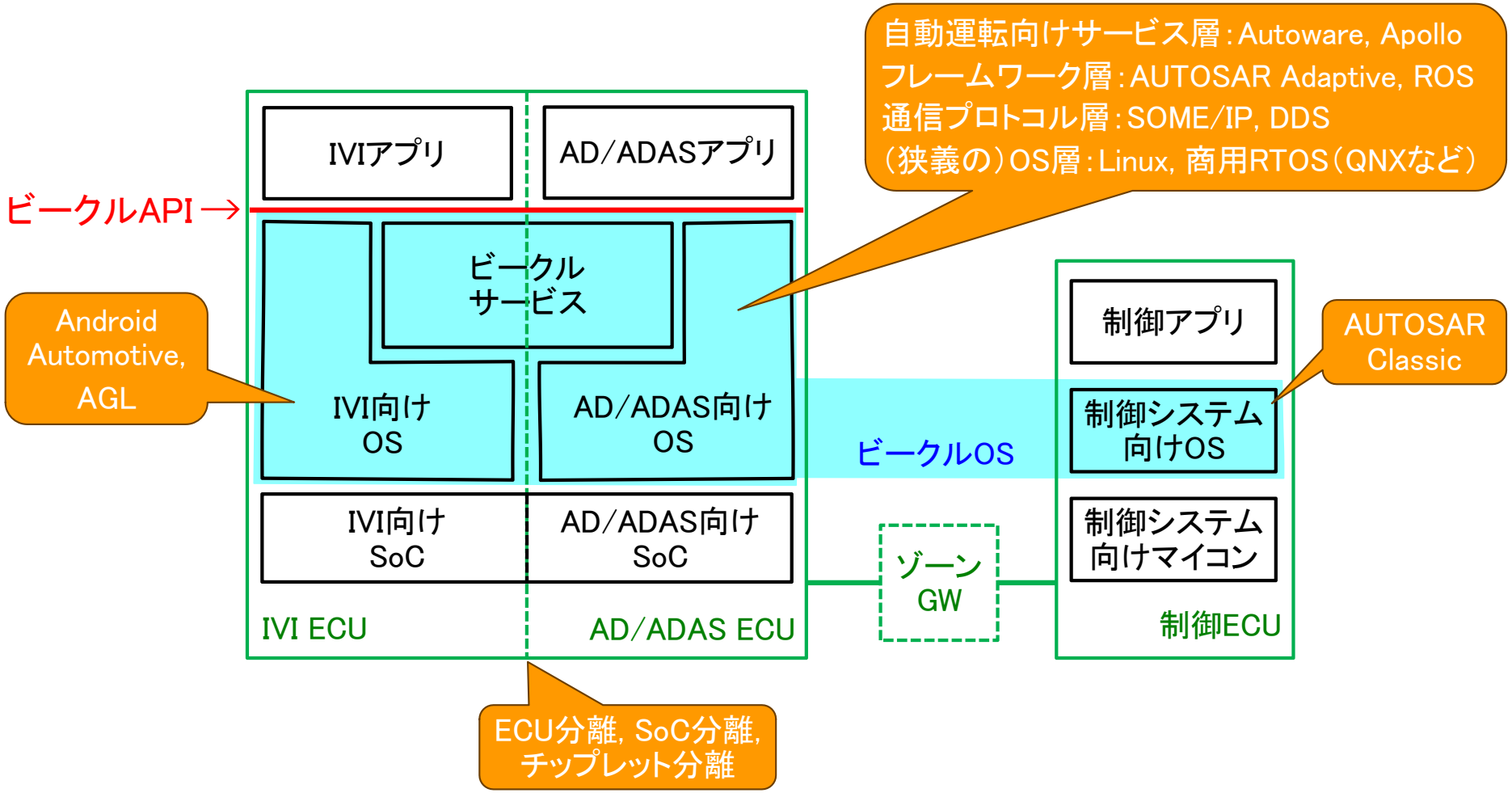
- コアビークルサービス
  - OEMが提供
  - 開発は，OSメーカが行う（OSメーカが開発してOEMに提供する）可能性や，OSSを用いる（OEMがOSSを利用する）可能性もある
- 拡張ビークルサービス
  - OEMが提供する場合もあれば，サードパーティが提供する場合もある
  - OEM，サプライヤ，OSメーカ，アプリケーション開発者の誰が開発する可能性もある
- ビークルデバイスドライバ
  - OEMが提供
  - 開発は，サプライヤが担当することも多いと思われる

ビークルOS内部の構造（提供者注釈付き）



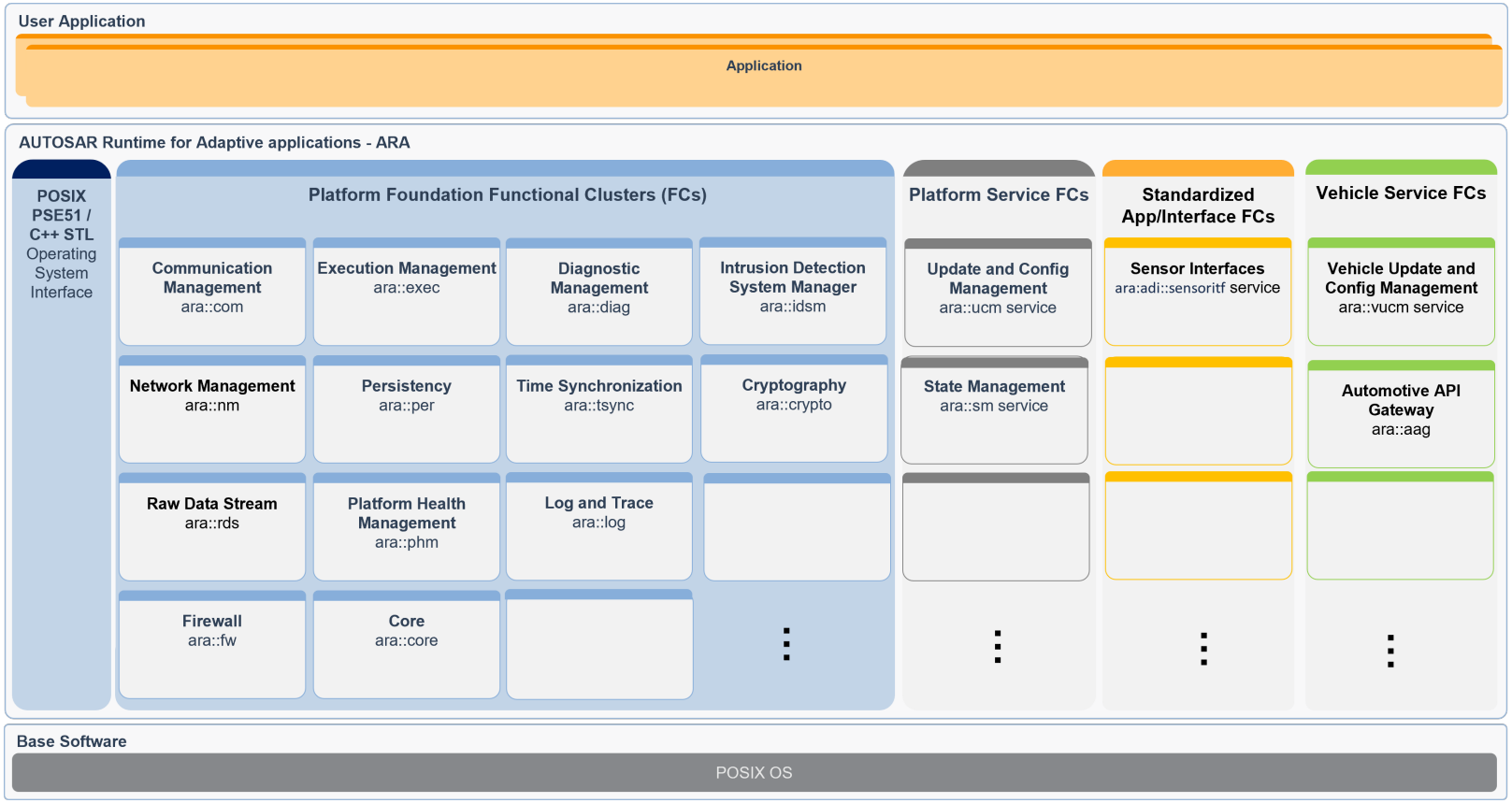
実装構造の例

車両のE/Eアーキテクチャと各種のOSの位置付け



AUTOSAR AP上に実装する場合

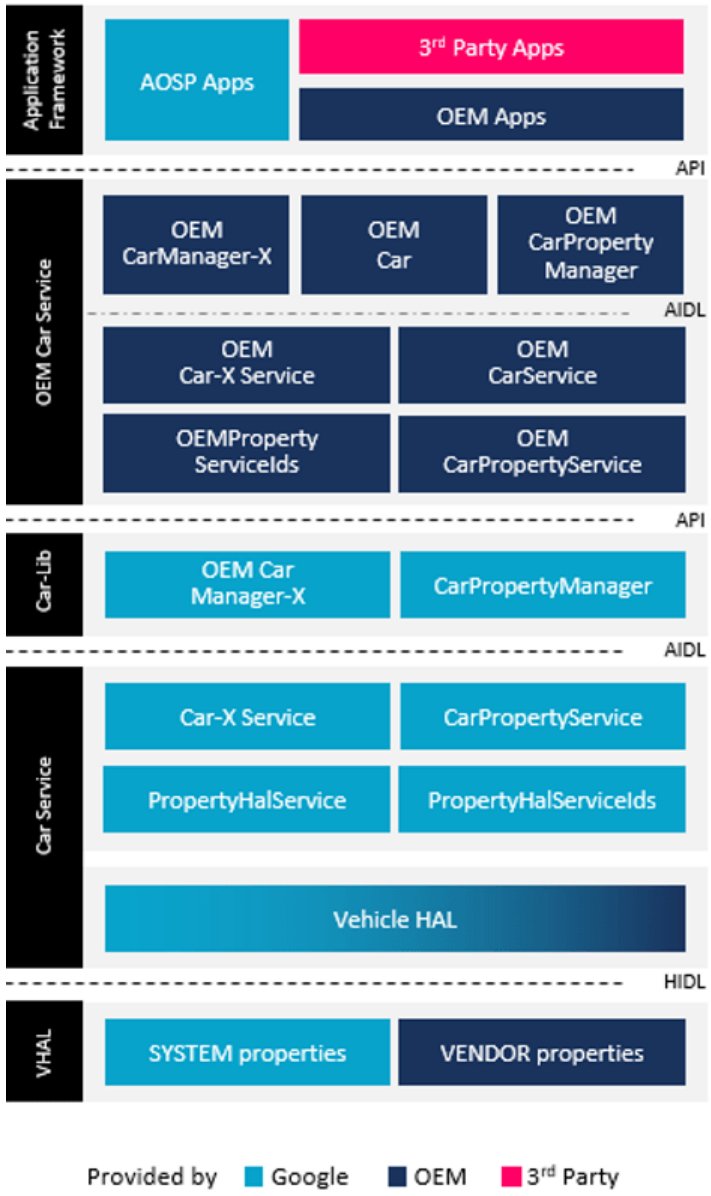
- 現在のAUTOSAR APの構成図（下図）では，User ApplicationとARAの間にService層を設け，ビークルサービスを，Service層で実現することになる



