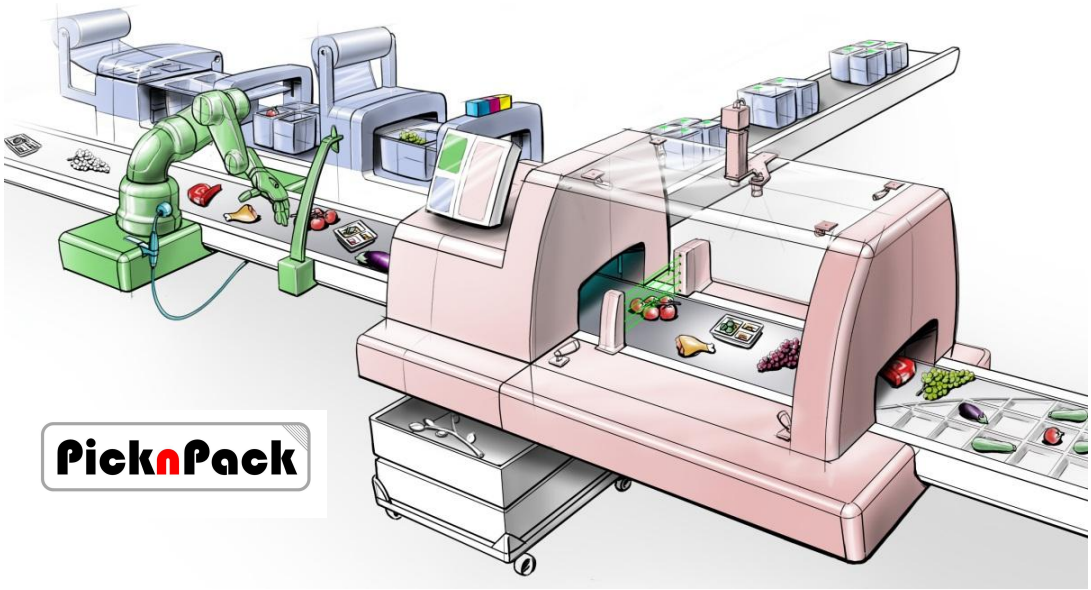
D2.1 — Report on Task-Skill-Motion models

Herman Bruyninckx

25/10/2014



Flexible robotic systems for automated adaptive packaging of fresh and processed food products

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n^o 311987.

Dissemination level		
PU	Public	**X**
PR	Restricted to other programme participants (including the EC Services)	
RE	Restricted to a group specified by the consortium (including the EC Services)	
CO	Confidential, only for members of the consortium (including the EC Services)	

Table of Contents

1 Introduction & Overview . . . 3

2 The "Task-Skill-Motion" approach . . . 4

3 The System Composition Pattern . . . 7
- 3.1 System Composition Pattern . . . 7

4 Guidelines for Line design . . . 10
- 4.1 Role of "container" context . . . 10
- 4.2 Guidelines for "Life-Cycle State Machine" model . . . 11
- 4.3 Guidelines for domain data semantics . . . 11
- 4.4 HDF5 as industry-grade data model . . . 12
- 4.5 Guidelines for food processing "world model" . . . 12

5 Guidelines for Module design . . . 12
- 5.1 Geometrical data . . . 13
- 5.2 Communication . . . 13
- 5.3 Software and data bridges . . . 13

6 Guidelines for Machine design . . . 14

7 Suggestions for traceable Globally Unique Identifiers . . . 14

8 Flexible GUI design . . . 15

9 Milestone MS1 . . . 15

10 Conclusions . . . 16

1 Introduction & Overview

This Deliverable resumes two years of activity in the Work Package on *"Flexible systems integration"*, more in particular in the ambitions described in the *Description of Work*:

- *to make the project's "flexibility" promise a reality, via a structured methodology that is (i) simple enough for all project partners to realise the ambitious integration and flexibility goals in the project, and (ii) generic and powerful enough to be reusable as a general methodology also beyond the scope of this project, in similar large-scale integration projects.*

  This objective is realised via the *System Composition Pattern*, Sec. 3.

- *to decrease the dependency of system builders on specific vendors, middleware, hardware and software platforms, etc., without compromising on their integration efficiency.*

  This objective is realised via *Software and Data bridges*, Sec. 5.3.

- *development of compositionality guidelines: best practices, standards, and templates for system integrators to build predictable, traceable and flexible systems in a methodological way.*

  This objective is realised via the *Guidelines* in Sec. 4.

- *development of composability guidelines: best practices, standards, and templates to help developers of components in making their modules much easier to integrate in larger systems, imposing a minimum of constraints on the integrators.*

  This objective is realised via the *Guidelines* in Sections 5–6.

"Flexibility" has many interpretations, and almost three years have passed since the writing of the paragraphs above, so, in this project's context of food processing *systems*, the research in Work Package 2 has resulted in more specific descriptions and solutions. That is, we aim for food processing automation that can be more flexible than the existing ones in any of the following meanings of that word:

- *separation of task and platform*: every piece of automation infrastructure can be discussed from a "user perspective" (*what* is the machine providing as useful functionality?) and from a "developer perspective" (*how* is the functionality implemented?). In order to maximize the *exchangeability* of automation equipment, their designers must keep both aspects as separated as possible, while still allowing their full integration.

  This objective resulted in the "Task-Skill-Motion" design methodology presented in Sec. 2.

- *auto-configuration of modules*: when a new module is brought into a food processing line, for the first time or after it has been swapped out for cleaning or updating, the act of plugging together the physical connectors (power and data) should start off a process of auto-configuration, in which the software of the line and module controllers interact with each others as much as possible without human intervention.

  The core contribution reported upon in this Deliverable is the *System Composition Pattern* of Sec. 3, which helps machine, module and line developers with the *structure* ("architecture") of their design challenges, and "configuration" gets a very prominent place in that context.

- *traceability*: the control software of a Pick-n-Pack food processing line must be able to answer queries from internal or external users to find back the data about every set of batches of food that has gone into a line, about every set of batches that was produced by the line, and about the processing

that each batch has undergone. This challenge is not just a matter of developing appropriate *data base software*, but more about suggestion *data models* that could, *eventually*, be introduced into the community as standards for food processing traceability.

The contributions of Work Package 2 in this context are two-fold:

  - *integration with Work Package 3*, Sec. 7.
  - *HDF5 as industrial-grade data model*, Sec. 4.4.

- *data semantics*: the above-mentioned *data models* should be more than the traditional standards, in that also the *meaning* of the data is captured in so-called "ontologies" or "semantic models". For example, "cherry tomato" is not just a string representing the name of a type of vegetable, but it is a *semantic tag* for which *formal relationships* are being defined to many other aspects of the Pick-n-Pack food processing activities: sensing, quality assessment, robotic manipulation guidelines, packaging guidelines, etc.

  Since the beginning of the project, DLO and KUL have been working on various iterations of a "vine tomato ontology", which eventually will represent all of the above-mentioned knowledge aspects. (We refer to Deliverable D2.4 for more information on this work.)

- *location adaptability*: one particular food processing module should be integratable in many different processing lines, and in different locations in that line, but still its control software must be able to adapt its working to the actual geometric location of the module in the line.

  This implies that a Pick-n-Pack module's software is designed to work with an explicit *"world model"* of the line's geometric layout and its own position and role in that line. Section 4.5 has the concrete *Guidelines* for such a "world model'.

