Robot development: from components to systems

Sylvain Joyeux, Jan Albiez

Robot development: from components to systems

Sylvain Joyeux and Jan Albiez
sylvain.joyeux@dfki.de,

1 Introduction

While the last twenty years have seen the development of numerous robot integration frameworks, the last years have seen some kind of a consolidation. The obvious suspect being ROS [11], with trailing behind Orocos/RTT [13], OpenRobots and OpenRTM-aist; there are also older frameworks like MCA [12], which seem to have stopped their development, but provided basic ideas the aforementioned systems. This is list by no means meant to be exhaustive, it just lists – from the point of view of the authors – what component layers are getting adopted in the community to date.

Two main trends exist in these integration frameworks: communication layers and component-based frameworks. In a nutshell, component-based layers try to separate the concerns of developping a component and deploying a system by giving little control to the component itself on how it gets integrated with the other components, while communication-oriented frameworks will share the responsibility of integration between the component and the system deployer. Modern component layers (mostly) offer a superset of the functionality offered by communication paradigms, at the price of steeper learning curves. One could say that ROS would go in the first category while OROCOS and GenoM would go in the second one, but – as always – nothing is that clear-cut.

The purpose of this paper is to get back one step and consider the bigger picture of a system, regardless of the actual underlying integration layer. It will focus on two specific aspects:

- language design and implementation. We will discuss the usage of so-called embedded DSLs, i.e. domain-specific languages that are not implemented as separate languages, but as libraries in an existing language (usually, a dynamic language)
- techniques to model and specify systems of components. We will present a novel system integration paradigm that adapts modern programming techniques that have become common in the software development community.

This discussion will be conducted through the prism of the Robot Construction Kit (Rock[1]) that has been developped in the last years on top of the Orocos/RTT component layer at the DFKI Robotics Innovation Center in Bremen.

The next section will focus on programming paradigms and the usage of eDSLs in robotic systems. The example of the Roby plan management layer, that Rock uses as its main execution and monitoring layer, will be given as an example. Then, our approach for system modelling and deployment will be presented. The paper will finally be concluded by discussing guidelines about component development and will discuss future work.

2 About model-based and "explicit" programming

The task of managing a complete robotic system has very early been identified as a challenge to classical programming paradigms. Specific methodologies, languages and programming environments have been designed to tackle that issue. Among those, one can identify three main trends.

[1] `http://rock-robotics.org`

Specialized languages the Procedural Reasoning System (PRS [6]) and the Execution Support Language (ESL [4]) were designed as novel ways to structure programs to ease the handling of the highly asynchronous and parallel nature of the component layers in robotic systems.

Formal languages and formal methods More recently, formal programming languages such as Lustre or Esterel [9] have been recently been used to replace other, non-formal, languages such as PRS or ESL. The immediate gain of using these languages is that one can provide formal proofs of correctness on the written programs. Related to these approaches is the design of the component-centric BIP [1] framework, in which one models all possible interactions between all the components in order to generate a model-checked controller *for the complete system*.

Planning-centric approaches IDEA and its successor, T-REX [10] have been built with the underlying idea of managing the robotic system using solely planning techniques. In IDEA, multiple planning agents are organized organically to achieve a global distributed planning process, while T-REX replaces the organical structure by a much more hierarchical one.

The common trait of all these approaches is that, from an architectural point of view, there is a strong separation of level of details: to manage the component layer, one only has at its disposition the representations and tools that the language / method / tool offer *and nothing else*. Moreover, when dealing with model-based methods, it is even limited to what can be represented in said tool. For instance, if using T-REX, one is limited to non-probabilistic, non-conditional reasoning because of the underlying plan representation (so-called plan database).

To alleviate this issue, multiple components are being used, handling different aspects of the reasoning. The main critic against that approach is that the resulting system has no global point of view (see [8] for a discussion of some implications), that it involves ad-hoc integration of tools and representations (as opposed to reusable integration of reasoning components) and that it lacks a fair amount of flexibility that has become critical to successfully exploit the possibilities offered by the modern results in data processing and in control. Moreover, in the case of formal methods, one has the issue of exponential explosion, and that one must therefore even reduce the scope of the formal method to what the method can handle on "standard" computing resources.

2.1 Embedded DSLs: using general programming languages to design special-purpose frameworks

A programming technique has taken hold in the last ten years in the software engineering community, not in small parts thanks to the mainstream adoption of dynamic languages such as Ruby and Python and of the appearance of the Ruby on Rails web development framework.[2]

The idea behind embedded DSLs is to use both the flexibility of syntax and the metaprogramming abilities of these dynamic languages to create new special-purpose "languages" that are, in fact, simply an API that is crafted in a special way.

For instance, the component generation tool used in Rock, oroGen, is using a Ruby-based embedded DSL (eDSL in short) for its specification language. See Fig. 1 to see how the eDSL is equivalent to a more classical programming syntax.