※ 図は，“Explanation of Adaptive Platform Design (R24-11)”より

Android Automotive上に実装する場合

- Android Automotive の構成図（右図）では，ビークルサービスを，“OEM Car Service” と “Vehicle HAL” で実現することになる



※ 図は，“Android Automotive OS Whitepaper”より

APIに関する用語とデータ型

用語の定義

オブジェクト

- ビークルOSが操作対象とするリソース（車両のデバイスや外部リソース）を，ビークルオブジェクト，または単にオブジェクトと呼ぶ
  - 何をオブジェクトとするかは，個別に検討する
- 各オブジェクトに対して，同一種類のオブジェクトが1つのみ（シングルトン）か，複数ある（マルチインスタンス）かを定める
- マルチインスタンスの場合には，必要に応じて，オブジェクトクラス（または単にクラス），オブジェクトインスタンス（または単にインスタンス）と呼んで区別する
- COVESAでは，“item” という用語が使われており，その方が良いかもしれない

構成（configuration）と状態（status）

- オブジェクトの属性の中で，静的に決まるものを「構成（configuration）」または「構成情報」，動的に変化するものを「状態（status）」と呼ぶ
- 両者の中間的な属性もあるため，「構成」が変わる場合もあるものとする

  例）トレーラーが牽引しているもの

  - アプリケーションが依存している「構成」が変わった場合には，再起動が必要となっても良い

状態とイベント

- オブジェクトは，（一般に）状態を持つ
- オブジェクトの状態の（離散的な）変化のことを，イベントと呼ぶ

  - イベントには，状態変化の原因や状態変化前後の状態などの情報が付属している場合がある

IDとハンドル

- 静的なリソース(ビークルオブジェクトなど)は，種類毎に，IDと呼ばれる番号で識別する
  - IDは，1から連続する正の整数値とする
  - IDを表すデータ型を，IdTypeとする
- 動的なリソースは，ハンドルと呼ばれる値で識別する
  - リソースを生成するサービスコールが，ハンドルの値を決定し，アプリケーションに知らせる
  - ハンドルのデータ型は，リソース毎に定める

標準制御

- OEMが自動車の出荷時に組み込んでいる制御ロジックを，標準制御と呼ぶ
  - 標準制御は，アプリケーション，ビークルOS，ハードウェアのいずれで実現されている場合もある
  - 標準制御の動作を止めることができるのを原則とするが，止められないものがあっても良い（ハードウェアで実現されている場合など）

データ型

データ型の用途

- Open SDV APIの論理APIでは，以下の情報に対して，データ型を定める
  - 車両の構成情報
  - サービスコールのパラメータとリターンパラメータ
  - イベント

データ型の規定レベル

- 様々なプログラミング環境（プログラミング言語やOS）に適用できるように，データ型は論理レベルで規定する
  - 具体的には，取りうる値と値の意味を定める
  - 数値を表すデータ型については，単位まで規定する
- 実装におけるデータ型との対応については，物理API（バインディング）で定める

用意するデータ型

- 基本データ型(ブール型，符号付き/なし整数型，浮動小数点数型)，文字列型
- 固定長配列，可変長配列
- 列挙型，(列挙型の)集合型
- 構造体，タグ付き共用体
- オプショナル型

用意しないデータ型

- 文字型，ポインタ型，参照型

データ型の記述法

- データ型を記述するために，形式的な記述法を導入する
  - 既存のもので目的に合致するものがあれば使いたいが，現状では見つかっていないため，独自に定義した

論理APIにおけるデータ型の定義方法

原則と例外

- 論理APIにおけるデータ型の定義では，取りうる値と値の意味（数値を表すデータ型の場合は単位まで）を規定する
  - 取りうる値は，後述のデータ型の記述法で記述する
- ただし，プログラミング環境に依存するデータ型については，物理APIに規定を委ねる場合もある

  例）ApplicationIdType（アプリケーションのインスタンスの識別子の型）

型名のコンベンション

- データ型の名称は，先頭が大文字のキャメルケース（PascalCase）とし，末尾に “Type” を付けるものとする
  - ただし，基本データ型と文字列型は例外とする

データ型の詳細定義

基本データ型

- ブール型
  - 真（true）と偽（false）の2つの値のみを取るデータ型
- 符号付き整数型
  - 8/16/32/64ビット以上の長さの符号付き整数型
  - 取りうる値の範囲（特殊値を含む）を指定できる
- 符号なし整数型
  - 8/16/32/64ビット以上の長さの符号なし整数型
  - 取りうる値の範囲（特殊値を含む）を指定できる
- 浮動小数点数型
  - IEEE 754準拠の32/64ビットの浮動小数点数型
  - 取りうる値の範囲（特殊値を含む）を指定できる

取りうる値の範囲（特殊値を含む）の指定

- 数値を表すデータ型には，取りうる値の範囲を，以下の指定の任意個の和集合の形で指定できる
  - 最小値と最大値で指定 [a, b]
  - 排他的下限と最大値で指定 (a, b]
  - 最小値と排他的上限で指定 [a, b)
  - 排他的下限と排他的上限で指定 (a, b)
  - 他の値と重複しない特殊値を定数名で指定

例）0以上100以下の値と，UNKNOWN，NONZERO のいずれかの値をとる

例）0.0以上で無限大（+INF）より小さい値と，NaN のいずれかの値をとる

取りうる値の範囲（特殊値を含む）の指定 ― 続き

- 特殊値の具体的な値は，別途規定する
- 特殊値も含めて，範囲指定対象のデータ型で表現できることが必要
  - 例えば，8ビットの符号なし整数型に対して，取りうる値の範囲を，0以上255以下の数値と，1つの特殊値の和集合とすることはできない
- 特殊値の定数名は，1つの数値型の中でユニークであれば良い
  - 特殊値の定数名が数値型を超えてユニークでなければならないプログラミング環境向けの物理APIでは，特殊値の定数名の前に，名前の衝突を避けるための文字列を付加する

文字列型

- Unicodeの文字列を取るデータ型

固定長配列

- 指定された要素型のデータの指定された個数の配列

可変長配列

- 指定された要素型のデータの任意個数の配列
  - C言語では，配列を置いた領域へのポインタと要素（データの個数）数で実現する
  - ベクタ（例えば，C++の std::vector）やリストで実現しても良い
- 要素数が0の場合もある
- 配列の要素数を得ることができる

列挙型

- あらかじめ定義された列挙子のいずれかを取るデータ型
- 列挙子は，1つの列挙型の中でユニークであれば良い
  - 列挙子が列挙型を超えてユニークでなければならないプログラミング環境向けの物理APIでは，列挙子の名前の前に，名前の衝突を避けるための文字列を付加する

（列挙型の）集合型

- 指定された列挙型の列挙子の集合を取るデータ型
  - C言語では，各列挙子が1つのビットに対応する符号なし整数型（ビットマップ）で実現する
- 列挙型以外の型の集合型も考えられるが，可変長配列で扱えば十分と考え，用意しない

構造体

- フィールド名とフィールド値の型が指定されたフィールドを，複数まとめて扱うデータ型
  - フィールド毎に型が異なっていても良い

タグ付き共用体

- データの種別を表すタグと，そのタグに対応するデータ（ペイロード）をまとめて扱うデータ型
  - 各タグに対するペイロードは，型が異なっていても良い
  - プログラミング言語によっては，バリアント型や関連値付き列挙型と呼ばれている
- タグのデータ型として，列挙型を指定する
- ペイロードを持たないタグがあっても良い
  - タグ付き共用体の定義で，タグとしてリストアップされていない列挙子に対しては，ペイロードを持たないものと扱う

オプショナル型

- 指定された型の値と，値が存在しないことを示す値（null）のいずれかを取るデータ型
- 値が存在するか否かを得ることができる
- 値が存在する場合には，その値を得ることができる

用意しないデータ型

- 文字型
  - Open SDV API仕様では使用しないと思われるため
- ポインタ型，参照型
  - 参照は別の方法で扱う

データ型の記述法

- 以下では，データ型が取りうる値を，プログラミング言語のようなスタイルで定義するための記述法を規定する
- データ型をYAMLで記述する必要があれば，YAMLによる記述方法を別途検討する

取りうる値の範囲の記述

- 数値のデータ型に指定できる取りうる値の範囲は，“()” 内に，数値の区間または特殊値を表す定数名を，“|” で区切って列挙することで記述する
- 数値の区間は，“[a, b]”，“(a, b]”，“[a, b)”，“(a, b)” のいずれかで表す

例）Int8 ([0,100] | UNKNOWN | NONZERO)

例）Float32 ([0.0, +INF) | NaN)

基本データ型

- Bool
- Int8 ＜範囲指定＞
- Int32 ＜範囲指定＞
- UInt8 ＜範囲指定＞
- UInt32 ＜範囲指定＞
- Float32 ＜範囲指定＞
- Int16 ＜範囲指定＞
- Int64 ＜範囲指定＞
- UInt16 ＜範囲指定＞
- UInt64 ＜範囲指定＞
- Float64 ＜範囲指定＞

文字列型

- String

固定長配列

- *ElementType*[*size*]

可変長配列

- *ElementType*[]

列挙型

- enum { *enumerator1*, *enumerator2*, … }

（列挙型の）集合型

- enumSet(*enumType*)

構造体

- struct {
  fieldName1 : *FieldType1*,
  fieldName2 : *FieldType2*,
  …
  }

タグ付き共用体

- union(*enumType*) {
  enumerator1 : *PayloadType1*,
  enumerator2 : *PayloadType2*,
  …
  }

オプショナル型

- *Type*?