- *runtime adaptability*: when taking a module out of a line (say, for hygienic cleaning), or when putting it back into a line, one should not have to stop the control software of the line and/or any of the other modules. However, traditional machine or modules do not have this ability of *"hot swapping"*[1].

  The Pick-n-Pack designs contain a *Life Cycle State Machine* functionality to do realise such flexible behaviour, Sec. 4.2.

2 The "Task-Skill-Motion" approach

The design concept of "Task-Skill-Motion" was, historically speaking, the input from partner KUL in the Pick-n-Project, from earlier research in sensor-based control. The "lessons learned" before Pick-n-Pack was that all earlier approaches to the specification, programming and control of complex "robot" tasks were insufficiently flexible, for many reasons, the most important of them being:

- the too high presence of (robot or sensing) *platform-specific parameters* in the specification and the software. For example:

  - the maximum speeds or forces that the platform can tolerate;
  - the maximum accuracy that the platform can achieve;
  - the reactions to platform "error conditions", like singularities, loss of tracking in sensing, computational errors, communication latencies, actuator overheating, etc.

[1] http://en.wikipedia.org/wiki/Hot_swapping

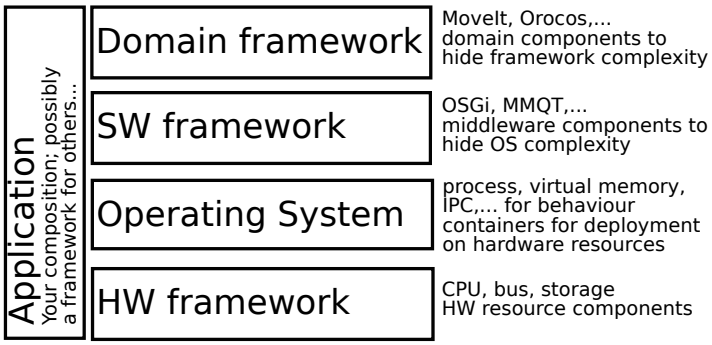


Figure 1: The various levels in the computational platforms. Each Pick-n-Pack partner is making concrete choices for all of them that, implicitly and unintentionally, introduce strong constraints on, both, the *integration abilities* and the *performance* of that partner's module or machine in the overal Pick-n-Pack line. For example, the choice of data structures, or communication middleware, or operating system, or programming language.

- the lack of *structural data models* to represent the various kinds of "platforms" that developers have to use in any realistic automation project, such as:
  - *mechanical structure*, Fig. 3: every device has power drivers, actuators, mechanical joints, kinematic joints and chains, each with different properties, limits, functionalities.
  - *mechanical functionality*, Fig. 2: different mechatronic devices have different "motion/action" capabilties.
  - *computational platforms*, Fig. 1: the necessary software can not be run on an "ideal" platform, but is confronted with deployment constraints at various levels.

The "flexibility" objectives of the Pick-n-Pack project can not be achieved without a *major* step change in the state of the art, in this context of **separating the "magic numbers" of the specific hardware and software platforms** offered by various providers, **from the functionality** that they provide. Helping all project partners to achieve such step changes is the key responsibility of KUL in the Pick-n-Pack project; this ambition is reached in a gradual way, and will not be achievable completely by the end of the project. However, the progress in the intended direction is reported in the other Sections of this Deliverable. As much as possible, the generic insights and design approaches are translated into "guidelines" that are specific for the Pick-n-Pack contextl more in particular, for *food processing* in *lines*, consisting of several *modules*, each one again consisting of possibly several *machines*.

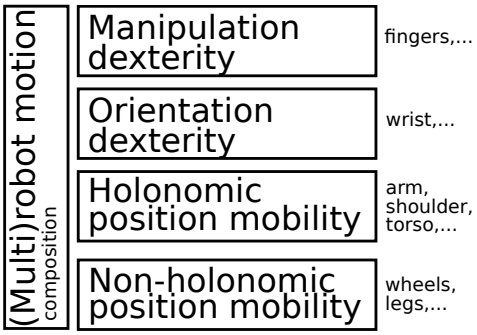


Figure 2: The various complementary functionalities of mechatronic devices. In the context of Pick-n-Pack, the "non-holonomic position mobility" (which is typically provided by so-called Automated Guided Vehicles) is beyond the scope of the research and/or the machinery contributed by all partners.

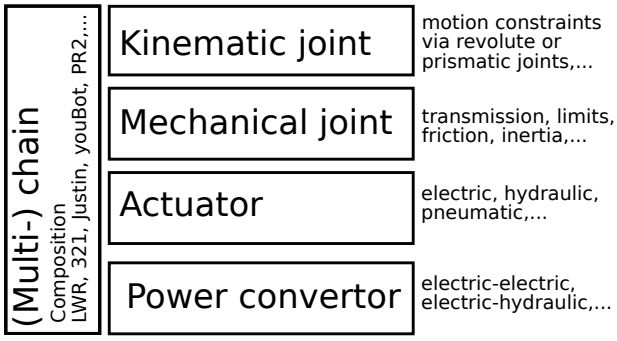


Figure 3: The various complementary structural components of mechatronic devices. This part of the machinery in a system is one of the largest sources of integration problems, since most of the time only the *functionalities* of the "kinematic chain" are of interest to the food processing developments of the partners, while many of the pragmatic integration constraints come from the specific but implicit limits on all of the physical levels below that functionality.

3 The System Composition Pattern

A major achievement in Year 2 of the project was the consolidation of the "grand unification" approach towards building *systems* of the *complexity* of a Pick-n-Pack food processing line, while still being able to identify how exactly each of the *flexibilities* of Sec. 1 can be realised. Figure 4 depicts a representative example of the so-called *System Composition Pattern*.

The following two references (which are added as appendices to this Deliverable) provide a lot more details about the System Composition Pattern, in two complementary ways: reference [1][2] focuses on *how to use* the pattern for developers of functionalities, while reference [2] is meant to guide *software developers* who have to support production systems.

The following sub-sections provide a more "birds' eye view" on the technical detail of the mentioned references, focusing on the concrete needs and circumstances of the Pick-n-Pack partners and Work Packages.

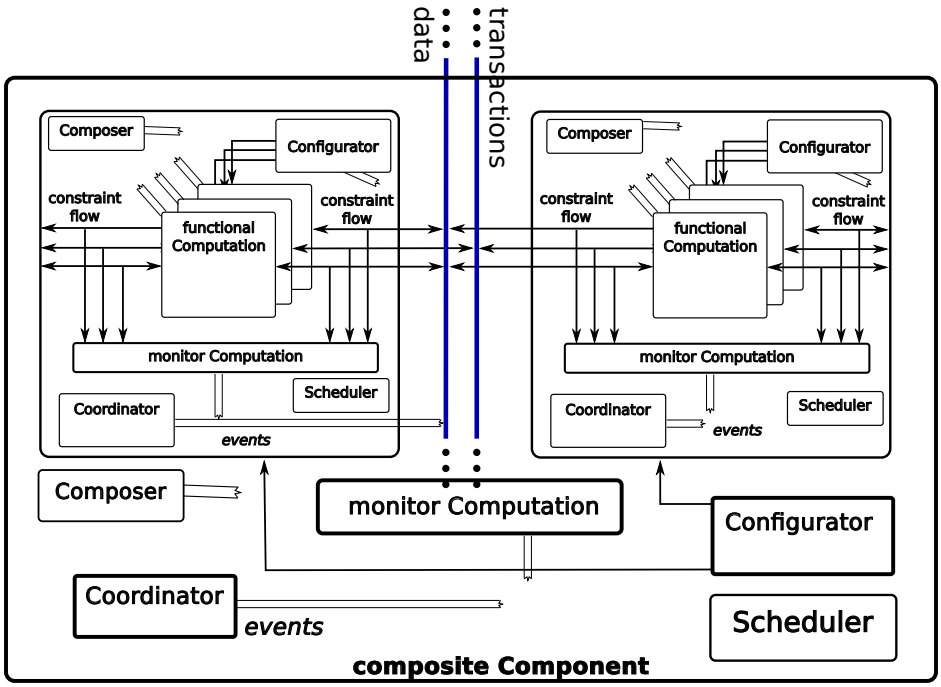


Figure 4: A conceptual overview of the *System Composition Pattern*.