Using embedded DSLs has multiple advantages:

extensible since statements in the eDSL are methods on the objects, extending an eDSL implemented in Ruby (i.e. implementing plugins) is simply a matter of adding methods / attributes to existing classes – something that is allowed by the Ruby language

reflexivity the one-to-one mapping between the description files and the API ensures that the API is constructive and descriptive enough to allow access to the models, as well as their online modification

[2] the authors acknowledge that the ability to create embedded DSLs – and the idea of doing so – is much older. Some, for instance, existed within smalltalk. What we are talking about here is the mainstream development of this technique.

```
task_context 'BaseTask' do
  output_port('solution', '/gps/Solution').
    doc "the GPS solution as reported by the hardware"
  output_port('position_samples', '/base/samples/RigidBodyState').
    doc "computed position in m"
  error_states :IO_ERROR, :IO_TIMEOUT
  property("utm_zone", "int", 32).
    doc "UTM zone for conversion of WGS84 to UTM"
end
```

```
project = Project.new
task = project.task_context 'BaseTask'
p = task.output_port('solution', '/gps/Solution')
p.doc="the GPS solution as reported by the hardware"
p = task.output_port('position_samples', '/base/samples/RigidBodyState')
p.doc="computed position in m"
task.error_states :IO_ERROR, :IO_TIMEOUT
p = task.property("utm_zone", "int", 32)
p.doc="UTM zone for conversion of WGS84 to UTM"
```

Figure 1: eDSL (top) and classical syntax (bottom). Both are completely equivalent: the eDSL emerges simply from the host language's flexibility in the syntax (here, Ruby)

ability to bind programming and models while the oroGen eDSL presented above is purely declarative, eDSLs have the added advantage that one can easily link the model and the implementation

reuse of the parser and type system of the host language one thing that everyone has to do when creating a new programming language is to implement a type system and a parser. Using eDSLs, one can reuse the type system of the host language, and focus on the functionality

The following subsection will present a more complete example of an eDSL based on Ruby. This example is grounded on the Roby plan management framework, which is used in Rock as its main system execution and monitoring layer. Then, section 3 will present a novel layer that is built on top of Roby for the purpose of modelling, deploying and managing networks of components.

2.2 Embedded DSL implementation in the Roby plan manager

In [7], we present the general model of the Roby plan manager. In this section, we will present the key aspects of the *implementation* of this plan manager. In particular, how using of an eDSL allowed to very easily link a model-based representation with the actual implementation within a programming environment.

In a nutshell, the core of the Roby plan model is a representation of long-time processes (*tasks*) that are linked together in so-called relation graphs. These relation graphs represent how processes interact with each other. Moreover, to add a notion of progression, *events* are defined, which represent specific achievements during the task execution. For instance, standard events are `start`, `stop`. To be able to control the tasks, events are defined as either *contingent* (they can not be controlled) or *controlable* (the model offers a command that will make the event happen).

For instance, a task with a controlable start event is a task that can be started by the plan manager itself. One with a contingent start event would be a task that can only be started by something outside of the plan manager.

```
1  class MoveTo < Roby::Task
2    argument :target
3    event :start do
4      emit :start
5    end
6    on :start do
7      puts "started #{self}"
8    end
9    event :blocked
10   poll do
11     if State.pos.x < 10
12       emit :blocked
13     end
14   end
15 end
```

L1 models are represented by classes. The base Roby task model is therefore represented by the Roby::Task class. This line creates a submodel called MoveTo by subclassing Roby::Task.
L2 calls a class method defined on MoveTo to add a new argument to the class that represents the MoveTo model.
L3-L5 *overrides* the default start event and makes it controlable by defining in **L3-L5** the code block (program) that is responsible for starting the task
L6-L8 defines a code block that will be called each time a MoveTo task has been started (emitted event)
L9 defines a new contingent event
L10-L14 defines a code block that is called periodically to translate state changes and messages into task events (here, monitoring of the system's progression)

Figure 2: Broken down definition of a new task model in Roby

Model representation in the Ruby programming language A generic way to represent, in the Ruby programming language, a model-based system such as the one of Roby is to map models and instances to the already existing type system of the language: classes and objects[3]. It results in a natural way to both create new models and add programming code to these models (Fig. 2).

An important effect of this usage of classes as representation of models is that one can also define "helper methods", i.e. methods that make it simpler to manage or extend a certain category of tasks by extending the class that represents that particular category (the base class of all the task models in that category).

2.3 Programming errors and safety

One natural concern about mixing a model-based approach with a programming approach is the one of safety: how to make sure that programming errors won't leak into the general system management. These concerns can be addressed easily in a system like Roby. In Roby, errors are represented as objects that are part of a certain context. This context can be a task, a set of tasks or a specific event. When an error appears, various mechanisms allow to (i) handle it and let the system continue or (ii) kill the tasks that are affected to avoid long-term effects.

In this representation, any language exception (Ruby exception) originating from the code in the models (such as: event commands, polling block code, ...) is caught and transformed in a normal Roby error. In other words, it is caught at the boundaries of the task and injected into the normal Roby exception handling. We believe that, this way, one reaches the same level of safety than with a system where code and models are separated. I.e. it is robust to "obvious" errors (errors that are detected by validation routines inside the code itself), but neither more or less robust to "silent" errors (errors that a diagnostic component could catch).

3 System deployment and supervision in Rock

While the most widely spread topic about architectures in the robotic community has been focussed on functional layers, the (much more complex) problem of making system networks out of said

[3] a very important aspect of Ruby is that classes are themselves first-class objects of class `Class`

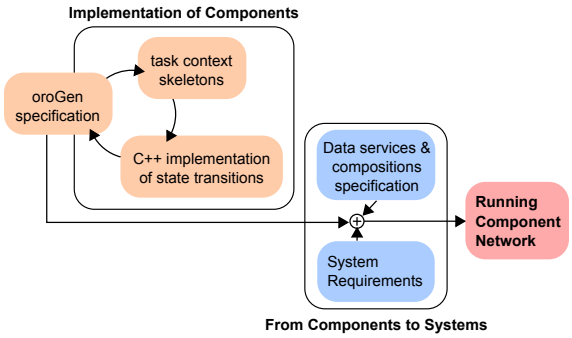


Figure 3: The development process using our toolchain

components have been seldom studied. This is a very important topic, as the easy *composition* of components is critical to their reuse. Tooling is meant to allow the integration of a component in a system with as little knowledge as possible on their intrinsics. Ideally, simply requiring the use of a component in the context of a specific robotic system would be enough to see that component integrated in the data and control flow of the robot's functional layer.