型への名前付け

- type *TypeName* = ＜型記述＞;

YAMLでのデータ記述法

- 以下では，車両の構成記述ファイル等で，YAMLによりデータを記述する方法を規定する
  - データ型をYAMLで記述する方法ではないことに注意

基本データ型，文字列型

- YAMLの Scalar の文法に従って記述する

固定長配列，可変長配列

- 配列の要素を，YAMLの Sequence で記述する

列挙型

- 列挙子を文字列で記述する

（列挙型の）集合型

- 集合に含まれる列挙子を，YAMLの Sequence で記述する

構造体

- フィールド名とフィールド値を，YAMLの Mapping により記述する
- フィールド名は，文字列で記述する

タグ付き共用体

- タグとペイロードを，YAMLの Mapping により記述する
- ペイロードを持たない場合には，ペイロードを記述しないか，“null” と記述する

オプショナル型

- 値が存在しない場合には，値を記述しないか，“null” と記述する

車両の構成記述

車両の構成記述

車両の構成記述

- 車両の構成記述は，車両の構造・装備・機能・性能など，車両の静的な属性を表現するものである

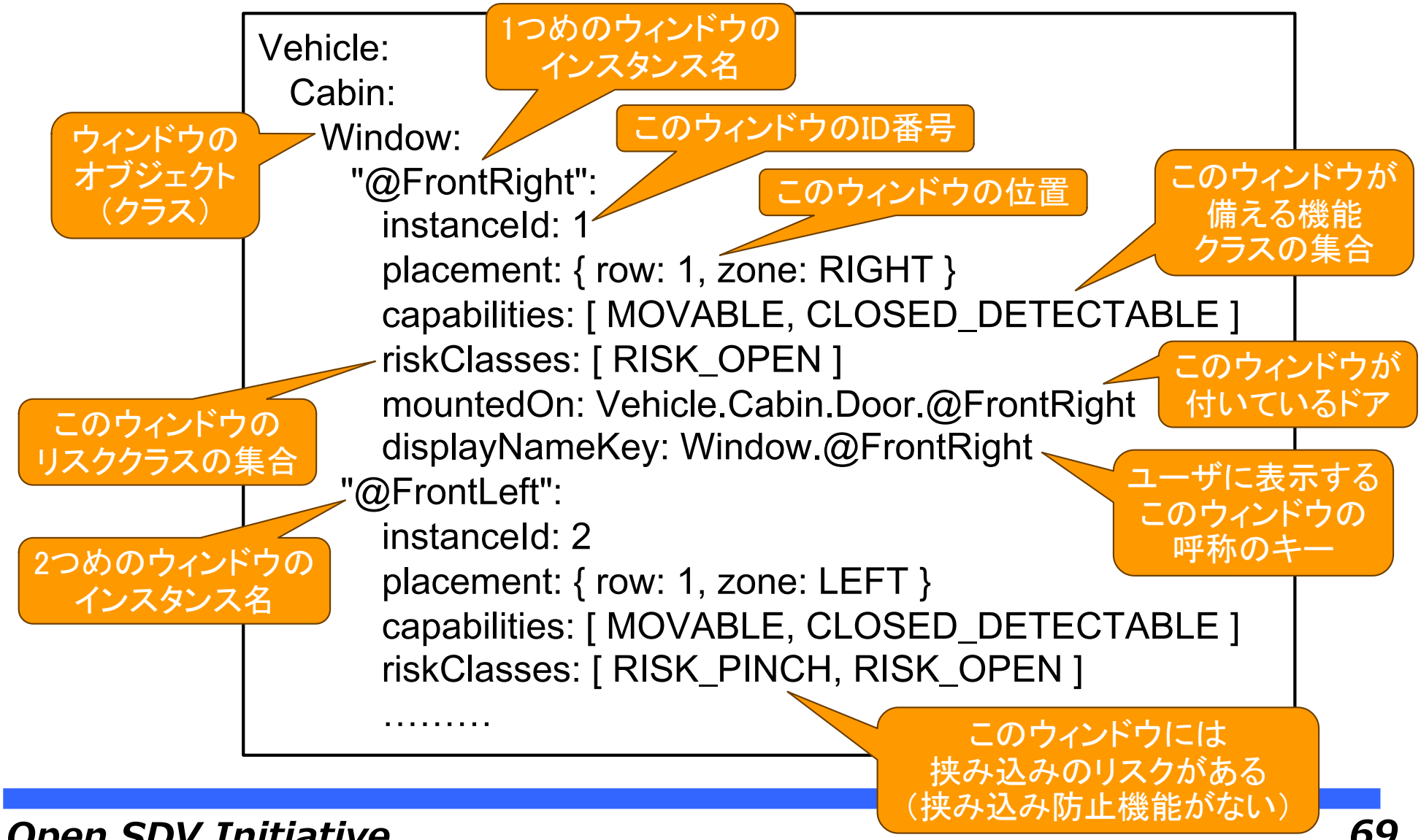
構成記述の基本的な考え方

- 車両の構成を階層構造で表現する
  - 階層構造は，COVESA VSSをベースに，DVE+Uモデル[1]も参考に定める
  - 結果的には，VSSとかなり違ったものとなっている
- APIの対象となるオブジェクトは，階層構造の中間ノードに位置付ける
- 車両の構成記述では，階層構造の各末端ノードに対して値を定義する

[1] M. Gondo: “(The) Four Principles of SDV”, 15th AUTOSAR OPEN CONFERENCE, 2025年6月.

車両構成記述の例

- ある車両のウィンドウの構成情報を，YAMLで記述した例



構成記述に対する要求・留意点

- 車両の構造・機能を，トップダウンに整理する
  - 例えば，洗車モードに遷移させるアプリケーションを作成する場合，水が入る可能性がある開口部をすべて数え挙げられることが必要
  - そのためには，キャビンの開口部に何があるか（例えば，ドア，ウィンドウ，ガラス）を整理し，各開口部の性質（例えば，ウィンドウは閉めれば水を通さない）を規定する必要がある
- 車両依存の拡張を考慮した記述方法とする
  - 車両の多様性を考えると，車両の構成記述法を網羅的に規定するのは難しく，車両依存で拡張できることが望ましい
- 階層構造のレベルは，あまり深くならないようにする

トップレベル

- VSSと同様に，Vehicle とする
  - DVE+Uモデルをベースに考えると，トップレベルを Vehicle，Driver，Environment の3つとする考え方もできるが，Driver は車両の運転者であること，Environment も車両の周辺環境であることを考えて，COVESA VSS と同様に，トップレベルは Vehicle だけとする

第2レベル

- DVE+Uモデルでの Vehicle は，VSS をベースに，階層をあまり深くしない，Vehicle 直下にオブジェクトを置かないなどの方針から，以下のようにする
  - Cabin … 乗員が乗る空間
  - CargoSpace … 荷物を乗せる空間（物流を考えて独立させる，VSSにはない）
  - Body … 主に車両の外側にある装備
  - Infotainment … 文字通り（VSSでは Cabin の下にある）
  - HMI … 文字通り（VSSでは Cabin.Infotainment の下にある）
  - Motion … 車両の運動制御（VSSにはない）
  - CurrentLocation … 車両の現在位置
  - Specification … 車両諸元（VSSにはない）

第2レベル – 続き

- Vehicle 関係の以下の第2レベルについては，十分に検討できておらず，現時点ではVSSのままとする
  - Powertrain … 情報を取るだけで制御はしない
  - Chassis … 情報を取るだけで制御はしない
  - ADAS … 現時点ではVSSのままとする
  - VehicleIdentification … 現時点ではVSSのままとする
- VSSでVehicle直下にある車両の移動に関する属性は Motion の下に，その他の車両諸元は Specification の下に置く

第2レベル – 続き

- DVE+Uモデルでの Driver は，運転者以外の乗員も含んでいるため，VSSと同様，以下の2つを第2レベルとする
  - Driver … 運転者（遠隔運転の場合は，車内にいるとは限らない）
  - Occupant … 運転者以外の乗員
- DVE+Uモデルでの Environment は，ISO 23150のコンセプトを導入して，以下の2つを第2レベルとする
  - SurroundModel … 車両の周辺モデル（ISO 23150で定義される情報，VSSにはない）
  - Atmosphere … 車両の環境情報（VSSでは Exterior）
- TODO：VSSにあるその他の第2レベルについては，さらに検討が必要

インスタンスを表すノードとインスタンスID

- マルチインスタンスのオブジェクトに対しては，クラスを表すノードの下に，インスタンスを表すノードを置く
  - インスタンスを表すノード名は，インスタンス名の前に“@”を付けた文字列とする
    - ✓YAMLでは，“@” は予約文字であるため，ノード名を “”” で囲む必要がある。そのため，予約文字でない “$” にする案も考えられる
- インスタンスの構成情報には，サービスコールなどでインスタンスを数値で識別する場合に用いるインスタンスID（“instanceId”）を含める

オブジェクトに付属するオブジェクト

- オブジェクトに付属するオブジェクトは，前者のオブジェクトのインスタンスの下に置くのではなく，独立したオブジェクトと扱う
  - 付属関係は，属性（構成情報）により表現する
- 例えば，ウィンドウやガラスに付属しているシェード（日除け）は，各種のシェードをすべて Cabin の下の Shade オブジェクトのインスタンスとし，属性（構成情報）を使ってウィンドウやガラスと紐付ける

  例）Vehicle.Cabin.Shade.@RearRight.covers を
  Vehicle.Cabin.Window.@RearRight に設定する
  - どのシェードも同じサービスコールで制御する

OEM固有のオブジェクト／属性

- OEMは，固有のオブジェクトや属性（構成情報や状態）を追加することができる
- TODO：標準のオブジェクト／属性と名称が衝突しないように，OEM固有のオブジェクト／属性の命名規則を定める
- 残課題：OEM固有の構成情報を追加した場合，getConfig/getConfigAll で取得する構成情報のサイズが大きくなるため，（C言語バインディングのように）サービスコールを呼び出す側でメモリ領域を取得する場合に問題となる
  - API仕様の改版により，構成情報が追加された場合にも，同様の問題がある

車両の構成情報の取得方法

- 車両の構成情報を取得する方法には，次の2つがある
- 構成記述ファイルの参照
  - 車両の構成記述を含むファイルを，車両の構成記述ファイルと呼ぶ
  - 構成記述ファイルは，階層構造をYAML形式で記述する(「車両構成記述の例」のスライドを参照)
- 構成情報を取得するためのサービスコール
  - シングルトンのオブジェクトに対しては，getConfigサービスコールにより，その全構成情報を参照できる
  - マルチインスタンスのオブジェクトに対しては，getConfigAllサービスコールにより，そのすべてのインスタンスの全構成情報を参照できる

