OROCOS: Open Robot Control Software

Гончаров Олег Игоревич

Факультет вычислительной математики и кибернетики,
Московский государственный университет имени М.В. Ломоносова

2012

История проекта OROCOS

Полное название "Open RObot COntrol Software".

- Концепция предложена: Herman Bruyninckx (University of Leuven) в 2000 г.

  Причина — трудности использования коммерческого ПО для роботов в исследованиях.

- В 2001 г. получен двухгодичный грант от EU. Три рабочие группы (Бельгия, Франция, Швеция).
- В 2003 г. первая рабочая версия.
- В 2008 г. издана версия 1.0.0
- На 2013 г. последняя версия 2.6.0

Основная цель: создание свободной **компонентно-ориентированной** платформы для разработки систем управления реального времени для приложений робототехники.

Архитектура OROCOS

Основное требование:

- Работа в реальном времени: обеспечит работу задач PB и обмен данными. Поддержка LinuxRT, RTAI, Xenomai.

Особенности:

- Компонентно-ориентированная система: нескольких взаимодействующих компонент со стандартными интерфейсами.
- Настройка системы: настройка компонент и их соединение происходит на этапе времени выполнения при помощи скриптов или XML.
- Мониторинг системы: интерфейс компонент доступен во время выполнения для средств мониторинга.
- Распределенная система: процесс — несколько процессов — несколько машин. CORBA.
- Внешние интерфейсы: интерфейс CORBA. OROCOS встроена в ROS, кодогенератор MatLab.

Области применения

- Совместное управление несколькими роботами.
- Мобильные роботы: системы управления для автомобилей.
- Идентификация динамики и нагрузки. Управление с учетом динамики.
- Управление по силе.
- Визуальная обратная связь.

Состав OROCOS

Ядро платформы

Orocos Toolchain — набор средств для создания приложений.

- Real-Time Toolkit (RTT) библиотека для написания компонент
- Orocos Component Library компоненты для развертывания и настройки (`deployer`, `scripting`), мониторинга (`Reporter`, `Logger`, `TaskBrowser`).
- OroGen и TypeGen обеспечивают регистрацию в системе пользовательских типов данных.