Moreover, system-level definition of a component network is also critical to integrate higher-level layers: these layers cannot deal with the specifics of a functional layer. They instead rely on the availability of high-level services, services that are required and rejected by high-level decision-making components, leading to a dynamic reconfiguration of the underlying component network.

This issue of creating composition of components, i.e. reusable building blocks that are themselves aggregates of components, is a problem that has been studied in the software engineering community. However, one critical aspect in robotic systems is that compositions get reused in different parts of the system (for instance, it is common for multiple components to depend on data produced by a single sensor). This aspect of composition *with sharing* is a lot less common. In the Fractal component model, Bruneton et al. [2] tackles it. While being very related to Fractal, our approach extends it by using some form of dependency injection and aspect-orientation, and also by providing ways to reconfigure the component network online.

Moreover, approaches do exist that allow to *verify* that a given system design is valid. The BIP component system [1] allows to model and compose components at the levels of their behaviours, interactions and priorities and verifies that the resulting composition is valid by generating a model-checked controller. The software engineering community also, obviously, concerns itself with the topic of checking that components are compatible at both the *interface* and *protocol* levels [5, 14]. We believe that the approach presented in the remainder of this paper is complementary to these approaches: we provide a way to *specify* requirements that are translated into a working network of components. Said network could then be checked with the methods listed above.

3.1 System integration in Rock

The Rock development environment borrowed the idea of having a component generation and specification tool from the GenoM tool [3]. This tool, oroGen, uses a specification of the component(s) interface(s) and generates the C++ code of a RTT component that matches this specification. The purpose of this tools is therefore twofold:

- models of the component interfaces is made available to system-building algorithms, and
- guarantee that the actually implemented components do match their specification

Based on this specification, various tools have been built to streamline the process of system integration. The resulting development process is two-stage (Fig. 3).

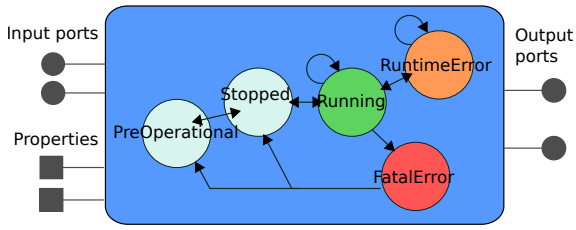


Figure 4: Interface of components in the Orocos RealTime Toolkit, which is the underlying C++ implementation in Rock. Output ports are explicitly connected to other component's input ports to form a data flow network. Properties are used to set and query configuration values. The component model offers an internal state machine representation, thanks to which the monitoring of components is easily doable in the supervision layer. Finally, a remote procedure call interface is also available, but not represented in this figure.

First, the components must be defined and implemented. Importantly, as already stated, thanks to the use of the oroGen tool, *specification* of the component interface is made available to the rest of the system (Fig. 4). This interface includes dataflow inputs and outputs, configuration properties and state machine information (what states are available). This last point is critical to our approach, as it constitutes the standard way through which tasks report their progress to the plan management layer.

Second, the components are bound together to form an actual running system. The remainder of this section will give an overview of this process. The details of the underlying of the algorithms are left out of its scope though.

To illustrate the system-building process, this paper will use the example of an AUV, named Avalon, that is being developped at the DFKI Robotics Innovation Center and participated in the 2009 and 2010 Sauc-E competitions[4]. The Rock toolchain is used on this AUV to deploy and monitor network of components. In particular, the system's control loop, pose estimation and a pipeline following behaviour (in which the system uses cameras, image processing and a visual servoing controller to detect and follow an underwater pipeline) are going to be used to illustrate our methodology.

3.2 From components to systems

The main purpose of the system-building process is to aggregate the individual components to form functional networks. The idea of the system-building tool that is presented here is to design models so that this aggregation is both *robust* and *flexible*. Indeed, its most interesting aspect is that it allows to transparently reconfigure the system at runtime.

In short, the general idea of the approach is to adapt modern principles from aspect-oriented programming and dependency injection patterns to the realm of network of components.

The main system-building block in our model is the usage of a composition pattern. Compositions represent subnetworks of components that, taken together, form a functioning sub-system. For instance, the `PoseEstimator` composition in Fig. 5 offers a *functioning* pose estimation service to the complete system.