構成記述ファイルの多言語対応

- 車両の構成情報の中で，ユーザに対するオブジェクトの呼称（displayName）などは，以下の方法で多言語対応する
- 構成記述ファイル中には，多言語対応する文字列のキーを記述する（例：Window.@FrontRight）
- 言語毎の文字列は，ロケール毎に用意する別のファイルに記述する
  - 言語毎の文字列を記述するファイルの例

displayName_ja-JP.yaml

```
Window:
  "@FrontRight": "運転席の窓"
  "@FrontLeft": "助手席の窓"
  "@RearRight": "後席右側の窓"
  "@RearLeft": "後席左側の窓"
```

getConfigAll によるインスタンス名の参照

- マルチインスタンスのオブジェクトに対する getConfigAll は，インスタンスの全構成情報を含む構造体の，インスタンス数の長さの可変長配列を返す
  - getConfigAll でインスタンス名を取得するために，インスタンスの構成情報の構造体に，インスタンス名を格納するためのフィールド（“instanceName”）を追加する
  - ただし，連想配列が扱えるプログラミング環境では，連想配列として扱っても良い

getConfig/getConfigAll でのオブジェクトへの参照の扱い

- getConfig/getConfigAll では，オブジェクトへの参照である構成情報（例えば，ウィンドウの “mountedOn”）は，オブジェクトの種類とインスタンスID（またはインスタンス名）の2つのフィールドを持つ構造体で返しても良い（物理APIで選択）

構成記述ファイルのユースケース

- ビークルOSは，構成記述ファイルを読み込む
  - ビークルOSが単一の車両のみに対応するのであれば，ビークルOS自身は，車両の構成記述を参照することは必須ではない
- アプリケーション開発者は，アプリケーションを対応させたい車両群の構成記述ファイルを参考にすることができる
- シミュレータや各種ツールが，構成記述ファイルを利用することも考えられる
- アプリケーションが，車両の構成情報の取得するために，構成記述ファイルを読んでも良い
  - 一般には，getConfig/getConfigAll サービスコールを使用した方が効率的（各アプリケーションが構成記述ファイルを読むより，ビークルOSが読んだ方が効率的）

APIの要素と設計方針

APIの規定方法と規定内容

共通規定とオブジェクト毎の規定

- APIは，操作対象のオブジェクト毎に規定する
- 複数のオブジェクトに共通の事項については，「共通規定」において規定する

オブジェクト毎のAPIの規定内容

- 名称（車両の構成記述での表記）と略称
- 位置付けと機能
- 機能クラス，リスククラス
- 構成情報，状態
- サービスコール（一覧），イベント
- データ型の定義
- 各サービスコールの詳細規定

名称と略称

名称

- オブジェクトの名称（正式名）は，車両の構成記述における階層構造を示す名称（フルパス名）とする

  例）Vehicle.Cabin.Window

略称

- 各オブジェクトに対して，そのオブジェクトを識別するためのユニークな略称を与える

  例）Window

- 略称は，ObjectKindType の列挙子などに使用する

位置付けと機能

位置付け

- オブジェクトの車両内での位置付けや性質を説明する

  例）Windowは，車室の開閉できる開口部で，車両の走行中も開けておいて差し支えないものである

機能

- オブジェクトの持つ機能や動作について説明する
  - オブジェクトが Movableオブジェクトである場合には，そのことを明記する
- オブジェクトに対する操作の調停方法について説明する
  - 標準的な調停機能（後勝ちの原則）が適用される場合には，そのことを明記する
  - オブジェクトが Lockableオブジェクトである場合には，そのことを明記する

機能クラス

機能クラスとは？

- オブジェクトの備える機能を分類したもの

  例）ウィンドウの機能クラス

  - MOVABLE … APIにより開閉できる
  - CLOSED_DETECTABLE … 全閉状態を検知できる
  - HAS_MOVE2 … 二次元に移動できる
- 車両の構成記述で，オブジェクト（のインスタンス）毎に，備えている機能クラスの集合を記述する

機能クラスに関する規定事項

- Open SDV API仕様では，オブジェクト毎に，以下のことを規定する
  - 機能クラスのリスト
  - 各機能クラスの名称と説明

リスククラス

リスククラスとは？

- オブジェクトを操作することで生じるリスクを分類したもの
- 詳しくは，「リスク制御の概要」の項を参照

リスククラスに関する規定事項

- Open SDV API仕様では，操作することによって何らかのリスクが生じる可能性があるオブジェクト毎に，以下のことを規定する
  - リスククラスのリスト
  - 各リスククラスの名称と説明
  - 危険を生じる可能性がある操作の種類（サービスコールとそのパラメータ）と車両状態 → サービスコールの詳細規定に記述する

構成情報

構成情報とは？

- オブジェクトの機能・性能など，オブジェクトの静的な属性を表現するもの
  - マルチインスタンスのオブジェクトでは，インスタンス毎に構成情報を持つ

構成情報に関する規定事項

- Open SDV API仕様では，オブジェクト毎に，構成情報の名称・データ型・意味などを規定する

サービスコール

サービスコールとは？

- アプリケーションが，ビークルOSのサービスを呼び出すインタフェース

パラメータとリターンパラメータ

- サービスコールは，パラメータとリターンパラメータを持つ
- マルチインスタンスのオブジェクトに対するサービスコールにおいては，原則として，第1パラメータを，インスタンスを識別するインスタンスIDとする
  - オブジェクト指向言語などでは，このパラメータはメソッドのパラメータとはせず，メソッドを受け取るオブジェクトで表す

サービスコールの戻り値

- リターンパラメータには，サービスコールが正常に実行されたかエラーになったか，エラーになった場合にはエラーの種類を示す「戻り値」を含める
- 戻り値のデータ型（ReturnType）は，すべてのサービスコールで共通とする
- そのため，特定のサービスコールに特化したエラーの種類は設けず，以下の例のように一般的なエラーの種類を用いる
  - 自分がロックしていないオブジェクトに対して unlock を呼んだ場合のエラーは，E_OBJECT_STATUS とする
  - 空のイベントキューに対して getEvent を呼んだ場合のエラーは，E_NO_DATA とする
- サービスコールが複数のエラーを検出した場合には，その内のどのエラーの種類を返しても良いものとする
- 物理APIで，エラーの種類を追加する場合がある

サービスコールに関する規定事項

- Open SDV API仕様では，オブジェクト毎に，サービスコールの名称と機能，パラメータとリターンパラメータの名称・データ型・意味などを規定する

サービスコール実装上の前提

- ビークルサービスは以下の情報を取得できるものとする
  - サービスコールを呼び出したアプリケーションのインスタンスのID。制御の調停などに用いる
  - サービスコールを呼び出したアプリケーションのリスク制御情報（後述）。リスク制御に用いる
  - サービスコールを呼び出したアプリケーションのロケール（言語および地域設定）

イベントとイベントキュー

イベントとは？

- アプリケーションが，ビークルOSの状態変化を知るために受け取るデータ

イベントのパラメータ

- イベントは，イベント種別に加えて，イベントの発生原因や発生前後の状態などを伝えるためのパラメータを持つ

イベントのデータ型

- イベントのデータ型は，イベントが発生したインスタンスのID（マルチインスタンスオブジェクトの場合のみ）と，イベント情報をフィールドに持つ構造体とする
- イベント情報のデータ型は，イベント種別をタグ，パラメータ（パラメータが複数ある場合には，構造体にまとめる）をペイロードとするタグ付き共用体とする

イベントに関する規定事項

- Open SDV API仕様では，オブジェクト毎に，イベントのデータ型，イベント種別の名称・意味，イベント種別毎のパラメータの名称・データ型・意味などを規定する

イベントキュー

- ビークルOSからのイベントはイベントキューに格納され，アプリケーションは，イベントキューからイベントを受け取る
- アプリケーションは，複数のイベントキューを持つことができる
  - イベントキューは通知ハンドルで識別する
- イベントキューには，複数のイベントを入れることができる
  - イベントキューは，FIFO順でイベントを管理する
  - イベントの優先度管理をしたい場合には，複数のイベントキューを用いて，アプリケーション側で実現する

イベント通知の要求と停止

- イベント通知を要求するサービスコール（notify）を，オブジェクト毎に用意する
  - 1回のnotifyサービスコール呼び出しに対して，1つのイベントキューが生成される。notifyサービスコールが正常に実行されると，通知ハンドルが返される
  - 受け取るイベントの種類を限定する場合には，notifyサービスコールのパラメータで指定する
  - オブジェクトに対して複数のnotifyサービスコールを設けても良い
- 通知ハンドルを指定してイベント通知を停止するサービスコール（unnotify）を，オブジェクト毎に用意する
  - 対応するイベントキューは削除される