3.1 System Composition Pattern

As mentioned above, Figure 4 is a *sketch* of a typical complex system design. Below we give an overview of the major properties of this Pattern; KUL helps individual Pick-n-Pack partners to translate these properties into the concrete (sub)system designs and implementations that they are confronted with. In Year 2, DLO was the major collaborator in this respect, and the focus of this intensive collaboration was to transform the visual tomato recognition functionalities developed in DLO into a something that is 100% conforming to the System Composition Pattern, such that this example can be used by other partners as a concrete *reference implementation*.

So, here is the list of major insights into, and properties of, the System Composition Pattern:

[2]This paper is currently under review; preliminary work in this domain has been reported in [3].

- *loose hierarchy*: although the Figure seems to suggest a strict hierarchy, it is not: each "level", represented by a rounded rectangle, represents a *context* in which the components inside have access to a well-defined number of *data and knowledge models* from which to query for the "magic numbers" they need. Data and event flows can cross several contexts. Each component can be contained in several contexts at the same time.

  For example, DLO's visual inspection component gets its "magic numbers" from the "tomato ontology" (which also spans the other components that deal with the same tomatoes) but also from the "robot platform" context of the device on which the cameras are mounted.

- *functional Computations*: these are the algorithms that provide the "behaviour" needed by the system in order to realise its added value. *All* the other parts in the composite Component are "overhead", that is needed to help realise these functionalities, in a structured way, with clear indications of roles and responsibilities.

  For example, DLO's visual inspection component is an example of a functional Computation.

- *Coordinator*: this is the *singleton* in the composite Component that is making decisions about when and why the system's behaviour has to change. It *supervises* the *behaviour* of how the composite use its shared resource(s).

  For example, the *line* module will decide that all the *modules* in the line will be asked to reconfigure themselves whenever a new batch of vegetables is being introduced onto the line.

- *Configurator*: this is a component that is responsible for bringing a *set* of functional Computational components into a configuration that makes them—*together* and in a synchronised way—realise a different system behaviour. There can be multiple Configurators, each dealing with a disjoint subset of functional Computational components. Configurators are triggered by the Coordinator, and they shield the latter from having to know *how* the behavioural change that is expects is realised exactly by the available Computational components.

  For example, each *module* control software has configurators to switch the behaviour of its *machines* in a coordinated way: the visual sensing component, the gripper component, the robot motion component, the data tracing component.

- *Composer*: this is the (*singleton*) software activity that creates a new *structural* model of the composite Component, that is, it introduces new data exchanges and functional components, couples existing or new functional components to existing or new data flows, etc.

  For example, the *line* controller must keep the information about which modules it contains at even given instant in time, and how they are interconnected.

- *Scheduler*: this is the (*singleton*) activity that *triggers* the functional Computational components when they are needed to realise the system's overall behaviour. It *computes* the *access* to the *shared resource(s)* of the composite.

  For example, when one would deploy two robot arms in the same module, (partially) sharing each other's workspace, there must be a software component that computes when and where each robot arm can use the shared workspace areas. Such a workspace sharing has been identified as an important future requirement for food processing systems, since the overall footprint of a food processing line is currently a major cost factor.

- *Monitor*: these are the software activities that play a *dual* role to that of the Configurators: the latter's role is to take action such that the system's behaviour can be *expected* to be what is required, while the former's role is to check whether the *actual* behaviour corresponds to the expected one. As with Configurators, one single Monitor can be responsible for a *set* of functional Computational components. Whenever the discrepancy between expected and actual behaviour has become "too large" (which should be a *configurable* threshold!), the Monitor fires an event, to which all other components can react.

  For example, each *module* must monitor the *Quality of Service* with which the *machines* inside the module are performing their expected functionality; if that performance quality drops below a threshold (*configured* by the *line* controller!), an event is raised, to which, first, the module Coordinator can react to take corrective actions. If it does not find a solution, the event should be forwarded to the line Coordinator.

- *data* exchange: while (the very popular) "publish-subscribe" is only one particular, uni-directional, way of exchanging information between components, one should in general allow *bi-directional* alternatives, with other exchange policies than just publish-subscribe; for example, data busses, shared memory, ring buffers, lock-free buffers, etc.

  For example, the cooperation between *modules* in a *line* works best if they keep each other informed about the *Quality of Service* with which they perform their functionality, so that they can adapt to each other's performance without stopping the line for "error recovery" or performance reconfiguration.

- *transactions*: some data has to be archived to so-called *persistent storage*, from time to time, in order to allow the later inspection of the system's activities.

  Obviously, the *traceability* of the food quality data is an essential aspect of the Pick-n-Pack ambition.

- the *whole* pattern is *the* software interface, not the API of the data connections, or of the methods in the functional blocks.

  For most partners, this has turned out to be causing most re-training of their software development attitude, because the requirements to let *concurrently active (sub)systems* interact are quite different from building one single functional component with *object-oriented* design guidelines.

  The experience of Year 2 shows that this aspect of the whole integration is one of the key "project risks" to start monitoring more closely in the future.

- for each *level of Composition*, there is *only one Coordinator*, *only one Composer* and *only one Scheduler*, with possibly multiple Computational, Configuration, Communication components.

- these "singletons" are *the only places* where *specific knowledge* of the other components can be used, via "*query/pull*" or "*configuration/push*" from the component's *context*.

- the *pattern applies fractally*: *every* sub-component can, in itself, be an instance of the pattern again. One specific version of this fractality shows up in the *separation* of the following:

  - *functionality packaging & deployment* (e.g., Task Specification), and
  - *packaging & deployment on operating system*.

  The latter *adds an OS container context*; for example:

  - *Linux containers* (`http://docker.io`), with

  - *LXC* (http://en.wikipedia.org/wiki/Lxc) for Configuration,
  - *supervisord* (http://supervisord.org/) for Coordination, and
  - *cgroups* (http://en.wikipedia.org/wiki/Cgroups for hierarchical Composition.