However, one can see that a composition such as this one ends up being system-specific. From a functional point of view, it is a necessary evil: pose estimation in field robotics typically depends on the type of sensors that are present on the system and is rarely completely generic. However, from a system-building point of view, it is desirable to add some *abstraction* so that *(i)* other part of the systems can specify that they need *a* pose estimation without specifying which one and *(ii)* monitoring can be built incrementally – with generic monitoring routines being implemented at a generic level, and more specific one implemented only for the particular implementations it can

[4] `http://www.sauc-europe.org`

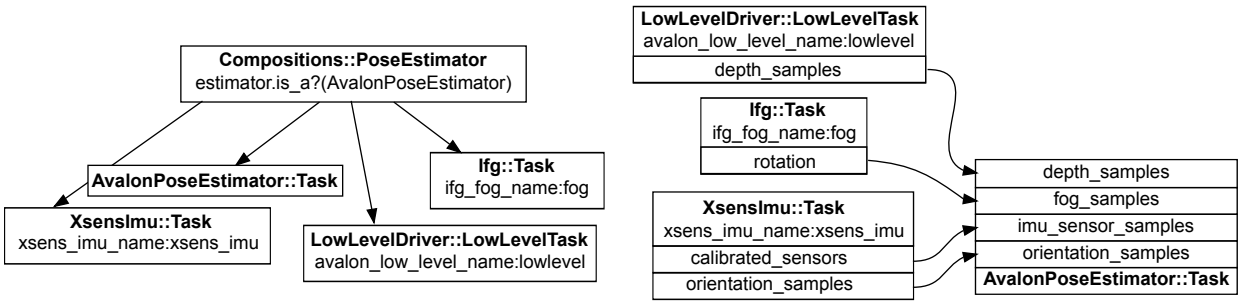
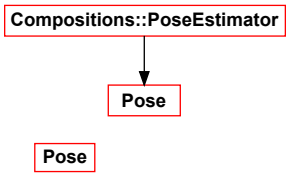


Figure 5: Pose estimator composition on our AUV system. The pose estimator composition provides a self-sufficient pose service, i.e. *by itself* it can provide an estimate of the AUV pose. The left part represents the dependencies that exist between the composition and its constituents: the compositions need its children to run properly for it to function properly. The right part represents the data flow between the component's inputs and outputs.




```
data_service 'Pose' do
    output_port 'pose_samples',
        '/base/samples/RigidBodyState'
end
composition 'PoseEstimator' do
    add Srv::Pose
end
```

Figure 6: Generic pose estimator composition, that can be reused across systems. The pose estimator depicted in Fig. 5 is a specialization of this one, for the purpose of our AUV. These specializations are meant to offer a mean to build an abstract understanding of a component network, for tools and algorithms that do not need to understand the specificities of a given system. Red boxes are used to denote abstract tasks and services. See Fig. 5 for an explanation of the graphs signification. On the right, the corresponding eDSL definition is given.
.

be applied on. In other words, it is desirable to abstract the common parts of a composition in common models, that are then refined into more specific ones (i.e. to build hierarchies of models).

To achieve this generality, two mechanisms are provided. First, components and compositions are mapped to *data providers* and *data services*. Data providers are components that provide a certain type of data (for instance, an IMU driver would be an `Orientation` provider). Data services are data providers that are *self-sufficient*. In other word, a Pose service – if selected – may provide a pose estimation *without the need of interaction with other components* while the corresponding Pose provider may, if instantiated by itself, do nothing.[5] Using software engineering terms, services and providers are abstract interfaces, and can be used as placeholders in compositions to form a type-safe dependency-injection scheme.

The dependency injection is done locally (on a per-composition basis) when adding new requirements for the deployed system. For instance, assuming a (fictional) SuperGPS device that would provide a full pose to the system, one could add the pose estimation deployment into the generated network with

```
add(Cmp::PoseEstimator).
  use(SuperGPS::Task)
```

(where `Cmp::` and `Srv::` are shortcuts for `Compositions::` and `DataServices::`). This injection is done *locally*, as it applies only on that particular Cmp::PoseEstimator, not to other Cmp::PoseEstimator compositions that are deployed in other parts of the system.

Second, compositions can be *specialized*. Specializations offer an implicit way to adapt composition models to the actual needs of the services/providers selected as their children. For instance, the generic `PoseEstimator` composition of Fig. 6 only specifies that it requires a Pose provider. However, once an actual Pose provider is selected (Fig. 5), the corresponding specialization is selected (denoted by the `estimator.is_a?(AvalonPoseEstimator::Task)` subtitle). In addition to the estimator child present in the generic `PoseEstimator` definition, this specialized composition adds all the needed sensors that the specific `AvalonPoseEstimator::Task` component needs to function. In software engineering terms, specializations offer to adapt the compositions based on different aspects of the actual components that provide the services. For instance, the specialization declaration of Fig. 7 turns the generic composition of Fig. 6 into the specific one of Fig. 5. A deployed composition is the result of its base definition on which all the specializations that apply have been overlaid.

During the system deployment (red step on Fig. 3), the requirement definition is used to map any data service referred in one of the composition definition to concrete tasks. This mapping is either explicit (listed in the system requirements) or implicit if the system offers only one concrete task that provides a given service. For instance, the specialization of pose estimator in Fig. 5 actually uses data providers to represent its needs from IMUs (`Orientation` and `IMUSensors` providers) and depth sensor (`Depth` provider). The reason for that is that it happens that new sensors are tried out on our system and that this composition is then reusable for other AUVs that do not have the same sensor suite. However, in configurations where the system has only one instance of each sensor, the deployment engine will automatically pick those leading to the fully concrete composition instantiation depicted in Fig. 5.

3.3 System deployment of an AUV

The part of our AUV's deployment that has the most value as an example are the integration of the detectors and servoing behaviours.