イベントの取り出し

- 通知ハンドルで指定したイベントキューからイベントを取り出すサービスコール（getEvent）を，オブジェクト毎に用意する

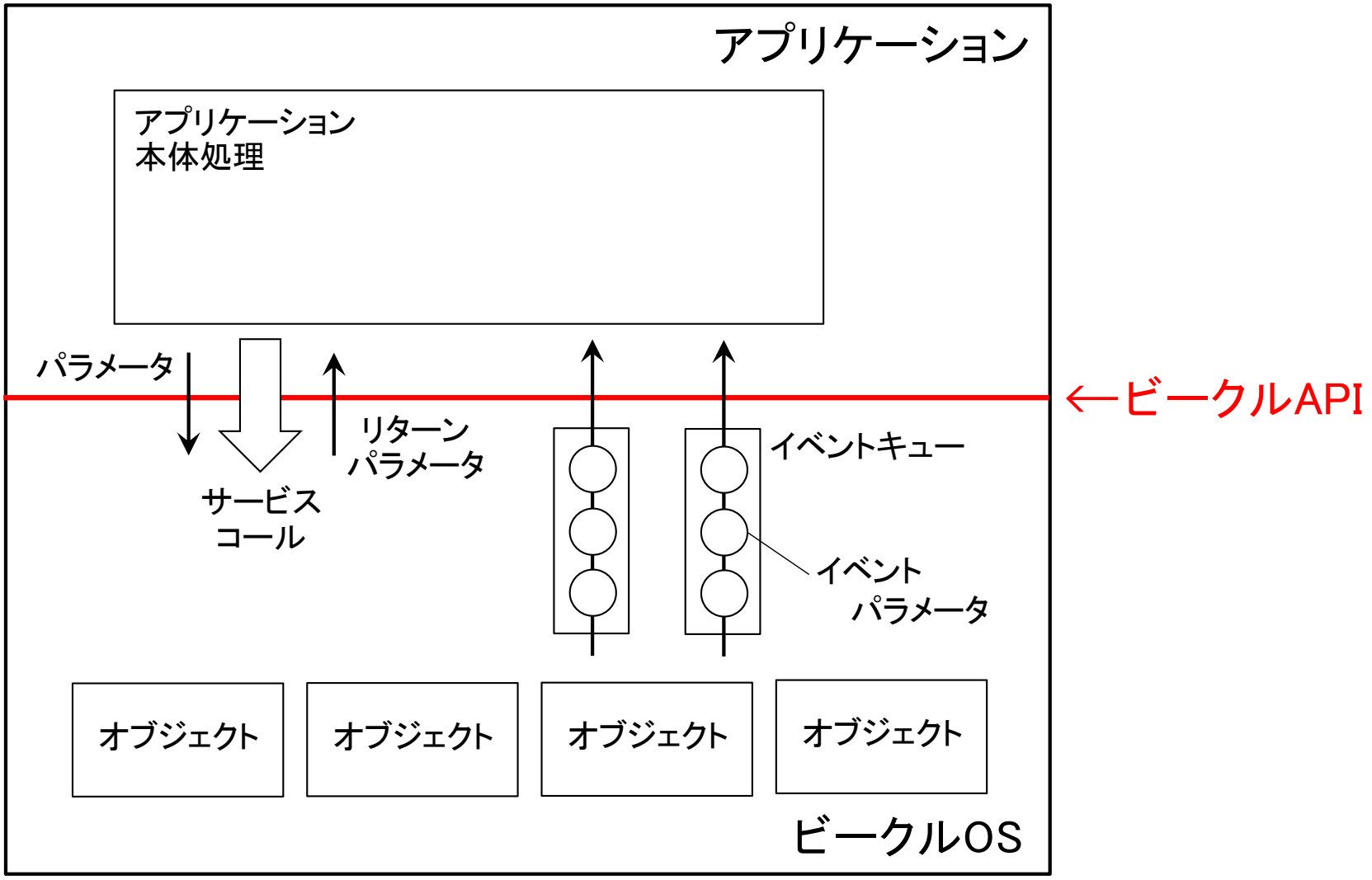
イベントキューの1本化

- プログラミング環境によっては，すべてのイベントを，1つのイベントキューで扱うこととしても良い
- この場合，以下の仕様とする
  - イベントのデータ型（構造体）に，オブジェクトの種別（クラス）と通知ハンドルのフィールドを追加する
  - notify/unnotifyサービスコールは，イベントキューの生成/削除を行わない
  - getEventサービスコールは，システム全体で1種類のみとし，パラメータに通知ハンドルを含めない

システムイベント

- 特定のオブジェクトに関連しないイベントを，システムイベントと呼び，Vehicle オブジェクトに対するイベントとして扱う
- 現時点では，以下のイベントをシステムイベントとすることを考えている（詳細は要検討）
  - 車両の構成の変更
  - オブジェクトに対するイベントキューのオーバフロー

APIの要素（図示）


アプリケーション
アプリケーション
本体処理
パラメータ
リターン
パラメータ
サービス
コール
←ビークルAPI
イベントキュー
イベント
パラメータ
オブジェクト
オブジェクト
オブジェクト
オブジェクト
ビークルOS

サービスコールの設計方針

サービスコールの継続時間

- サービスコールは，短い時間で処理が完了するような設計とする
  - 例えば，ウィンドウを操作するサービスコールは，開閉動作が完了するまで待つのではなく，目標位置を指定して，開閉動作を開始するのみとする

サービスコールの呼び出し頻度/回数

- サービスコールを短い周期（具体的には，10ミリ秒未満）で呼び出す必要があるような設計としない
- サービスコールの粒度を大きくし，呼び出し回数が少なくなるような設計とする
  - サービスコールが，ネットワークを越えて呼び出されることが想定されるため

サービスコールを呼び出す順序

- サービスコールを特定の順序で呼び出さなければならないような設計にしない
  - 具体的には，POSIXにおけるファイル/デバイスのopen/close のようなサービスコールは設けない

(コンピュータの) OSに対する想定

(コンピュータの)OSが持つべき機能

- ビークルサービスを実現するベースとなる(コンピュータの)OSは，OSが一般的に持つ機能を備えており，アプリケーションはそれらの機能を利用できるものとする
- 特に，以下の機能を持つことを想定する
  - ECUを越えたサービス呼び出し機能とPub/Sub通信機能
  - 他のECU上で新しいプロセスを立ち上げる機能

制御の調停とリスク制御

制御の調停機能

基本方針

- 複数のアプリケーションが，同一のオブジェクトを制御（出力装置に出力）する場合の調停機能を，ビークルOSに持たせる
  - 調停に必要となるロックと操作優先度の概念を，ビークルAPIに導入する
  - コアビークルサービスには汎用的な調停機能のみを持たせ，アプリケーション特有の調停機能が必要な場合には，拡張ビークルサービスで実現する
- TODO：制御の衝突判定（同時に実行してはならないアプリケーションの検出）機能もビークルOSに持たせたいが，ロック機能で十分か要検討

標準的な調停機能

- APIコンセプトでは，多くのオブジェクトに適用できる標準的な調停機能（調停方法，ロック，操作優先度）を定義する
- 標準的な調停機能では不十分なオブジェクト（例えば，音声出力）に対しては，オブジェクト固有の調停機能を持たせる

ユースケース1）ウィンドウの制御

- ウィンドウを閉めるサービスコールが呼び出された後に，開くサービスコールが呼び出されると，最初のサービスコールによる制御は上書きされる
- 洗車モード中は，ウィンドウ開閉スイッチによるウィンドウの開閉を禁止する

→ 標準的な調停機能（ロック機能を含む）で実現可能とする

ユースケース2）AD/ADAS

- 緊急ブレーキの動作中は，ACC（オートクルーズコントロール）による制御は受け付けない
  - ACCには，制御が受け付けられなかったことがわかるようにする
- 自動運転とACCは，同時に実行してはならない

→ 標準的な調停機能（ロック機能を含む）で実現可能とする

ユースケース3）音声出力

- アプリケーションAが音声出力中に，アプリケーションBが音声出力しようとした場合は，Aの音声出力が終了後に，Bの音声出力を行う
- アプリケーションAが音声出力中に，アプリケーションBが緊急性の高い音声を出力しようとした場合は，Aの音声出力を中断し，Bの音声出力を行う
  - アプリケーションAには，音声出力が中断されたことを通知する

→ オブジェクト固有の調停機能で対応する

標準的な調停機能

後勝ちの原則

- 後に呼び出されたサービスコールによる制御が，前に呼び出されたサービスコールによる制御を上書きすることを原則とする
  - 短い時間で処理が完了するようにサービスコールを設計するとしたことから，この原則が自然
- 原則の適用例
  - ウィンドウを閉めるサービスコールが呼び出され，ウィンドウが閉じる動作中に，開くサービスコールが呼び出されると，最初のサービスコールによる制御はキャンセルされ，ウィンドウは開く動作を始める（後述の操作優先度は，これには関係しない）

ロック

- 他のアプリケーション（のインスタンス）によるオブジェクトの操作を禁止することを，ロックと呼ぶ
  - 必要なオブジェクトに対して，ロックの機能を持たせる
  - ロックは，ハードウェアで行われる場合（例えば，ドアロック）と，ソフトウェアでのみ行われる場合がある
- 利用例
  - 自動運転アプリケーションは，車両運動の縦方向制御と横方向制御をロックし，他のアプリケーションが制御できないようにする

操作優先度

- あるアプリケーションがオブジェクトをロックしている場合に，他のアプリケーションがそれをオーバーライドすることを可能にするために，操作優先度の考え方を導入する
- TODO：操作優先度の意味と数値を定める
  - 例えば，緊急アプリ（10），特権ユーザ（30），特権アプリ（50），一般ユーザ（70），一般アプリ（90）
- 操作優先度は，PriorityTypeで表す
- 以下では，操作優先度の標準的な扱いを定義する
  - 操作優先度の標準的な扱いでは，操作優先度はロックに対してのみ意味を持つ
- オブジェクトによっては，操作優先度に対して追加の意味を持たせる場合もある

操作優先度 – 続き

- ロック機能を持つオブジェクトをロック/操作するサービスコールは，オプションのパラメータとして，操作優先度を持つ
  - 操作優先度を指定しない場合，そのアプリケーションが使用できる最高操作優先度を指定したものと扱う
- 操作優先度*P*でロックされているオブジェクトを，操作優先度*P'*でロック/操作するサービスコールを呼び出した場合の振る舞いは次の通り
  - *P'*が*P*と同じかより低い操作優先度である場合，ロック/操作サービスコールは，E_OBJECT_LOCKEDエラーとなる
  - *P'*が*P*より高い操作優先度である場合，ロックが強制解除され，ロック/操作サービスコールは成功する（ロックサービスコールの場合，ロックを奪うことになる）

他の調停方法

- 先勝ち
  - 先勝ちの調停は，ロックを用いて実現できる
  - 例えば，ウィンドウを閉めるサービスコールを呼び出してからウィンドウが完全に閉まるまでは，他のアプリケーションによる制御を許したくない場合には，ウィンドウをロックした後にウィンドウを閉めるサービスコールを呼び出し，ウィンドウが完全に閉まったら（イベントで通知を受けることができる）ロックを解除すれば良い
- 操作停止
  - TODO：矛盾した操作をした場合には，操作を止める調停方法が必要という意見がある。具体的なユースケースが見つかれば検討する

リスク制御の概要

リスク制御の目的

- アプリケーションに何らかのリスクがあるAPIの使用を許すかどうかを制御すること
  - 人身に対するリスクだけでなく，財産・物品・環境に対するリスクやプライバシ上のリスクも対象とする
- アプリケーションが侵害された場合に備えて，必要なサービスコール以外は呼び出せないようにするアクセス制御の機能は，必要であれば，リスク制御とは別に検討する