Состав OROCOS

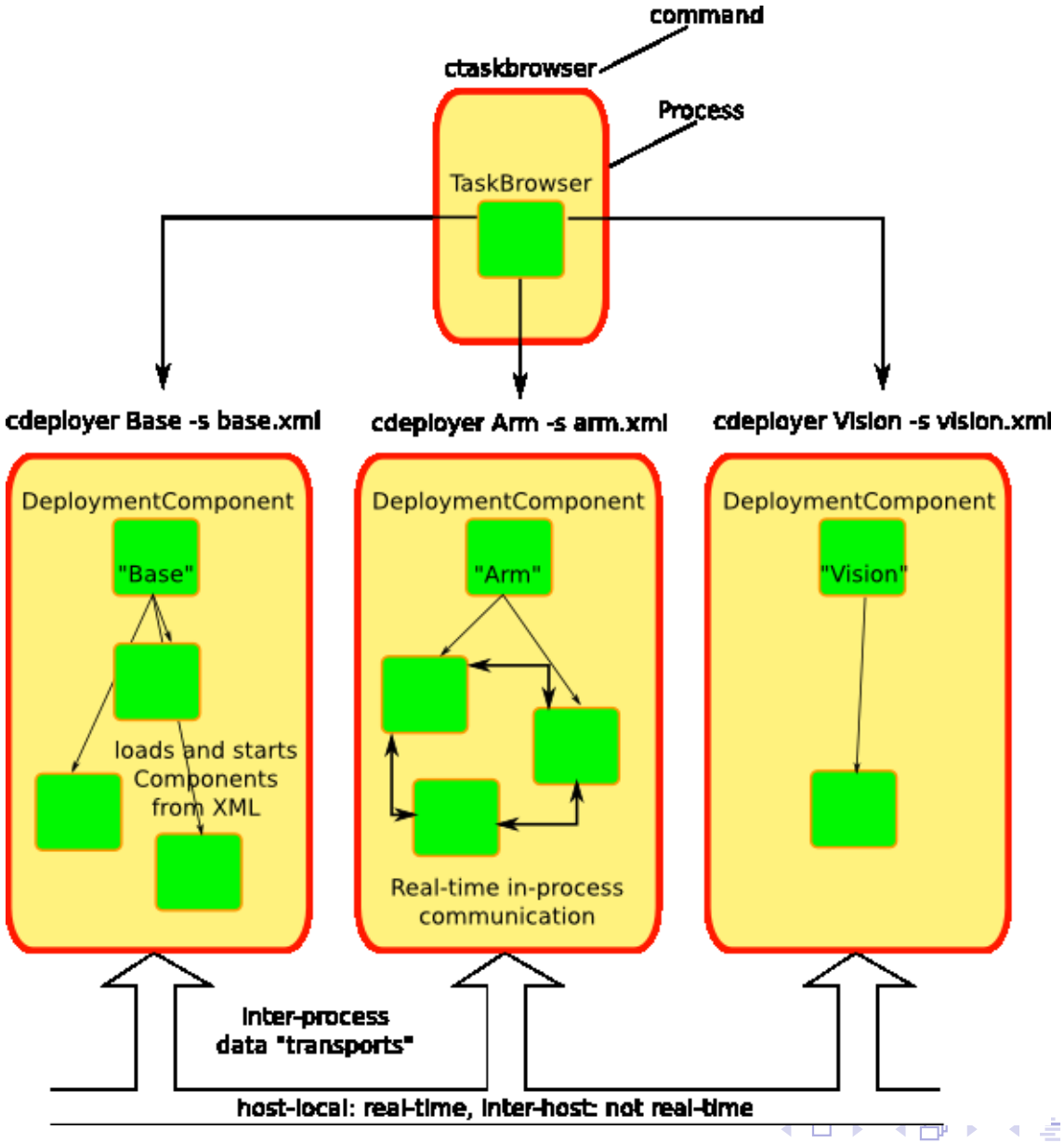
Вспомогательные библиотеки и компоненты

- Kinematics and Dynamics Library (KDL): описание кинематической модели, прямая и обратная задача кинематики.
- Bayesian Filtering Library (BFL): фильтры Байеса, Калмана и фильтры частиц.

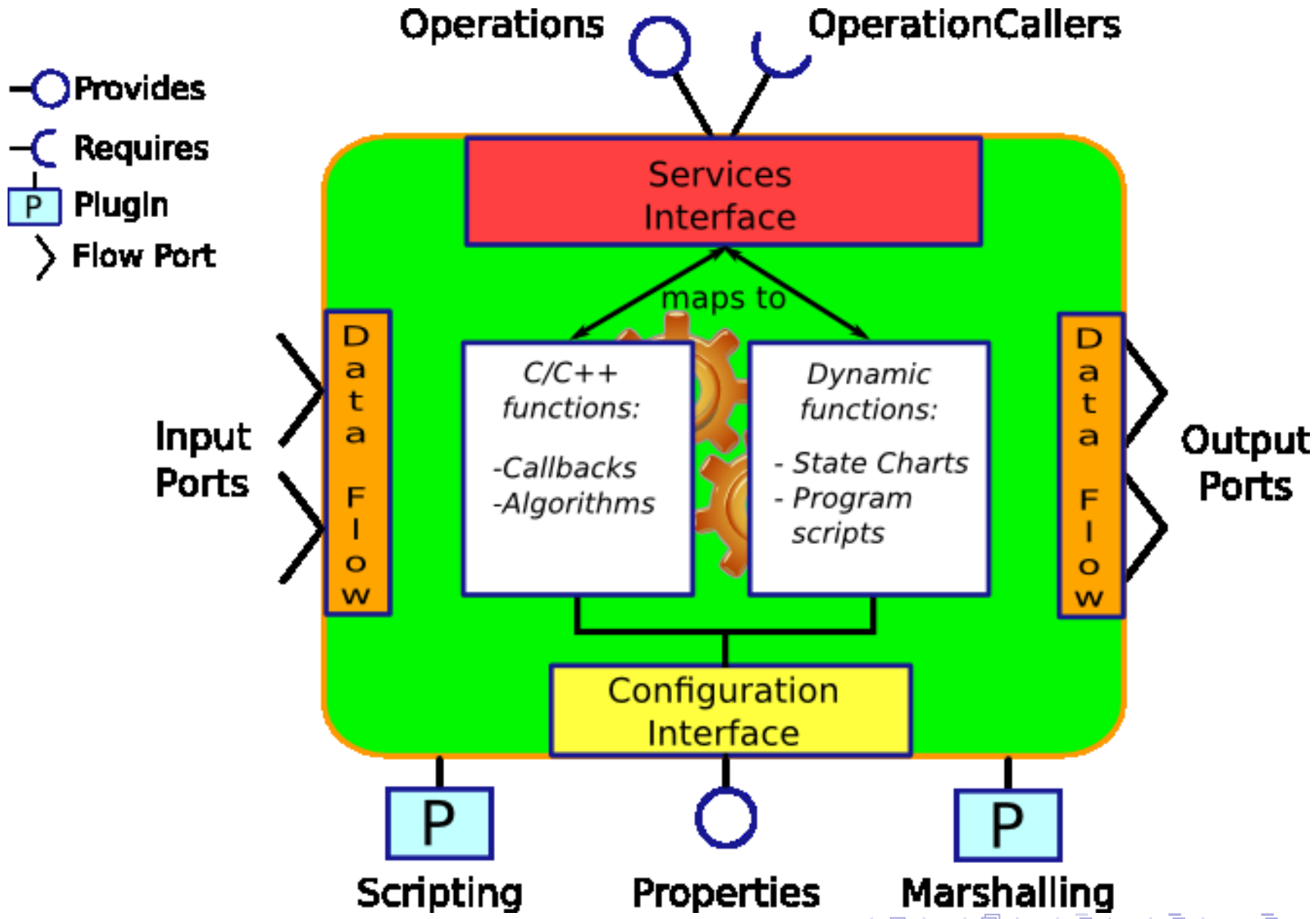
Не входят в OROCOS:

- ROS integration: `rtt_ros_integration` позволяет запускать компоненты OROCOS.
- Orocos Simulink Toolbox: генерация компонента OROCOS по модели Simulink.
- Средства для просмотра, анализа и отображения логов и сохраняемых данных.

Структура приложения



Интерфейс компонента



Интерфейс компонента

Абстрагируется классом `Service`.

- Интерфейс одного компонентам доступен другому компонентам, если он находится в области видимости. В том числе и для удаленных компонент.
- Интерфейс доступен из скриптового языка. Для мониторинга можно использовать `taskbrowser`.
- Типы данных должны быть зарегистрированы в системе: `typegen` и `typekits`. (Поддержка структур и `std::vector`, в автоматическом режиме не поддерживает `int64` и `enum`).

Интерфейс может быть расширен путем добавления расширений (plugins). Расширение обычно добавляется как дочерний сервис и является наследником класса `Service`.

Интерфейс **документируется** при помощи методов `doc`.

Интерфейс компонента: порты

- Операция записи/чтения всегда реального времени (для типов данных с “простым” копированием).
- Потоковая безопасность, отсутствие блокировок.
- Различные режимы работы:
  - буфер на одно значение,
  - очередь
- Различные виды транспорта (`Connection Policy`):
  - typelib между нитями, PT
  - mqueue сообщения, между процессами, PT
  - CORBA по сети, не является PT