Let's illustrate the System Composition Pattern with a familiar example from our daily life, in which centuries of human experience has solved all of the important system composition challenges, in a way that is 100% conforming to our *System Composition Pattern*. The example is that of a *School*:

- *functional Computations*: learning activities of the pupils, teaching activities of the teachers, logistic activities of non-educational staff,...
- *Coordinator*: School Director decides whether and when new staff has to be hired, where they have to fit in, with whom they have to cooperate, and what activity they should deploy.
- *Configurator*: staff person assigns teachers to classes and lecture schedules, and operational staff to concrete logistic roles;...
- *Composer*: realises structural plan when school building is renovated; assigns personnel to offices;...
- *Scheduler*: bell system coupled to the computer that runs the algorithm that computes the schedules of all lectures of each day.
- *Monitor*: written or oral tests and discussions.
- *data* exchange: mail boxes; telecom system; notice boards;...
- *transactions*: intermediate and final reports of pupils; financial records of school administration;...

4 Guidelines for Line design

This Section summarizes some extra concrete design guidelines for *line* developers; in the more particular Pick-n-Pack context, this is WP7's *"Fresh and processed food production line"*.

4.1 Role of "container" context

As was mentioned already before, the *System Composition Pattern* has a loose hierarchy of "context components" in which one or more other components can be "deployed". Design motivations behind this structural property, as relevant at the *line* design level, are now explained in somewhat more detail, :

- a composite component provides a *boundary* ("closing the world" in a "software container") in which *particular knowledge* can be applied, or particular learning can be integrated, *without* having
  - to integrate the knowledge inside *one particular* component within the composite, and
  - to care about taking into account "reasoning rules" that are not relevant to the context.

  For example, the properties of a particular kind of tomatoes are queried from the "tomato ontology server" and used as "magic numbers" in the configuration of the sensing and robot modules.

- when an *explicit* context primitive is lacking in system architecture design, these "closure" and "introspection" aspects of a sub-system always end up somewhere inside one or the other component, that then has *too much knowledge* about the *inner workings of the other components* to be practical for later scaling and/or further integration of this particular sub-system.

  This problem has been identified every time already when, in the first two years of Pick-n-Pack, KUL discussed "legacy" software components, implementations as well as designs. This phenomenon is rather easy to spot, by looking for the "magic numbers" that are not documented unambiguously, and that can not be traced one-to-one to the configuration information that comes from the module or line the component is being used in.

4.2 Guidelines for "Life-Cycle State Machine" model

A Pick-n-Pack food processing line is not a static machine, but will have to go through various forms of reconfiguration, for various reasons: cleaning of tools or machines; replacement of an old module by a new one; replacement of a module by one or more humans, and back; etc. Hence, the system design foresees a *mechanism* so support this dynamic reconfiguration, via the following *States* that the system can be in:

- `create`: all resources needed for the (software) functioning of the module are created.
- `configure`: the module is made ready to realise one particular kind of functionality.
- `start`: the module's functionality is starting up; this can involve some "transient effects" before the nomimal activity performance is reached.
- `run`: the system is providing it nominal activity and performance.
- `freeze`: the system's activity is frozen for some time, but it remains ready to continue with it immmediately.
- `stop`: the system's activity is shut down; this can involve some "transient effects" before the nomimal activity performance is brought to a halt.
- `delete`: all resources taken up by the module are freed.

4.3 Guidelines for domain data semantics

If one *really* wants to reach ultimate flexibility in the composition of a food processing line from modules provided by different vendors, and queryable from the "outside world" by various stakeholders, one *has to* come up with standardized data models for the domain. Very few domains already have done so, for example:

- *the "Web"*: HTML is a very successful (set of) standards, which was at the basis of the explosion of innovation on that platform.
- *meteorology*: weather institutes world wide have agreed on some standards to exchange the data of their weather stations; for example, NETCDF-CF, the *Climate and Forecast Metadata Conventions*.[3]
- *finite element data*: for example via *XMDF* (eXtensible Model Data Format).[4]

KUL has investigated all these formats, and many others, to be prepared when the Pick-n-Pack consortium will be ready with its own domain data model(s). The following subsections report on the two currently most mature suggestions in that direction.

[3] http://en.wikipedia.org/wiki/Climate_and_Forecast_Metadata_Conventions
[4] http://en.wikipedia.org/wiki/XMDF

4.4 HDF5 as industry-grade data model

Hierarchical Data Format[5] (HDF5) is a mature standard to represent the *numerical* parts of complex data structures. That is, all the "numbers" but without the "semantics"; the latter are supported by allowing *meta data* fields to be added to the numerical data.

KUL has already conducted several real-world experiments with the HDF5 software libraries provides by the *HDF Group*[6], to test the integration possibilities with communication middleware (see Sec. 5.2, such as ZeroMQ[7]) and *"big data"* databases to support traceable data storage and querying (such as iRODS[8]). All these tests resulted in positive acceptance, in that they all are capable of solving the respective needs of the Pick-n-Pack project. Pragmatically, all partners use only their custom-made, non-standardized data formats, such that a real integration will still require significant implementation efforts.

4.5 Guidelines for food processing "world model"

As was already mentioned before, the Pick-n-Pack project must store information about "the world" at the following complementary levels of detail:

- *line*: the layout of the whole line; the interconnections between modules; their footprints; and the location of peripheral devices (scales, storage space, conveyor belts,...).
- *module*: similar to the line, but now about the layout internal to a module, with the location of all machines and peripherals.
- *machine*: similar to the module, but now about how all mechanical structures, actuators, sensors, communication and computation components, etc., are connected.
- *food*: each individual or "batched" piece of food has geometric properties that have to be modelled, for sensing, gripping, packaging, weighing, etc.

No final decision has been taken already by the Pick-n-Pack consortium, but KUL has identified *XDMF* (see above) as a potential candidate for all of the above. KUL has already conducted some real-world experiments with XDMF, for its storage and retrieval (via iRODS) as well as its querying (via a networked GUI infrastructure of *Paraview*[9].

5 Guidelines for Module design

This Section summarizes some extra concrete design guidelines for *module* developers; in the more particular Pick-n-Pack context, these are WP4's *"Quality assessment and sensing"*, WP5's *"Robotic product handling"* and WP6's *"Adaptive packaging"* modules.

The major beneficiary of this particular work should be DTI: KUL, DTI and DLO have already drafted the design of the control and integration software of the new *"thermo-former"* module that is being developed under the supervision of DTI. Agreement was reached about the design of all of the difficult integration issues: how to synchronize all modules in the line with the timing constraints imposed by the physical workings of the thermo-former? how to provide all modules with the *"world model"* information that the thermo-former is creating at the beginning of the line?

[5] http://en.wikipedia.org/wiki/Hdf5
[6] http://www.hdfgroup.org/
[7] http://zeromq.org/
[8] http://irods-consortium.org/
[9] http://www.paraview.org/

5.1 Geometrical data

The above-discussed XDMF format is optimized to represent *grids* of all sorts, in a standardized numerical way with semantic meta data. Such grids representations fit perfectly to the thermo-former, since that device creates a "web" of packages formed on the line, from a configurable set of mould elements.

5.2 Communication

The practice in the domain of "robotics" and "data acquisition" devices is to use *publish-subscribe*[10] communication policies to connect two or more modules or components together. While this is arguably one of the simpler commmunication policies to understand and use quickly, it does not cover a list of also relevant data communication use cases:

- *data bus*: subscribe to *channel*, pick out any *topic* you like;
- *lock-free buffers*: these avoid all waiting overheadl
- *circular buffers*;
- *shared memory*;
- *blackboard*: shared memory with "topics";
- ...