リスク制御情報に関する留意点

- アプリケーションによるAPIの使用可否を制御するリスク制御情報は，車両に依存しないものとする
  - アプリケーションの審査工数が大きくなるのを防ぐため

考慮するユースケース

ユースケース1）車両の安全機能への依存性

- ウィンドウに挟み込み防止機能を持つ車両Xと，持たない車両Yがある場合を考える
- 「リスク制御情報」では，以下のようなことを表現したい
  - アプリケーションA（挟み込みのリスクに対応していない）は，車両Xではウィンドウを操作できるが，車両Yではウィンドウを操作できない
  - アプリケーションB（挟み込みのリスクに対応している）は，どちらの車両でも操作できる

→ 操作の可否は，車両（ビークルOS）側で安全性が確保されるかに依存する

- 車両の構成記述に，オブジェクト（の各インスタンス）に対して，車両側で安全性が確保されるかどうかの情報を持つことが必要

ユースケース2）操作対象と車両状態への依存性

- 運転者の前（インパネ）に設置されたディスプレイ内の（一部分の）表示領域を考える
- 「リスク制御情報」では，以下のようなことを表現したい
  - アプリケーションA（例えば，カーナビ）は，運転者が運転操作に注力する必要がある車両状態でも，その表示領域を使用できる
  - アプリケーションB（例えば，ゲーム）は，運転者が運転操作に注力する必要がない車両状態でしか，その表示領域を使用できない

→ APIの使用可否は，操作対象のオブジェクトインスタンスと，車両状態に依存する

- ただし，車両によって持っているオブジェクトインスタンスは異なるため，「リスク制御情報」中でインスタンスを指定する方法は採れない

リスク制御の実現方法

オブジェクト操作によるリスクの定義

- 操作することによって何らかのリスクが生じる可能性があるオブジェクトに対しては，Open SDV API仕様で，そのオブジェクトに対するリスククラス（「リスクを分類したもの」の意味）を定義する
  - 一般には，各オブジェクトに対して複数のリスククラスがある

例）ウィンドウに対するリスククラス

- RISK_PINCH … ウィンドウを閉めることに伴う挟み込みのリスク
- RISK_OPEN … ウィンドウを開けることに伴う諸々のリスク（さらに分類する手もある。今後の検討課題）

オブジェクト操作によるリスクの定義 – 続き

- あるリスククラスが，実際に危険を生じる可能性があるかは，操作の種類（サービスコールとそのパラメータ）や車両状態に依存する

例）ウィンドウに対するリスククラスが危険を生じない場合

- RISK_PINCH があっても，ウィンドウの状態参照やウィンドウを開く操作は危険を生じない

例）ビュー（ディスプレイの全体またはその一部）に対するリスククラスが危険を生じない場合

- RISK_DISTRACTION_CID（ドライバディストラクションのリスク）があっても，駐車状態では危険を生じない

- サービスコールの仕様において，リスククラスに対して危険を生じない条件がある場合には，それを明確にする

車両の構成記述でのリスク記述

- 車両の構成記述には，各オブジェクト(の各インスタンス)に対して，ビークルOSで対応できていない(危険を防止できていない)リスククラスの集合を記述する

例) 車両の構成記述におけるウィンドウのリスククラス

- 挟み込み防止機能を持つウィンドウ： [RISK_OPEN]
- 挟み込み防止機能を持たないウィンドウ：
  [RISK_PINCH, RISK_OPEN]

リスク制御情報でのリスク記述

- 何らかのリスクがあるAPIを使用するアプリケーションは，どのリスククラスにどのように対応しているかを申告して，OEM（またはOEMから委託された者）の審査を受ける
- 審査の結果，そのアプリケーションで対応できているリスククラスの集合を記述したリスク制御情報を作成する
- リスク制御情報には，操作するオブジェクト毎に，アプリケーションが使用できる操作優先度の最高値（OEMの審査で承認を得た値）も記述する
  - 使用できる最高操作優先度をどのように決定するかは，今後の課題
- リスク制御情報は，審査機関による署名付きで，アプリケーションと一緒に配布する

アプリケーションでのリスククラスへの対応

- アプリケーションでのリスククラスへの対応には，対策の強度が異なる複数の方法が考えられる

  例）ウィンドウのRISK_PINCHへの対応例（同様の対応を，ビークルOS側で行うことも可能）

  - 音声等で「これからウィンドウを閉めます」と伝え，ユーザの「OK」後に，閉動作を行う
  - 音声等で「これからウィンドウを閉めます」と伝え，ユーザがいつでも閉動作を停止できる手段を提供して，閉動作を行う
- リスククラスへの対応には，「ユーザに制限事項を説明する」や「アプリケーション提供者が事故に対する責任を取る」といった対応も含まれる
- アプリケーションの審査において，各リスククラスに対応したと認める基準を整備することが必要

ユーザの判断

- ビークルOSは，アプリケーションのインストール時に，アプリケーションで対応できている各リスククラスに対して，そのリスクを受け入れるかユーザに判断を求める

例）RISK_PINCHに対応しているアプリケーションのリスクを受け入れるかの判断を求めるメッセージの例

> 本アプリケーションは，ウィンドウの閉動作を行います。動作中，身体の一部や物品等が挟まれるおそれがあります。周囲の安全を確認し，異常時は直ちに停止してください。
> 本アプリケーションの使用により生じた損害について，[OEM]は責任を負いません。責任および補償の範囲は，[アプリケーション提供者]の利用規約に従います。
> アプリケーションによるウィンドウの閉動作を許可しますか？

- ユーザの判断結果を，運転者個人に紐づいたユーザプリファレンスデータ内に格納する

操作の可否の決定

- すべてのリスククラスに対応できている場合，アプリケーションはそのオブジェクトを操作できる
  - すべてのリスククラスに対応できている場合とは，ビークルOSによって対応できていないリスククラスの集合が，アプリケーションで対応できているリスククラスの集合に含まれており，ユーザがそれらのリスクを受け入れると判断した場合のこと
- 対応できていないリスククラスがある場合，危険を生じる可能性がある操作は禁止される
  - 前述の通り，危険を生じる可能性があるかは，操作の種類（サービスコールとそのパラメータ）や車両状態に依存する
- 使用できる最高操作優先度よりも高い操作優先度を用いた場合にも，操作は禁止される

サービスコールの戻り値

- サービスコールによる操作が，危険を生じる可能性があるために禁止された場合，サービスコールの戻り値は次の通りとする
  - アプリケーションが対応できていないリスククラスがあった場合は，E_RISK_APPLICATION
  - ユーザが受け入れると判断していないリスククラスがあった場合は，E_RISK_USER
- 使用できる最高操作優先度よりも高い操作優先度を用いた場合，戻り値は E_DENIED_PRIORITY とする
- 必要に応じて別途検討する「必要なサービスコール以外は呼び出せないようにする機能」により，サービスコールの呼び出しが拒否された場合，戻り値は E_DENIED_ACCESS とする

リスク制御に関する補足事項

プライバシ上のリスク

- プライバシ上のリスクも，同じ枠組みで扱う
- 例えば，自車位置・方位（CurrentLocation）に対するリスククラスには，プライバシ上のリスクを含む
  - RISK_PRIVACY … プライバシ上のリスク

リスククラスと機能安全（ISO 26262）

- リスククラスは，ISO 26262と関連づけて説明すると，「ハザードを，その重大度や対策の方法・難易度によって分類したもの」と定義することができる
  - ハザードをどれだけまとめるか（逆に言うと，どれだけ分離するか）は難しい課題。アプリケーションを想定し，ハザードの重大度と，ビークルOSおよびアプリケーションでの対策方法を検討した上で，決めることが望ましい

OEMの審査について

- OEMの審査は，国や地域毎に行う必要がある
  - 国や地域によって法制度や規制（法規制，自主規制）が異なるため
- OEMの審査は，OEM間で審査基準を統一できる場合には，複数のOEMに対して一括して行うことができる
- リスクに対するすべて責任を，アプリケーション提供者とユーザに転嫁できるのであれば，OEMの審査は必須ではない

ユーザプリファレンスデータ

- ユーザがアプリケーションのリスクを受け入れるかの判断結果は，運転者個人に紐づいたユーザプリファレンスデータ内に格納する
  - 同一の車両でも，運転者によって判断結果は異なる
  - 「設定」機能により判断結果を変更できるようにする
- TODO：複数の車両に対して同じユーザプリファレンスデータを使用できるようにするかは，今後の課題

ユーザの判断の省略

- OEMの純正アプリケーション（標準アプリケーションを含む）については，アプリケーション毎のユーザの判断を省略できるようにする
  - OEMは，車両を販売する時点で，ユーザから包括的な許可をもらうことを想定

オブジェクトに対するリスククラスの例

- ビュー（ディスプレイの全体またはその一部）に対するリスククラス
  - RISK_DISTRACTION_HUD …ヘッドアップディスプレイと同等のドライバディストラクションのリスク（運転者が見るために，視線の移動がほとんど必要ない。運転中は，運転に直結する情報のみ表示して良い）
  - RISK_DISTRACTION_IP … インパネ内のディスプレイと同等のドライバディストラクションのリスク（運転者が見るために，少しの視線移動が必要である。運転中は，運転に関連する情報のみ表示して良い）
  - RISK_DISTRACTION_CID … センターディスプレイと同等のドライバディストラクションのリスク（運転者が見るために，大きい視線移動が必要である）
  - ドライバが見ることのできないディスプレイは，ドライバディストラクションのリスクがない

オブジェクトに対するリスククラスの例 – 続き

- 運動制御(Motion)に対するリスククラス
  - RISK_COLLISION … 自車の責任で何らかの物体(他の車両，歩行者，障害物，落下物など)に衝突するリスク
  - RISK_GROUNDLOSS … 脱輪や落下のリスク
  - RISK_VIOLATION … 交通規則を違反するリスク
  - RISK_COLLISIONをさらに分類するかは今後の検討課題(ODDや走行速度によって分類する案がある)
  - この他にも，横転のリスクなどが考えられるが，車両(ビークルOS)側で対策できると思われる

車両状態の定義

車両状態の定義の必要性

- リスククラスが危険を生じる可能性がある車両状態を記述するには，車両状態が定義されていることが有用である

車両状態の定義方針

- 車両の状態（IG ON，ACC ON，OFFや，自動運転のレベルなど）によって，使用できるAPIやAPIの振る舞いは変化する
  - 一般に，車両状態は，車両（特にパワートレイン）による違いが大きく，標準化は難しいと考えられている
- Open SDV API仕様では，リスククラスが危険を生じる可能性がある車両状態の記述を容易にする目的で，車両状態を定義する