At the root of the deployment is an `AUVControlLoop` composition model (Fig. 8). This model gathers the three basic blocks needed for autonomous control of the vehicle:

- a `MotionController` service which represents the closed-loop control that gets absolute depth, heading and forward and transversal speeds as input commands,

[5] obviously, data services are data providers

```
specialize Cmp::PoseEstimator,
    Srv::Pose => AvalonPoseEstimator::Task do
    add Srv::DepthSensor
    add Ifg::Task
    add Srv::Orientation
    add Srv::IMUSensors
    autoconnect
end
```

eDSL declaration of a specialization. The `specialize` statement takes two arguments. The first one is the composition on which the specialization should be applied (`Cmp::PoseEstimator`). The second is a mapping from something that identifies a child in the composition (here the `Srv::Pose` service) to a selection of tasks and/or services that will trigger the specialization. Finally, the `do ... end` block is the specialization itself: it lists declarations that should be added in cases where the specialization applies. In effect, the statement can be read as: "specialize `Cmp::PoseEstimator` in this matter if a `AvalonPoseEstimator::Task` is selected in place of its child that provides the `Srv::Pose` service."

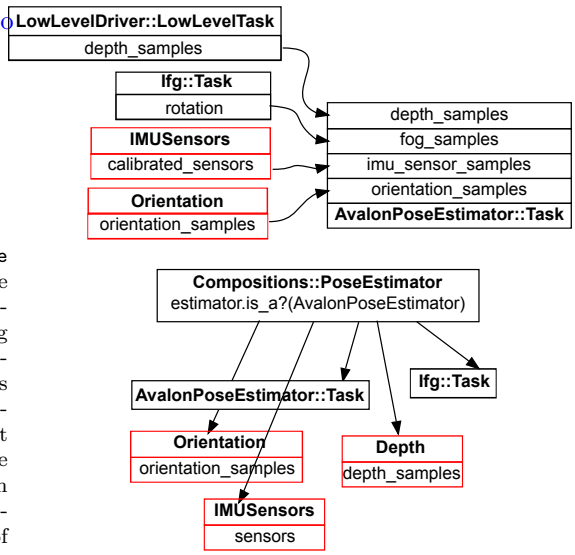


Figure 7: Declaration that turns the generic composition of Fig. 6 into the one that can be used to deploy the vehicle-specific pose estimation component (`AvalonPoseEstimator`). The composition of Fig. 5 is then obtained thanks to the deployment engine, which automatically selects the tasks as they are the only providers of the required services. The `autoconnect` statement computes connections *withing the scope of the composition*.

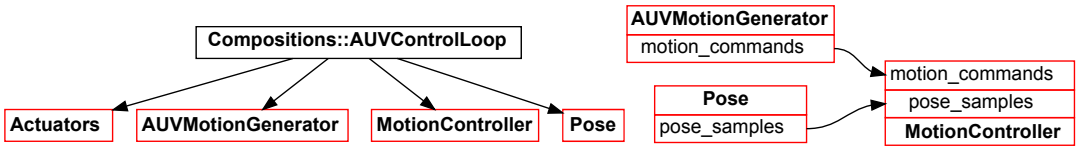


Figure 8: Abstract `AUVControlLoop` model. The red boxes are the abstract tasks (either abstract compositions or data services).

- a `Pose` service that provides the depth and heading required by the `MotionController`
- a `MotionGenerator` service which generates the commands for the closed-loop control formed by the Pose and MotionController services.

This model is completely generic, and can be reused on different AUV systems.

On our system, the `MotionController` service is provided by a single task. It can therefore be selected implicitly by the deployment engine (Fig. 9). Additionally, since the high-level motion control and the low-level motor control are separated, a specialization of `ControlLoop` has been created, that adds the thruster control task (`MotconController::MotconControllerTask`).

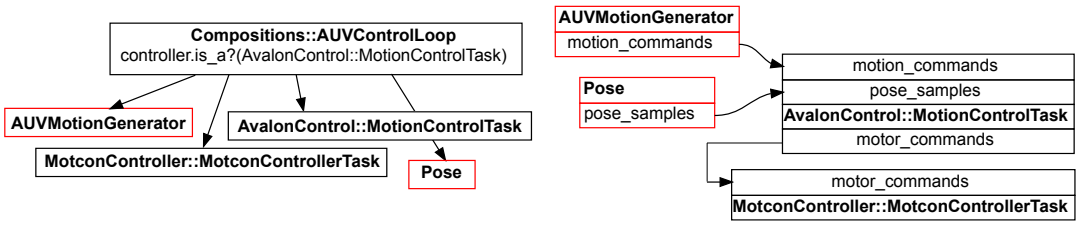


Figure 9: Base `AUVControlLoop` for our AUV. Since only one task provides the MotionController service in our AUV, the deployment engine has automatically selected it (`AvalonControl::MotionControlTask`). This led to the usage of the corresponding specialization, in which the system's motor controllers are also added (`MotconController::MotconControllerTask`) to form a functional block.