Sometimes it is also better to move the *computations* instead of the *data*. (But very few software frameworks or libraries exist to support this; the exceptions are data bases, that support *stored procedures* in one form or another.) It is also important to realise that *uni-directionality* makes *Quality of Service* monitoring and *adaptation* way more difficult than practically necessary. Hence, KUL is helping Pick-n-Pack partners in learning how to *separate*:

- communication mechanism and policy;
- message sensing and data model.

Two practical examples of software frameworks that support such separation very well are *0MQ*[11], or *nanomsg*[12].

5.3 Software and data bridges

It is the *module* level of the total system architecture where developers will be most confronted with "legacy" problems of various kinds and for which it is not practical to re-implement them: machines or components that are written in specific programming languages, have their own specific data structures that are more or less, but not exactly, the same as what was agreed upon at the line level, different operating system "containers" (threads, processes, core distribution policies, FPGAs, GPUs,...), specific "inter-process communication" (IPC) middleware, etc. In order to provide all partners with a unified system design approach for all of these problems, KUL has introduced the concept of *"bridges"* between two or more of such not directly compatible components. The core ideas of such bridges are:

[10] http://en.wikipedia.org/wiki/Publish-subscribe_pattern
[11] http://zguide.zeromq.org
[12] http://nanomsg.org

- *full separation between "data" and "functions"*: the most *conceptually clear* way to let several components interact with each other is by letting them communicate their "data" through "sockets", (although that interaction need not necessarily take place via real message sending infrastructure!). Adding a "socket wrapper" around existing software components is typically always possible, and that socket will only have *data* flowing through it. Hence, this insight will help new developments in the project to profit up front from the clean separation between data and functionality. In practice, this might mean a (perceived!) step backwards from "object oriented" design towards "old plain C"-like software development. But the paragraph below has a (partial) answer to this objection.

- *semantic data models to generate data structures*: when all data structures would be available in standard, programming-language independent, representations (such as, for example, HDF5), one can *generate* the data structures for all programming languages automatically, as well as all *read* and *write* operations. Or, in more modern settings, the *REST*[13] operations.

6 Guidelines for Machine design

In Year 3, KUL will go one level of detail deeper than the "line" and "module" levels of *Guidelines*, and adds some extra guidelines for the developers of individual machines, like robots or sensing modules.

The major beneficiaries of this particular work should be LACQ (integration of gripper onto robot modules) and Tecnalia (development of the new cable-driven robotics module); also the KUL group responsible for the quality sensing module has a need for the design guidelines presented above. Close to a dozen face-to-face workshops have already been held with all partners involved.

7 Suggestions for traceable Globally Unique Identifiers

A Globally Unique Identifier[14] (GUID) is a unique reference number used as an identifier in computer software. The term GUID typically refers to various implementations of the universally unique identifier (UUID) standard[15].

The major use of GUIDs in the Pick-n-Pack context is to be able to put them on all batches of food that are transformed in food processing lines. Most in particular, Work Packages 2 and 3 have to agree on a new suggestion for such a GUID, since the existing "bar code"-like identifiers in the food market are not rich enough for the detailed and much more automated tracing queries that consumers and/or authorities will want to have available in the future. While no final decision has yet been made about the exact form of such a GUID, the following requirements have been identified, and checed for practical realisability:

- a GUID data model like that of the *Universal Serial Bus*[16] (USB) is a minimum, to be able to deduce the *vendor*, *type* and *instance* of a food process. This could be enough for the *"outward-facing"* identification requirements of Work Package 3; however, since there are not yet internationally operating associations that have the authority to assign such "USB-for-food-processing" IDs, our suggestion can not materialise in the short term.

- Pick-n-Pack wants to introduce also *"inward-facing"* identification numbers, in order to be able to trace back *all* information that was used somewhere in the whole food processing operations. Again, it is not

[13] http://en.wikipedia.org/wiki/Representational_state_transfer
[14] http://en.wikipedia.org/wiki/Globally_unique_identifier
[15] http://en.wikipedia.org/wiki/Universally_unique_identifier
[16] http://en.wikipedia.org/wiki/USB

too difficult to suggest a realistic GUID method for this goal, in the form of the *composition* of GUIDs for (i) the line, (ii) the module, (iii) the machine, and (iv) their control software versions. Generally speaking, this is the structure that is borrowed straightforwardly from the *System Composition Pattern* that is being used to make the control software and data models for any food processing line designed in the "Pick-n-Pack way".

8 Flexible GUI design

Although the design of the *Graphical User Interfaces* for line, modules and machines are, strictly speaking, not the subject of this Deliverable but only to be provided in Month 36, the research in Work Package 2 has been conducted with the GUI design in mind all the time. The *System Composition Pattern* fits to this design challenge too, because:

- it fits 100% well to the *Model-View-ViewControl* GUI design pattern[17] that will be applied to the whole project in Year 3.
- because of the strict *5C separation*, the design of the actual line, module and machine "controllers" can trivially accomodate one or more "remote clients" of the data and event flows of each component. Indeed, any GUI, on any computer anywhere in the whole system, can become a "client" of the real control component's information without disturbing that component's behaviour and without having to change its software implementation; only the *Monitoring* and *Coordination* parts gets one or more extra *Communications* to support.

The other good news in this context is that the evolution in the *state of the practice* outside of the Pick-n-Pack project is going fast in the direction that is very much compatible with the *System Composition Pattern* that underlies the Pick-n-Pack system software design; more in particular:

- *Web Components*:[18] all major browsers are moving fast in the direction to support the HTML5 standard natively, which allows to make *"single page applications"* in a 100% standardized way, and with a *"5C"*-compatible design.
- *AngularJS*:[19] this is an example of a *major* software framework for connecting HTML5-based single-page GUI "apps" with the real-world systems that they are representing to the users, and with a special focus on supporting a large variety of *Communication* interconnections, from WebSockets (for "server"-side communication), to WebRTC (for "peer-to-peer" data communication), to local file access.

9 Milestone MS1

The Description of Work (page. 12 of 51) contains the following ambition as project Milestone after 24 months, for this Work Package 2: *"The component and Task-Skill-Motion models for a simple robot-gripper-sensor sub-system are realised"*. The progress towards this Milestone is given in more details in the accompanying Deliverable D2.4 (*"Integration of domain specific knowledge with the component model"*) because the most important progress towards this Milestone pertains to the work reported there. With respect to work reported in this Deliverable, Section 6 is most relevant: very intensive cooperation between DLO and KUL has taken place in two "Task-Skill-Motion" aspects:

[17] http://en.wikipedia.org/wiki/Model_View_ViewModel
[18] http://en.wikipedia.org/wiki/Web_Components
[19] https://angularjs.org/

- *5C separation of concerns*: the visual tomato recognition and tracking functionality driven by Wageningen has been designed for optimal configurability, with respect to, especially, the food-specific "magic numbers" that must be adapted every time that the vision component is expected to recognize and assess a new type of food, whose properties are available from the Pick-n-Pack knowledge base.
- *software development*: the transformation of conceptual designs for the "perfect" component software into real software is, always, a lot more labour-intensive than expected. The common KUL-DLO integration efforts, unfortunately, have turned out to be no exception to this "rule", but then mostly as far as the "semantic querying" is concerned, that is, the functionality that we foresee in each component or module to query a knowledge server to provide the "magic numbers" needed for the food- and line-specific configurations. Of course, given the extremely seminal context of this work, there is no *prior art* at all that KUL and DLO could start from.