車両状態の定義

- 車両状態は，以下の3つのサブ状態の組み合わせで表現する
  - 走行状態
  - 運転状態
  - 走行環境（オプション）
- さらに，以下のサブ状態の導入について検討する
  - バッテリ状態（バッテリ残量のレベル）
  - 乗員状態（乗員が乗っているか否か）
  - 異常状態（故障，事故，それらの深刻度）

走行状態

- 走行中
  - 走行している状態（速度がゼロでない状態）
  - 運転者は運転に対して何らかの義務を負っている（運転状態がドライバレスの場合を除く）
- 停止中
  - 走行途中に一時的に止まっている状態（速度がゼロの状態）
  - 運転者は運転席を離れてはならず，運転者は周囲の状況に注意を払う必要がある（運転状態がドライバレスの場合を除く）
  - ドアを開けても良い（要検討）

走行状態 – 続き

- 駐車中
  - シフトレバーがパーキング位置にある状態。または，パーキングブレーキを掛けている状態
  - 運転者は運転席を離れても良い
- 電源OFF
  - パワートレインの電源が切れている状態
    - ✓いわゆるアクセサリ（ACC）モードは，ここに含める

運転状態

- 手動運転(レベル1，ハンズオフ可能でないレベル2)
  - 運転者が何らかの運動制御を行う状態。運転者は，安全運転に注力する義務がある
- ハンズオフ(ハンズオフ可能なレベル2)
  - 車両の運動制御は自動化されているが，運転者は常に周囲の状況を監視し，システムが対応できない状況に備える必要がある状態
- アイズオフ(レベル3)
  - 車両の運動制御は完全に自動化されており，運転者は運転操作に注力する必要がない状態。ただし，運転者は運転席に座っている必要があり，システムからの操作引き継ぎ要求に速やかに対応する義務がある
- ドライバレス(レベル4～5，ブレインオフ)
  - 車両の運動制御は完全に自動化されており，運転者がいなくても車両が安全に運行できる状態

走行状態と運転状態の依存性

- 走行状態が「駐車中」または「電源OFF」の場合，運転状態は「ドライバレス」とする
  - これにより，サービスコールを使用できる条件をシンプルに記述できるようになる

走行環境（オプション）

- 公道走行状態
- 非公道走行状態（通称：サーキットモード）
  - 公道でない道路を走行している場合に使用できるAPIはあるものと思われる
- 走行環境の状態を持たない車両があってもよい
  - 走行環境の状態を持たない車両においては，走行環境に対する条件があるサービスコールは，許可されない（言い換えると，すべての走行環境で呼び出せるサービスコールのみ使用できる）
- 走行環境をより細かく分類することも考えられる

  例）自動車専用道と一般道

  例）走行している国・地域（適用される交通ルール）

その他の車両状態

- バッテリ状態（バッテリ残量のレベル）
  - 審査の結果，あるAPIを，バッテリ残量によっては使用できない（または使用できる）という状況があるか？
- 乗員状態（乗員が乗っているか否か）
  - 審査の結果，あるAPIを，乗員が乗っている状態（または乗っていない状態）では使用できない（または使用できる）という状況があるか？
- 異常状態（故障，事故，それらの深刻度）
  - 審査の結果，あるAPIを，異常状態では使用できない（または使用できる）という状況があるか？
  - 導入する場合には，異常状態のレベルを設ける（2～3段階程度？）

共通規定

共通データ型

ReturnType（現時点）

```
type ReturnType = enum {
    E_OK,                   // 正常終了
    E_INVALID_ID,           // IDが不正
    E_INVALID_HANDLE,       // ハンドルが不正
    E_INVALID_PARAMETER,    // その他のパラメータが不正
    E_RISK_APPLICATION,     // アプリケーションが対応していないリスクがある
    E_RISK_USER,            // ユーザがリスクを受け入れていない
    E_DENIED_PRIORITY,      // 使用できない操作優先度
    E_DENIED_ACCESS,        // アクセスが認められていない
    E_OBJECT_FAULT,         // 対象オブジェクトが故障中
    E_OBJECT_LOCKED,        // 対象オブジェクトがロックされている
    E_OBJECT_STATUS,        // 対象オブジェクトが指定された操作をできない状態に
                            // ある
    E_NO_DATA,              // データがない
    E_NOT_SUPPORTED,        // サポートされていない機能
    E_NOT_OK                // その他のエラー
}
```

- 今後，必要に応じてエラーの種類を追加する

IdType

- オブジェクトのインスタンスIDの型

```
type IdType = UInt16 ([1,65535]);
```

- オプショナル型をサポートしていないプログラミング環境では，IdType?でnullを表すために，値0を使用する
  - ID_NULL（＝0）で，IDが指定されていないことを示す

PriorityType

- 操作優先度の型

```
type PriorityType = UInt8 ([1,100]);
```

  - 値が小さい方が，優先度が高い
- オプショナル型をサポートしていないプログラミング環境では，PriorityType?でnullを表すために，値0を使用する
  - PRIORITY_NULL（＝0）で，操作優先度が指定されていないことを示す

ApplicationIdType

- アプリケーションのインスタンスの識別子の型
- 具体的な型は，物理APIにおいて規定する
- TODO:ビークルOS内の安全機構を表す特殊値を表現できるようにする案もある
  - これにより，安全機構による状態変化のイベントを，特殊扱いする必要がなくなるため

ObjectKindType

- オブジェクトの種別(クラス)を表す型

```
type ObjectKindType = enum {
    VEHICLE,
    BODY,
    CABIN,
    DOOR,
    WINDOW,
    ........
};
```

RelatedObjectType

- 関連するオブジェクトを表す型
- 以下の3種類の定義のいずれか(物理APIで選択)

```
type RelatedObjectType = String;
```

```
type RelatedObjectType = struct {
    objectKind : ObjectKindType,
    instanceName : String?
};
```

```
type RelatedObjectType = struct {
    objectKind : ObjectKindType,
    instanceId : IdType?
};
```

PlacementType

- キャビン内のオブジェクトの場所を表す型

```
type ZoneType = enum {
    LEFT,        // 左
    RIGHT,       // 右
    CENTER,      // 中央
    TOP,         // 上
    BOTTOM       // 下
};

type PlacementType = struct {
    row : UInt8,          // 座席の何列目か
    zone : ZoneType       // 左右上下方向の位置
};
```

すべてのオブジェクトが持つサービスコール

オブジェクトの構成参照サービスコール（シングルトンの場合）

- 名称：getConfig
- パラメータ
  - なし
- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類
config	*xxx*ConfigType	対象オブジェクトの全構成情報

- 機能
  - 対象オブジェクトの全構成情報を取得する

オブジェクトの構成参照サービスコール（マルチインスタンスの場合）

- 名称：getConfigAll
- パラメータ
  - なし
- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類
config	*xxx*ConfigType[]	対象オブジェクトのすべてのインスタンスの全構成情報

*xxx*ConfigTypeの可変長配列

- 機能
  - 対象オブジェクトのすべてのインスタンスの全構成情報を取得する

オブジェクトの状態参照サービスコール

- 名称：getStatus
- パラメータ

instanceId	IdType	操作対象のインスタンスID（マルチインスタンスの場合のみ）

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類
status	*xxx*StatusType	対象オブジェクトの状態

- 機能
  - 操作対象のオブジェクト（のインスタンス）の状態を取得する

イベント通知要求サービスコール

- 名称：notify
- パラメータ

instanceId	IdType?	イベント通知対象のインスタンスID（マルチインスタンスの場合のみ）
eventFilter	enumSet(*xxx*EventKindType)?	通知するイベントの種類の集合

  - 通知するイベントの種類をより細かく指定するために，パラメータを追加しても良い

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類
notifyHandle	NotifyHandleType	生成したイベントキューの通知ハンドル

イベント通知要求サービスコール – 続き

- 機能
  - イベント通知対象のオブジェクト(のインスタンス)で発生した eventFilter で指定したイベントの通知を要求する
  - イベントを格納するためのイベントキューを生成し，その通知ハンドルを notifyHandle に返す
  - instanceId を省略した場合，すべてのインスタンスで発生したイベントの通知を要求する
  - eventFilter を省略した場合，すべての種類のイベントの通知を要求する

イベント通知停止サービスコール

- 名称：unnotify
- パラメータ

  notifyHandle　NotifyHandleType　削除するイベントキューの通知ハンドル

- リターンパラメータ

  returnValue　ReturnType　正常終了またはエラーの種類

- 機能
  - notifyHandleで指定した通知を停止する
  - イベントを格納するためのイベントキューを削除する

イベント取得サービスコール

- 名称：getEvent
- パラメータ

notifyHandle	NotifyHandleType	イベントキューを識別するための通知ハンドル

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類
event	*xxx*EventType	取得したイベント

- 機能
  - notifyHandleで指定した通知用のイベントキューから，先頭のイベントを取り出す
  - 取り出したイベントは，イベントキューから削除される
- イベントがパラメータ等で渡されてくるプログラミング環境向けの物理APIでは，用意する必要がない

LockableObject

LockableObject の概要

- API仕様の記述を簡潔にするために，標準的なロック機能を，LockableObject として定義する
  - 「このオブジェクトは LockableObject である」という記述は，標準的なロック機能を持つことが意味する。これにより，ロック機能の説明を繰り返すことが避けられる
  - LockableObject は，インタフェースのインポートのようなものである（多重インポートもあり）。親クラスのようなものとも言えるが，実装を継承するとは限らない
- LockableObject は，以下のAPIを持つ
  - ロック／ロック解除のサービスコールを持つ
  - 状態参照によりロック状態を参照できる
  - ロック操作に関するイベントを持つ

ロック状態

- LockableObject は，オブジェクト状態を参照するサービスコール（getStatus）により，ロック状態を参照することができる
- ロック状態には，以下の情報を含む
  - ロックされているか否か（ロックされている場合に true）
  - ［ロックされている場合］ロックしたアプリケーションのインスタンスのID
  - ［ロックされている場合］ロック操作を行った操作優先度
- ロック状態のデータ型（LockStatusType）

```
type LockStatusType = struct {
    locked : Bool,                          // ロックされているか否か
    lockApplication : ApplicationIdType?,   // ロックしたアプリケーション
    lockPriority : PriorityType?            // ロック操作を行った操作優先度
};
```