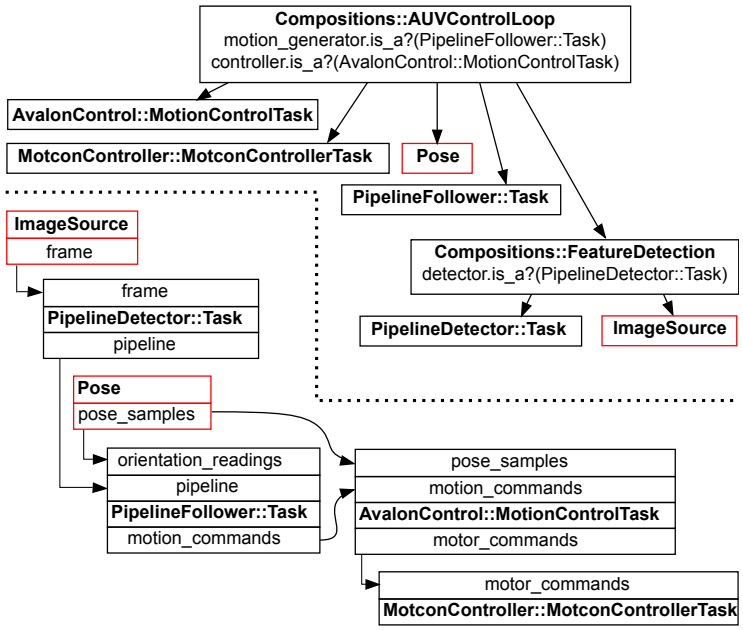


Figure 10: Component network instantiated for the pipeline following behaviour. The top part represents the dependencies: i.e. what composition depends on what task to run. The bottom part represents the computed data flow, where port names are inputs when above the task name (in bold) and outputs are below it. The corresponding requirement specification is detailed in the text.

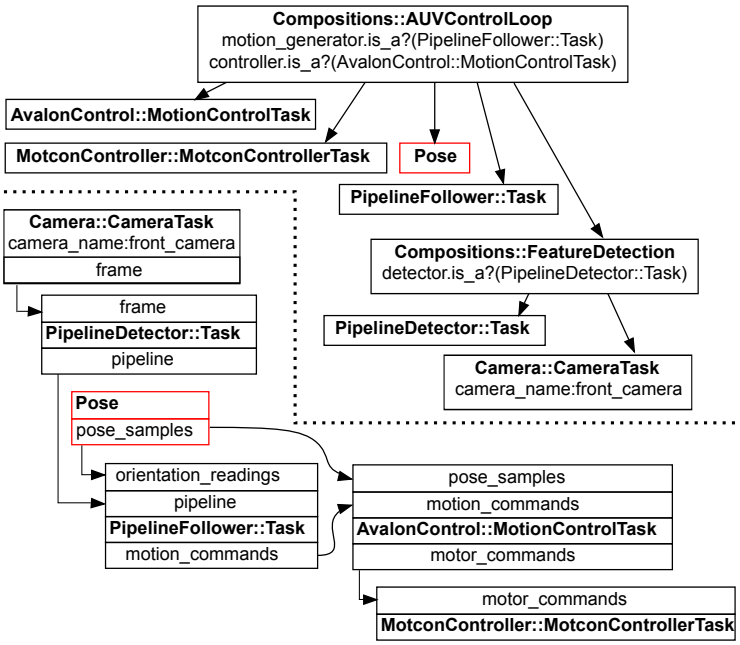


Figure 11: Complete component network instantiated for the pipeline following behaviour, including the selection of the camera. Indeed, as our AUV features two cameras, one must be selected explicitly. The top part represents the dependencies: i.e. what composition depends on what task to run. The bottom part represents the computed data flow, where port names are inputs when above the task name (in bold) and outputs are below it. The corresponding requirement specification is detailed in the text.

What is left to select are the `Pose` and `MotionGenerator` implementations. The first one is defined as a globally injected dependency for *all* Pose data services with

```
add Srv::Pose => Cmp::PoseEstimation.
    use('xsens_imu')
```

The second one is the one of interest for us, as it is the one where servoing components will be placed.

Let's assume that the pipeline following behaviour has to be activated. This would be translated as a system requirement into

```
add(AUVControlLoop).
  use(PipelineFollower::Task)
```

This leads to the network depicted on Fig. 10. However, one can see that the `ImageSource` service is not selected. This is because our system has two cameras, and one therefore needs to be selected. This is done with

```
add(AUVControlLoop).
  use(PipelineFollower::Task, 'front_camera')
```

Where "front_camera" is a subnetwork defined separately. This leads to the final pipeline following network depicted on Fig. 11.

It is important to notice that nothing forbids, in that network, the detector and the motion generators to be provided by the same task. If it is the case, the connections between that tasks and the rest of the system will reflect its current role: if it is used as a `MotionGenerator`, its motion command output will be connected to the motion generator. Otherwise, this output will *not* be connected, allowing to use the component as a detector only.

While the actual deployment algorithm is left out of the scope of this paper, one important property is that it is able to detect and eliminate redundant requirements. In other words, one can inject dependencies on a per-composition basis, without having to design or care about the rest of the system. The deployment engine will then compute the minimal network that performs the required function (if possible) or generate an error

Another important property is that it is able to *adapt* a running network to changing requirements at runtime, i.e. switch dynamically between different sets of requirements.

Some of the most important traits of the Orocos/RTT that we leverage is that connections must be *explicitly* created between the components. This is important for two reasons: *(i)* as we just showed, a single task can provide multiple services. What service is actually used in the system is enforced by connecting the relevant ports and leaving the other services unconnected. For instance, a component that is both an object detector and a visual servoing algorithm can be used as an object detector only by leaving its command output unconnected. *(ii)* when adapting the system network, the well-being of the complete system is less dependent of the good behaviour of a single task, as there is always the option to completely disconnect that task from the rest of the network to isolate it, even in case that it is impossible to stop it.

3.4 Plan Management

The key component that manages the functional layer is a *plan manager*. The basic idea behind a plan manager is to have a management of plans, i.e. to *not* use planning at the center of the architecture, but instead consider that planning is one more component in the system that helps with decision-making. However, a plan manager still uses a *plan* as its central data structure: it is still able to manipulate a data structure that represents the evolution of the system along time. It is just that it neither requires nor forbids its plan to be generated by a planning tool.