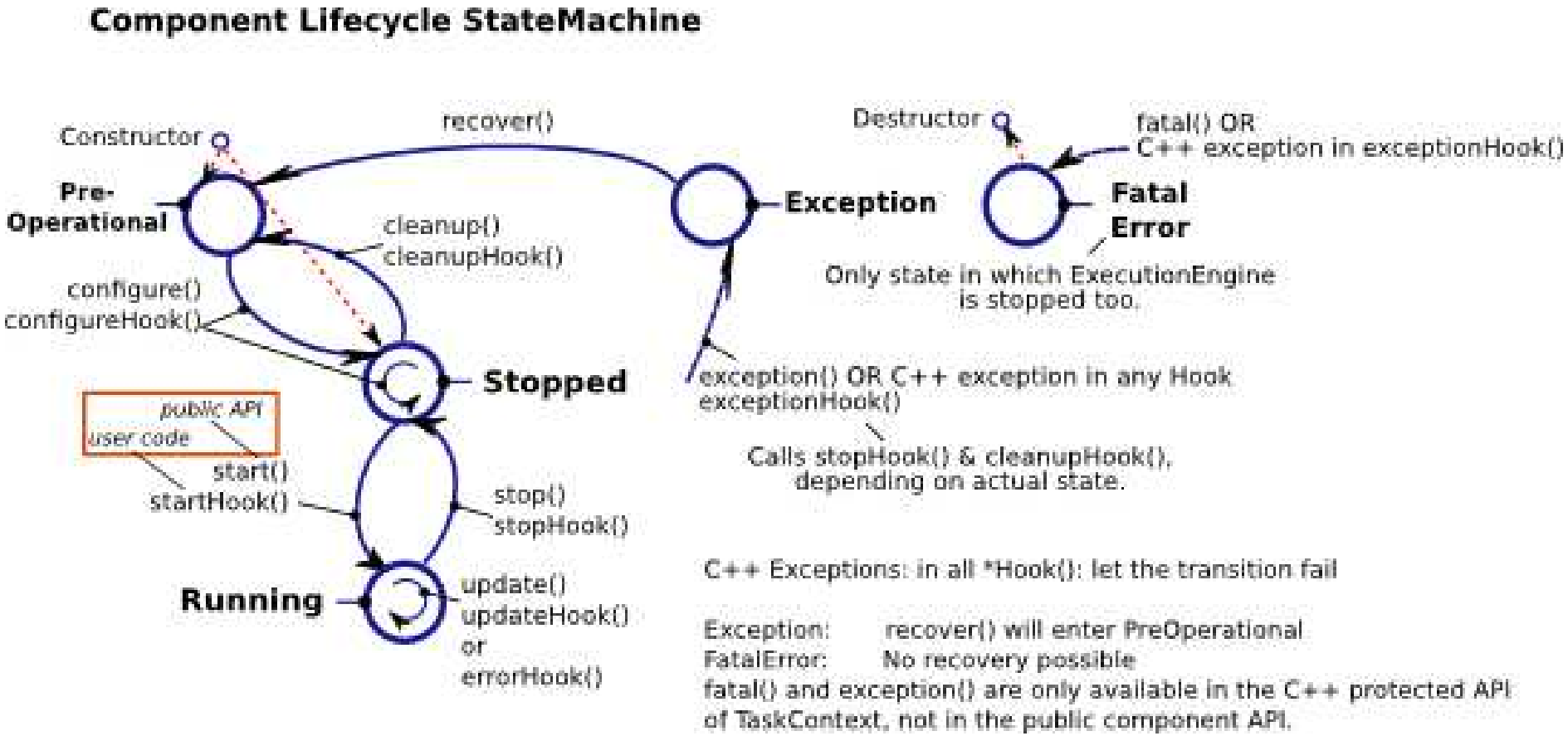
Интерфейс компонента: параметры и атрибуты

Имеют ту же семантику, что и поля классов.

- Предназначен для конфигурации компонента.
- Не являются потокобезопасными: компонент должен быть остановлен.
- Для параметров: импорт и экспорт из XML файлов (`marshalling`)

Состояния компонента

Компонент ведет себя как конечный автомат.



Интерфейс компонента: операции (методы)

Имеют ту же семантику, что и методы классов.

- На стороне клиента: `OperationCaller`
  - метод `Call` — синхронное исполнение, потокозащищено по отношению к клиенту.
  - метод `Send` — асинхронное.
- На стороне сервиса: `Operation`
  - флаг `ClientThread`: метод исполняет клиент или `GlobalExecutionEngine` (асинхронное исполнение).
  - флаг `OwnThread`: метод исполняет `ExecutionEngine` сервиса, потокозащищено по отношению к сервису.

Работа компонента

Любой компонент представляет собой реализацию класса `TaskContext`.

- Исполнение кода компонента: В компонент входит `ExecutionEngine`, исполняемая отдельной нитью `Thread` ассоциированной в `Activity`.
  - В состоянии Running: `ExecutionEngine` обрабатывает события, переданные на исполнение операции и вызывает `updateHook()`.
  - В остальных случаях методы исполняются клиентом или `GlobalExecutionEngine`.
- Характер вызовов `ExecutionEngine` управляется `Activity`:
  - переодический,
  - по появлению новых данных в портах,
  - по прерыванию.
- С `Activity` ассоциирована `Thread`:
  - планировщик и приоритет

Развертывание приложения

- Статическое развертывание.
- Динамическое развертывание: под управлением компонента `Deployer`.

Последовательность операций:

1. Запускаем программу `deployer`.
2. Загружаем компоненты.
3. Настраиваем параметры и соединяем порты.
4. Запускаем компоненты.

- Конфигурация описывается скриптом или XML файлом.
- Конфигурация может быть изменена интерактивно через `taskbrowser`.