10 Conclusions

- Co-development with other projects has been taking place, but Pick-n-Pack is definitely leading in the application of the Task-Skill-Motion approach, more particularly via the *System Composition Pattern*, into concrete software implementations.
- KUL supports partners in their concrete designs, at the three "levels" that are relevant to the Pick-n-Pack project: the *machine*, *module* and *line*.
- these three levels also provide, naturally, three levels of "world modelling" that must be supported: the *grid*, the *scene graph* and the *kinematic chain*.
- KUL provides partners with as many well-supported standards and software frameworks as relevant for the project, and possible within the pragmatic constraints of available resources: HDF5, 0MQ, (Linux) containers, etc.

References

[1] D. Vanthienen, T. De Laet, and H. Bruyninckx. Systematic robot application development: Applying the Composition Pattern to constraint-based programming. *IEEE Rob. Autom. Mag.*, 2014. Submitted.

[2] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx. The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming. *J. Softw. Eng. in Robotics*, 5(1):17–35, 2014.

[3] D. Vanthienen, M. Klotzbücher, T. De Laet, J. De Schutter, and H. Bruyninckx. Rapid application development of constrained-based task modelling and execution using domain specific languages. In *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, pages 1860–1866, Tokyo, Japan, 2013.

Systematic Robot Application Development: Applying the Composition Pattern to Constraint-Based Programming

Dominick Vanthienen, Tinne De Laet, and Herman Bruyninckx

Fig. 1: Scene of the tomato picking running example. The PR2 robot has to pick the tomato from the counter (left), and drop it in the basket on the fridge (right). A person doing the dishes between those locations forms an dynamic obstacle during all phases of the task at hand.

Robotics has seen a growth in demonstrations of complex behavior on platforms with an increasing number of degrees-of-freedom (DOF), types of actuation mechanisms, communication networks, sensors, and processors. Robot competitions among such complex, highly autonomous systems attract a lot of attention from the robotics community and beyond. Well-known examples include the Darpa Robot Challenge [1] and the Robocup [2] competition. Given the scale and complexity, as well as the increasing demand of flexible application reprogramming and portability to different platforms, application development has become an effort shared by teams of developers, each with different levels and fields of expertise. In order to create an application, these developers have to create and compose compatible and interoperable building blocks. This integration process, often including parts of a team's legacy software, commonly jeopardizes the success of a project.

This paper addresses application development challenges through the methodological combination of *structure* and *behavior*. More concretely, it formalizes the *Composition Pattern* design methodology and applies this to application development using *constraint-based programming*. Moreover, this paper shows how to refactor existing applications to more reusable systems by looking at both these aspects in an integrated way.

Our recent, complementary work [3] describes the Composition Pattern mostly as a *software* architectural pattern, resulting from the multiple refactoring efforts on the iTaSC software framework [4]; in contrast, this paper focuses on the Composition Pattern as *methodology* to systematically create robot *applications* by developing and composing reusable, compatible, and interoperable building blocks. This methodology can be applied to the software frameworks, tools, or languages prefered by developers.

The paper will use a tomato pick-and-place application as running example throughout the paper. In this application, a PR2 robot [5] has (i) to find a tomato located in the neighbourhood of a dedicated pick-up spot, (ii) to pick up the tomato, obviously not damaging it, (iii) and to deposit it in a dedicated basket a few meters away. The platform has to operate in a cluttered and populated environment, as shown in Figure 1. It is evident that all these tasks should take into account the limitations of the platform, and that the whole setup conforms to safety requirements. The running example is a typical pick-and-place robot application. It is rather 'simple' to pre-program this in an *ad-hoc manner*, provided that the robot operates in a human-shielded environment, and pick and drop location are within reach. However, when any of these limiting simplifications must be relaxed, developing the application quickly increases in complexity. Hence it becomes relevant to adopt a *methodology* that helps creating reusable

Fig. 2: Example grasping strategies to pick up a tomato: grasping the tomato using the gripper (left), or grasping it between the two grippers (right). The orange forms an obstacle when grasping the tomato. In an alternative scenario, the orange must be grasped, and the tomato avoided.

and adaptable applications. The methodology introduced in this paper aims at helping developers to deal with this escalating complexity, which comes in many forms. For example when (i) *changing the platform*, e.g. replacing the PR2 by an autonomous humanoid robot; or adding a sensor to the PR2, and use this information where useful; (ii) *changing the tasks*, e.g. grasping the orange, instead of the tomato; grasping the tomato between the two (closed) grippers to avoid squeezing it, rather than using a single gripper (Figure 2); increasing the number of tasks to execute simultaneously, e.g. grasping the orange with the other gripper; or executing the tasks in a more cluttered and populated environment with high levels of uncertainty, e.g. when human actions obstruct the view on the tomato; (iii) *changing the knowledge level*, e.g. replacing the given sequence of sub-tasks by a high level goal and a reasoning algorithm.

The paper uses constraint-based optimization as a unifying approach to create robot task descriptions: every task is a set of objective functions and constraints that the robot controller has to satisfy, with contributions from joint space, Cartesian space, and/or sensor space. A major reason for the growing success of the constraint-based approach is that constraints and objective functions are *composable*. This paper exploits this composability property by applying the constraint-based approach not just strictly to the robot tasks alone, but to all entities of a complete robotic application, such as platform-specific constraints, or constraints imposed by the manipulated object (Figure 4).

The paper is organized as follows: The following, Related Work section links existing approaches in literature to this paper. The next section states the Composition Pattern and describes its underlying concepts. The subsequent section applies the Composition Pattern to the example domain of constraint-based programming. Next, a discussion section details the benefits of the Composition Pattern and its role in reuse and refactoring. Further, this section compares the concepts introduced in this paper to existing approaches. Finally, the last section states the conclusions.

RELATED WORK

This section discusses related work on constraint-based programming, and existing architectures, frameworks, and methodologies for robotics.

Constraint-based programming

One constraint-based programming approach, named **i**nstantaneous **Ta**sk **S**pecification using **C**onstraints (iTaSC) [6]–[8], introduces particular sets of auxiliary coordinates to express task constraints and model uncertainty. These auxiliary coordinates are specified between *object frames* defined on the robots and objects involved in the application. Where possible, these object frames have a *semantic* meaning in the context of the task, for example a specific 'corner of a table'. The *composition* of the constraints of all (sub-)tasks, defined on possibly a multitude of robots, objects, and sensors, translates to a numerical *constrained optimization problem*. The developer can introduce *weights* and/or *priorities* between the different concurrent tasks. In the instantaneous version, a *solver algorithm* computes at each moment in time the best setpoints (for example joint velocities or accelerations) for all the robots involved in the application. A software framework and modelling tools for iTaSC are available [4], [9].

Related approaches that define task specification as a constraint-based optimization problem include the Stack of Tasks (SoT) [10] framework and the Stanford Whole-Body Control framework (SWBC) [11]. The concept of constraint-based task specification and control to define the overall robot task as a composition of individual composable constraints will prove to match the composition in the Composition Pattern, as will be detailed further on. Hence it makes an apposite choice as example domain.

Frameworks, architectures, and methodologies

Past research resulted in different frameworks, architectures, and methodologies to deal with complexity in robotics. Kortenkamp and Simmons give an overview of robot system architectures in [12]. The following paragraphs give an overview of recent advances.