The plan manager we are currently using, Roby, has already been presented in great details in [7, 8] and will therefore not be detailed in this paper. In the architecture presented in this paper, it has two jobs:

- mix prepared plans (plan libraries) and online decision-making based on reasoning on the available information.
- represent the links between what happens in the lower layers and high-level information about the mission progress. For instance, the pipeline and leak detectors emit events that are propagated in the higher layers to represent the progression of the mission.
- supervision, i.e. execution monitoring. Thanks to the dependency structure that our plan manager supports, the plan manager can detect what are the non-nominal situations in the lower layers, and what impact does these situations have on the higher layers in the plan.

To achieve the integration of the network of components within the plan management system, we leverage the flexibility offered by the usage of Ruby-based eDSLs in all the stages (specification, plan manager interface, ...). What it means in practice is that a Roby plugin generates on-the-fly new models (i.e. Ruby classes that are subclasses of the `Roby::Task` base class) to represent the oroGen tasks. This mapping is very straightforward to create as it is simply using the intrinsic reflexivity of the method: the Roby plugin and the task themselves are reflexive, meaning that they have access to the task models as well as means to modify themselves to match these models. The other elements of the system deployment specification are also modelled in the same way. As stated previously, this allows to extend the available API to factor out design patterns. For instance, the visual servoing case presented above is a pretty common case on our AUV (there is four existing detector/servoing pairs, including some in which the detector and servoing tasks are the same). To avoid duplicating the specification work, a `visual_servoing` method has been added to the class that represents the `AUVControlLoop` model. Thanks to that, declaring a new visual servoing behaviour in our system is reduced to:

Cmp::ControlLoop.visual_servoing PipelineFollower::Task, PipelineDetector::Task

Finally, the deployment process itself is represented in Roby plans. This is done by injecting so-called *modality tasks* into the high level plans, and have the plan manager replace these tasks, when it is relevant – when applicable – by the corresponding deployed system task.

For instance, in Fig. 12 represents the transition from manual driving to the pipeline inspection mission described in this paper. The high level part of the component network needed for remote operation of our AUV is depicted as a dependency graph (top). The bottom part represents the prepared plan for pipeline inspection. This prepared plan specifies that the pipeline following behaviour and the feature detection (in this case: the leak detection) are needed by the mission. The system deployment engine then generates the network required and adapts the running system to the new requirements (Fig. 13). It reuses running components as much as possible, and spawns new components if needed.

3.5 Reusability: Design Guidelines for Component Implementation

One important aspect, when using a system design methodology such as the one proposed in this paper, is the issue of *reusability*. How much of the components can be reused? How much of the plan management code can be?

The main focus of the system deployment component presented earlier is to offer a simple interface to build complex component networks, and adapt them at runtime. This being a given, one can try to keep components as atomic as possible, thus supporting the goal of creating reliable systems. Still, in some cases, it is better to keep some conceptually-separated behaviours and/or detectors together for reasons of ease of implementation. This is also supported by our architecture, as the need for *explicit* connections allows to use only the detector part of a joint detector/servoing component by omitting the connection between the servoing part and the rest of the system.

As already mentioned, the plan management side also offers mechanisms to keep generic things generic, and specialize for specific systems only when needed. So, in principle, the plan management models and monitoring code can be kept as generic as possible. However, we still have little experience with porting plan management models and code from systems to systems, so this claim is hard to evaluate in practice.

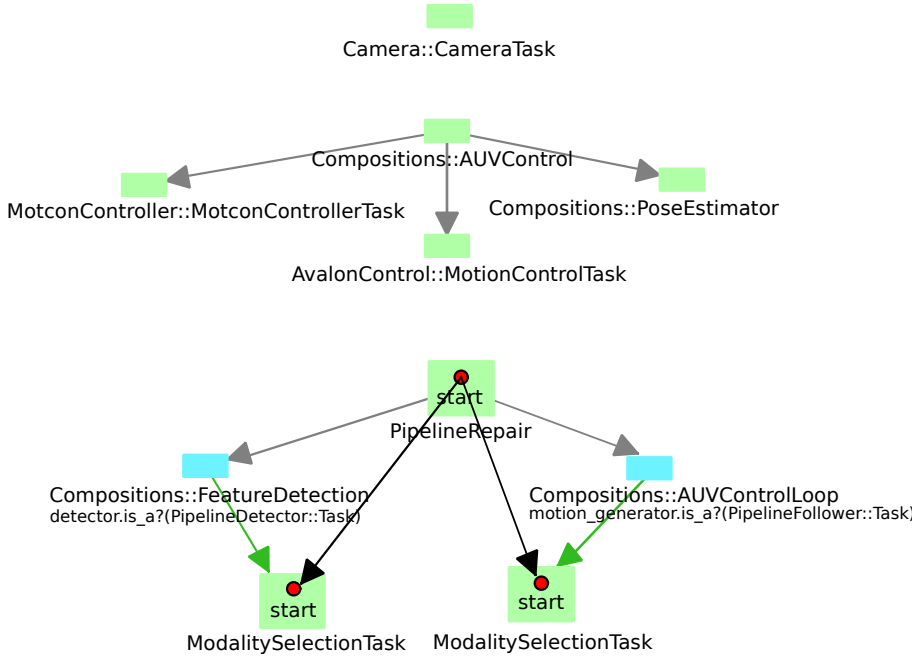


Figure 12: Example of modality selection in plans. This represents the first stage of the transition from a system set up for remote operation to one that is doing the pipeline inspection mission. The `ModalitySelection` tasks represent the needs of the `PipelineRepair` mission in abstract terms (`FeatureDetection` for the leak – which is provided by the same detector than the pipeline in our case – and `AUVControl` for the pipeline following) which are then taken by the system deployment engine to adapt the component network (Fig. 13). Solid arrows between the "start" events represent signals: the plan represents that the modality adaptation has to start as soon as the `PipelineRepair` mission starts. This is live data from an experimental run.