ロックサービスコール

- 名称：lock
- パラメータ

instanceId	IdType	ロック対象のインスタンスID（マルチインスタンスの場合のみ）
priority	PriorityType?	操作優先度

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類

- 機能
  - ロック対象のオブジェクト（のインスタンス）を，指定した操作優先度でロックする

ロック解除サービスコール

- 名称：unlock
- パラメータ

instanceId	IdType	ロック対象インスタンスのID（マルチインスタンスの場合のみ）

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類

- 機能
  - ロック解除対象のオブジェクト（のインスタンス）を，ロック解除する

ロック操作に関するイベント

- LockableObject は，以下のイベントを持つ
  - OWN_LOCK_REVOKED
    - ✓自分がロックしているオブジェクトが，他のアプリケーションによって強制的にロック解除された
  - OTHER_LOCK_RELEASED
    - ✓自分がロックしていないオブジェクトが，他のアプリケーションによってロック解除された

MovableObject

MovableObject の概要

- API仕様の記述を簡潔にするために，移動する可動部を持つオブジェクトに共通の機能を，MovableObject として定義する
  - 「このオブジェクトは MovableObject である」という記述は，ここで説明する機能を持つことを意味する
  - 参考：COVESAでは，MovableItem という用語が使われている
- MovableObject は，以下のAPIを持つ
  - 状態参照により移動に関する状態を参照できる
  - 移動を開始／停止するサービスコールを持つ
  - 移動に関する状態の変化のイベントを持つ

MovableObject の機能

- MovableObject は一次元に移動する可動部を持つ
  - 二次元（以上）に移動する可動部を持つオブジェクトも，MovableObject を拡張する形で規定できる
- 可動部の移動には時間がかかる
- MovableObject は，可動部の現在位置と目標位置を状態に持つ
- 現在位置と目標位置が一致していない場合，目標位置に向けて可動部が移動し，現在位置と目標位置が一致したら，移動を停止する
  - 現在位置の検知精度が低いなどの理由で，現在位置が目標位置に完全には一致させられない場合，目標位置を変更して，現在位置に一致させる
- 現在位置と目標位置は，パーセンテージで表す
  - 片方の移動端を0，反対側の移動端を100で表し，どちらの移動端を0とするかはオブジェクト毎に定める

MovableObject の機能 – 続き

- 現在位置の検知精度は，車両に依存する
  - 現在位置が全くわからない場合，現在位置とし UNKNOWN を返す
  - 現在位置が，0 で表される移動端でないことのみがわかっている場合，現在位置として NONZERO を返す
  - 現在位置が，どちらの移動端でもないことがのみがわかっている場合，現在位置として 50 を返しても良い
- 目標位置の設定精度は，車両に依存する
  - 目標位置に 0 または 100 を指定した場合，指定された移動端まで移動する
  - 目標位置にそれ以外の値を指定した場合，最終的な位置は移動端にはならない（移動端まで移動してから，少し戻る可能性はある）

位置を表すデータ型（PositionType）

- MovableObject の可動部の位置をパーセンテージで表すデータ型
  - 0と100の意味は，オブジェクト毎に定める（例えばウィンドウの場合，0は全閉，100は全開）

```
type PositionType = Int8 ([0,100] | UNKNOWN | NONZERO)
```

  - UNKNOWN（＝–1）は，位置が不明であることを示す
  - NONZERO（＝–2）は，0 で表される移動端でないことを示す

移動に関する状態

- MovableObject は，オブジェクト状態を参照するサービスコールにより，移動に関する状態を参照することができる
- 移動に関する状態には，以下の情報を含む
  - 移動主状態
  - 現在位置
  - 目標位置
- 移動主状態は，以下のいずれか
  - 停止中
  - 位置のパーセンテージが大きくなる方へ移動中
  - 位置のパーセンテージが小さくなる方へ移動中
  - 故障中
    - ✓恒久故障（修理するまで治らない）と一時故障（自然に治る可能性がある）は区別しない

移動に関する状態のデータ型

- 移動主状態

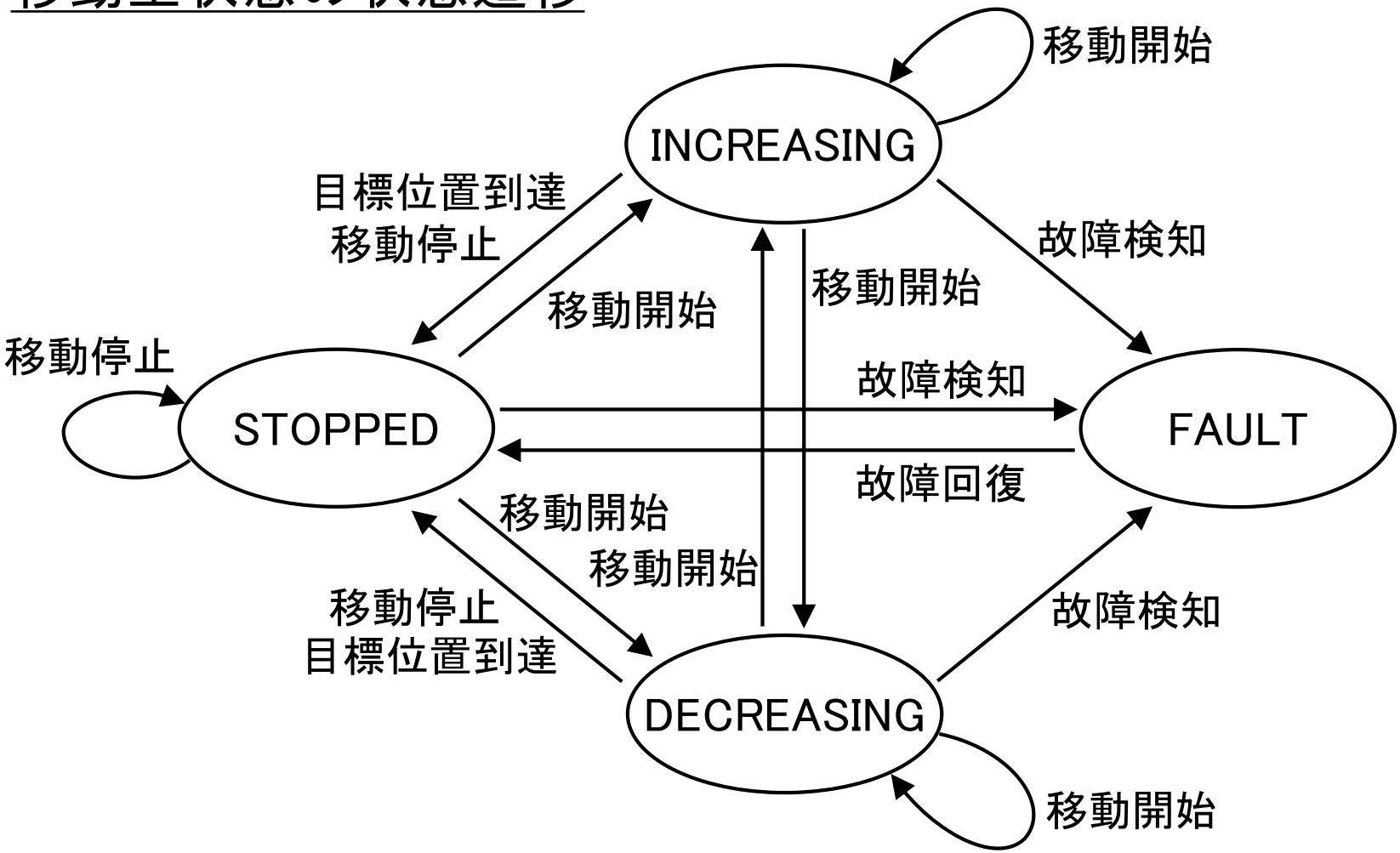
```
type xxxMainStatusType = enum {
    STOPPED,          // 停止中
    INCREASING,       // 位置のパーセンテージが大きくなる方へ移動中
    DECREASING,       // 位置のパーセンテージが小さくなる方へ移動中
    FAULT             // 故障中
};
```

  - INCREASING と DECREASING は，オブジェクトの動きに合わせて名前を変えても良い

- 移動に関する状態（xxxStatusTypeの一部分）

```
type xxxStatusType = struct {
    mainStatus : xxxMainStatusType,      // 移動状態
    position : PositionType,             // 現在位置
    targetPosition : PositionType,       // 目標位置
    .........
};
```

移動主状態の状態遷移



- この他に，オブジェクト固有の状態遷移があっても良い

移動開始サービスコール

- 名称：startMove
- パラメータ

instanceId	IdType	操作対象のインスタンスID（マルチインスタンスの場合のみ）
targetPosition	PositionType	目標位置
priority	PriorityType?	操作優先度

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類

- 機能
  - 目標位置を targetPosition に設定して，操作対象の MovableObject の可動部の移動を開始する

移動停止サービスコール

- 名称：stopMove
- パラメータ

instanceId	IdType	操作対象のインスタンスID（マルチインスタンスの場合のみ）
priority	PriorityType?	操作優先度

- リターンパラメータ

returnValue	ReturnType	正常終了またはエラーの種類

- 機能
  - 操作対象の MovableObject の可動部の移動を停止する

MovableObject が持つイベント

- MovableObject は，以下のイベントを持つ
  - TARGET_REACHED
    - ✓指定された目標位置に到達して停止した
  - CONTROLLED_BY_OTHER
    - ✓他のアプリケーションが制御した
    - ✓アプリケーションのインスタンスのIDと，制御に使用したサービスコールの種類を，パラメータに持つ
  - FAULT_DETECTED
    - ✓故障が検知された
  - FAULT_RECOVERED
    - ✓故障から回復した
- 他の理由で移動主状態が変化するオブジェクトの場合には，その移動状態の変化に関するイベントを追加する

MovableObject が持つイベントのデータ型

- イベント種別（*xxx*EventKindTypeの一部分）

```
type xxxEventKindType = enum {
    TARGET_REACHED,          // 目標位置に到達
    CONTROLLED_BY_OTHER,     // 他のアプリケーションによる制御
    FAULT_DETECTED,          // 故障検知
    FAULT_RECOVERED,         // 故障回復
    .........
};
```

- イベント（*xxx*EventTypeの一部分）

```
type xxxEventType = struct {
    instanceId : IdType;    // マルチインスタンスオブジェクトの場合のみ
    eventInfo : union(xxxEventKindType) {
        controlledByOther : struct {
            sourceApplication : ApplicationIdType,
            sourceService : WindowServiceType
        }
    }
    .........
};
```