A first type of frameworks uses *hierarchical (concurrent) state machines or flow charts*, as pioneered by Nilsson [13]. Control-focused frameworks of this type include Skill/Manipulation Primitive Nets [14], [15], which provide state machines of hybrid force/position control setpoints, and more recently LightRocks [16], which extends this idea using a modeling approach and introducing levels of abstraction of task specifications built on hybrid force/position control. General application-focused frameworks of this type include SMACH [17] and ROSCo [18].

Another type of frameworks starts from a *multi-tiered architecture* [19]. Angerer et al. [20] present a recent two-tiered object-oriented architecture for industrial robotics, robAPI. It consists of a robotics API tier, comprising a command and an activity layer, and a real-time robot control core tier.

The increase of knowledge and interactions between parts of knowledge results in the need for tools to manage this knowledge. Two state-of-the-art *knowledge driven approaches* include CRAM and the high-level mission specification by Doherty et al. [21]. CRAM [22], a light-weight reasoning mechanism that can infer control decisions, merges the features found in the planning and sequencing layers of 3T (three-tiered) architectures [19]. Examples of the application of this framework to complex situations include two robots cooperating in making pancakes [23]. Doherty et al. [21] present a formal framework and agent-based software architecture based on delegation for automated specification, generation, and execution of high-level collaborative missions.

In contrast to the frameworks and architectures above, this paper does not introduce the 'single best system architecture', but helps developers in defining a system architecture that fits their application's needs using a systematic approach.

Next to the frameworks and architectures mentioned above, there are a number of *framework eco-systems* that use data flow or component-based techniques. These framework eco-systems allow developers to create large systems from modular

components or nodes that encapsulate certain functionality. These components are intended to be substitutable blocks of computation that communicate data or events with other components. Component-based tools possibly allow to call functions (services) on other components. Examples include modeling framework eco-systems such as LabVIEW [24] and Simulink [25], and code-oriented framework eco-systems such as ROS [26] and Orocos [27].

This section further compares different framework eco-system aspects using the terminology of the 5C's principle of separation of concerns [28], [29]. This principle separates the *communication, computation, coordination, configuration,* and *composition* aspects in software functionality, and forms a basis for the here introduced Composition Pattern. In addition, this paper considers an *entity* as a concept or model that maps to software components, agents, objects, modules, processes, activities... The framework eco-systems primarily focus on *functional entities*, conforming to algorithms or computations (data processing), and their communication. *Support entities* that 'manage' functional entities, by handling configuration, composition, coordination, monitoring, and scheduling, are of secondary importance for most of these framework eco-systems; the introduction of support entities as well as the consistency of their usage, are generally left to the programmer. In contrast, the Composition Pattern introduced in this paper introduces these support entities in a systematic way, already in the conceptual and architectural design phase.

Most framework eco-systems provide the possibility to separate *configurable* parameters from the computation functionality, for example using the ROS parameter server or Orocos properties. The *composition* of components is fixed by design and possibly hierarchical in the LabVIEW and Simulink case, ROS and Orocos on the other hand, allow flat but runtime changeable compositions. State machines are commonly used for *coordination*, for example rFSM [29], SMACH [17] for ROS, or Stateflow for Simulink [30]. The number of *scheduling* options varies among the tools, for example the implicit scheduling based on block connectivity as default in Simulink, or its more advanced Common Function Call Initiators. Orocos assigns periodical (timer triggered), non-periodic (user triggered), or slave (coupled to another) activities ('threads') to components, and allows to choose a real-time scheduler or not. All of these framework eco-systems regard *monitoring* as a functionality to be created by the programmer, similar to other computations.

This paper does not focus on the functionalities offered by these frameworks, but on the structured and systematic approach to application (architecture) design and the resulting consequences for software engineering design, which can be applied to the framework eco-system of choice.

COMPOSITION PATTERN: CONCEPTS FOR A SYSTEMATIC APPROACH

This section defines *concepts* to divide a (robotics) problem into sub-problems. These concepts apply throughout the design, from the conceptual design to the software modelling phase.

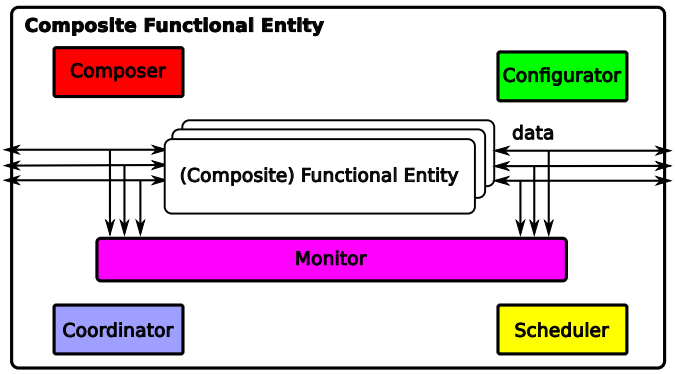


Fig. 3: Entity types within the Composition Pattern, represented by a different color. Each block represents an entity of a specific type: a `Composer`, `Configurator`, `Coordinator`, `Scheduler`, `Monitor`, and one or more (`Composite`) `Functional Entities`. All entities within the composite interact with each other. Moreover, (`Composite`) `Functional Entities` communicate data among each other, and across the composite boundary.

The Composition Pattern approach of application development consists of following four concepts: *metamodeling*, *composition*, *hierarchy*, and *semantic context*.

Metamodeling

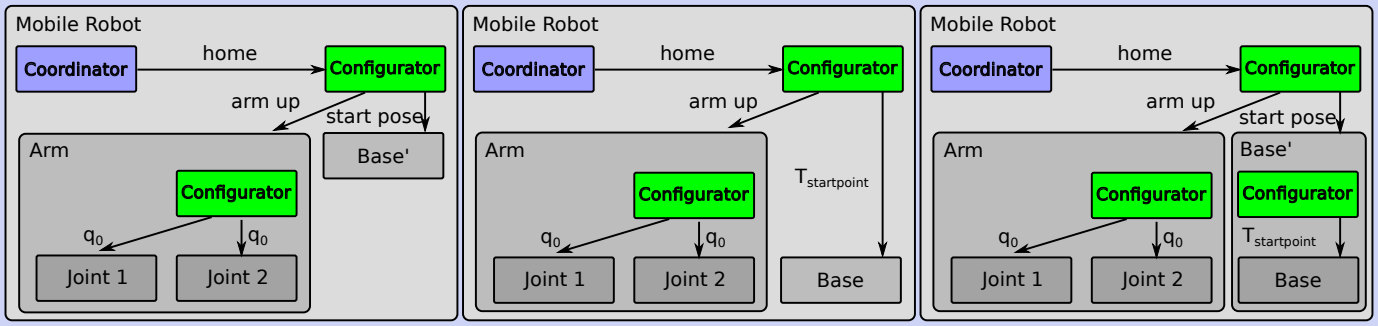
This paper follows the meta-model approach and terminology of Model Driven Engineering [31] as advocated by Bézivin [32]. It considers *all entities to be models*, as opposed to the code-centric principle of *all entities are objects*. We restrict ourselves to the key concepts of metamodeling relevant for this paper, and refer the reader to the work of Bézivin [32] for a discussion of the consequences of this paradigm shift.