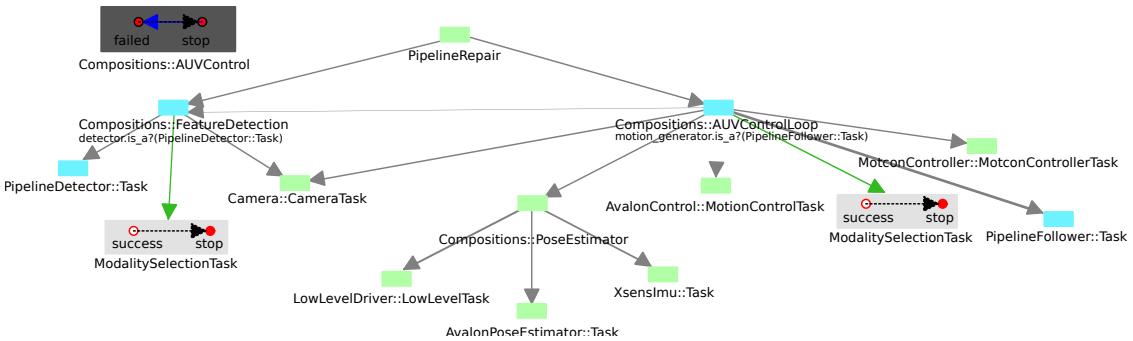


Figure 13: Deployed pipeline inspection mission, as adapted from the remote operation network of Fig. 12. One can see that all but the `Compositions::AUVControl` task on the top-left are reused from the manual driving configuration, the old `AUVControl` task being interrupted by the plan manager. Dotted lines between events (for instance on the top-left task) represent event generalization: the "success" event is a special case of the task's "stop" event. This is a snapshot of live data from an experimental run.

4 Conclusion

We have discussed in this paper the issue of robotic development, from the point of view of robustly creating systems of components. We have used the example of the Robot Construction Kit's (Rock) own toolchain as a way to go in the right direction.

The tooling and methodology presented in this paper has been applied to 5 systems in four different projects: two very different underwater vehicles and three very different ground systems (the Asguard lightweight rover, a more heavy rover system and a legged system). Unifying the models that all these systems are using is still to be done, but until now the reusability of some specific parts of the system modelling has shown great promises.

The main challenge remains: how to, in development toolchains, integrate formal methods with more classical approaches. In other words: how to make model-based reasoning techniques *practical to use*: reusable, flexible, and integrated with non-formal methods – that will remain needed to handle at least part of our robotic systems in the mid-term.

References

[1] A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods*, 2006.

[2] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of the 7th ECOOP International Workshop on Component-Oriented Programming (WCOP02)*, pages 1–8. Citeseer, 2002.

[3] S. Fleury, M. Herrb, and R. Chatila. Genom: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 842–848, 1997.

[4] E. Gat. ESL: a language for supporting robust plan execution in embedded autonomous agents. In *Proceedings of the IEEE Aerospace Conference*, 1997.

[5] D. Giannakopoulou, G.T. Leavens, and M. Sitaraman. SAVCBS 2001 Proceedings. In *Specification and Verification of Component-Based Systems - Workshop at OOPSLA 2001*. Citeseer, 2001.

[6] F. Ingrand, R. Chatila, R. Alami, and F. Robert. PRS: A high level supervision and control language for autonomous mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, 1996.

[7] S. Joyeux, R. Alami, S. Lacroix, and Roland Philippsen. A Plan Manager for Multi-robot Systems. *The International Journal of Robotics Research*, 2008.

[8] Sylvain Joyeux, Frank Kirchner, and Simon Lacroix. Managing plans: Integrating deliberation and reactive execution schemes. *Robotics and Autonomous Systems*, June 2010.

[9] Michel Lemaitre and Gerard Verfaillie. Towards the formal verication of the functional architecture of autonomous satellite onboard flight software. In *Proceedings of CAR'2008*, 2008.

[10] Conor McGann, Frederic Py, Kanna Rajan, Hans Thomas, Richard Henthorn, and Rob McEwen. A deliberative architecture for AUV control. In *2008 IEEE International Conference on Robotics and Automation*, pages 1049–1054, Pasadena, May 2008. Ieee.

[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. In *Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, number Figure 1, 2009.

[12] Kay-Ullrich Scholl, Jan Albiez, and Bernd Gassmann. Mca - an expandable modular controller architecture. In *In Proccedings of the 3rd Real-Time Linux Workshop*, 2001.

[13] Peter Soetens. *A Software Framework for Real-Time and Distributed Robot and Machine Control*. Phd, KU Leuven, 2006.

[14] A. Speck, E. Pulvermuller, M. Jerger, and B. Franczyk. Component composition validation. *International Journal of Applied Mathematics and Computer Science*, 12(4):581590, 2002.