In this paper, a *model* captures a view or aspect of a system, it groups semantics. A *meta-model* presents the language to describe a model; it is a formal specification of an abstraction of a (sub)-domain. A model *conforms to* one or more meta-models. An *implementation* is an instance of, or is represented by a model. From a model a concrete implementation can be generated or hand-coded. However, this paper will not elaborate on implementations; we refer the reader to Vanthienen et al. [3] for a discussion on implementations.

In the examples of following sections, teletype font names indicate `meta-models`[1], and italic font names indicate *models*. For example a *Tomato* `Object` denotes a *Tomato* model, conforming to the `Object` meta-model. De Laet et al. [33] present one example of the use and usefulness of metamodeling in robotics, compatible with the Composition Pattern. In this example a `Geometric Semantics` meta-model presents a language and rules on geometric relations and operations between rigid bodies. A concrete model represents the semantics of a specific relation or operation, for example

[1] `Teletype font names` will also be used to indicate a non-specific model that conforms to the meta-model with the same name, in places where it is clear from the context. For example a `Task` indicates a non-specific task model that conforms to the `Task` meta-model.

Box 1: Example composition and inter-context translation



The figures above consider the example in which the *home* position should be configured on a mobile robot consisting of a mobile base and an arm. A *Mobile Robot* entity composes a *Base'* or *Base* entity, and an *Arm* entity. The latter *Arm* entity is common to the three figures, it composes the $Joint_i$ entities. Hence, the $Joint_i$ entities are at depth two (D2), and the *Arm* entity at depth one (D1) with respect to the root composite *Mobile Robot* at depth zero (D0). The leaf nodes (functional entities) control the base as a whole (*Base'* or *Base* entity), and each joint of the arm respectively ($Joint_i$ entities). In this example the three depth levels coincide with different levels of abstraction. Entities at a deeper depth level have a darker shade of grey. We consider three variations of this example, as shown from left to right:

1) In the figure to the left, the D0 *Mobile Robot Configurator* configures *Base'* and *Arm* with parameters that belong to the same level of abstraction: 'arm up' and 'start pose'. The *Base'* is able to interpret and act on parameters of this level of abstraction. The support entities of the *Arm* composite further translate this parameter to the concrete numerical joint setpoints q_0.
2) In the figure in the middle, *Base* is an entity that can only interpret parameters of a lower level of abstraction, similar to the level of abstraction of the $Joint_i$ entities. The D0 *Mobile Robot Configurator* is adapted with respect to the left figure, to translate 'home' to 'arm up' and '$T_{startpoint}$', a concrete pose of the base. However, the need for adaption of the D0 *Mobile Robot Configurator*, as well as the translation to different abstraction level is regrettable from a design point of view.
3) In the figure to the right, the D2 *Base* entity is wrapped by a composite *Base'* that translates the 'home' received from the D0 *Mobile Robot Configurator* to '$T_{startpoint}$', which *Base* can interpret. This case represents one possible way to integrate the existing *Base* entity in the left figure composite, while preventing duplication of parts of models or code, such as the D0 *Mobile Robot Configurator*.

The first two examples demand less effort to develop. However, in the long term, the third option, consisting of fine-grained entities with depth levels coinciding with levels of abstraction, will prove to be more reusable.

the *End-Effector Pose* of a robot. This model can be translated to an implementation using an existing geometric library such as KDL [34] or the ROS geometry stack [35]. The metamodeling approach enables automatic checks for semantic correctness of geometric operations and representations, and their correct deduction.

Composition

A **Composite Functional Entity** (Figure 3), further referred to as 'composite', explicitly separates all aspects of composing ('grouping') different functionality based on following questions:

- What is the core *behavior* of the composition? This forms one or more **Functional Entities** (computation) of a composition. It typically represents continuous time and space behavior. A Functional Entity within a composite can be replaced by a Composite Functional Entity.
- How is the functionality *interconnected* within the composite? This forms the **Composer** entity.
- How to *coordinate* the behavior of this group of functionalities? This forms the **Coordinator** entity, which commands actions from the other entities within a composite, which on their turn report back to the Coordinator. It gives the composite the autonomy to handle certain situations locally.
- How to apply *configuration* to this functionality? The **Configurator** applies settings, i.e. data and parameters, to an entity, when triggered by the Coordinator. In this step the Configurator translates the data and parameters to the context of the entity. Therefore, the Configurator is a 'parameter translator' and the point where knowledge from a knowledge base can be introduced. Klotzbücher et al. introduced this separation of commanding and executing configuration as the Coordinator-Configurator pattern [36].
- What conditions of this composite need to be *monitored*? This forms the **Monitor** entity that reports on conditions of the functionality within the composition.
- Which *timing* constraints are important for this group of functionalities? This forms the **Scheduler** entity.

It is however possible that not all concerns are relevant for the composite at hand, and hence certain entities can be left out. For example the Monitor can be left out, when the composite has no data to monitor.

Hierarchy

The Composition Pattern helps to derive a set of modular entities as building blocks that are easily adapted or replaced, since each entity's behavior has a limited scope (separation of concerns), and a clear meaning. Applying the composition pattern iteratively, results in a tree of entities with a recurring, fractal structure. The level of granularity of the leaf nodes of the composition tree, i.e. their 'depth', does not need to be identical for all branches of the tree. Hence each Functional Entity can be replaced by a composite, until the granularity required by the specific application is

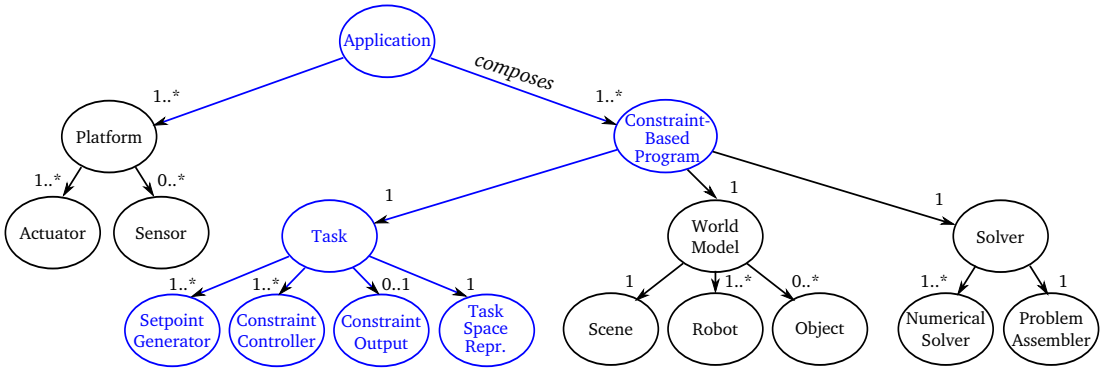


Fig. 4: Composition tree of a robot constraint-based application. A node represents an entity meta-model. An arrow indicates composition: the node at the beginning of the arrow composes entities that conform to the meta-model at the end of the arrow, with a multiplicity indicated next to the arrow. Moreover, a composite functional entity can compose entities of the same meta-model, which is not shown on the figure. For example an entity conforming to the `Task` meta-model can compose different entities that conform also to the `Task` meta-model. The application shown on top is the root composite, the entities shown at the bottom are the leaf entities considered here. The text will focus on the branch of the tree shown in blue.

achieved. At a design phase, a trade-off needs to be made based on considerations such as the existing functionality at your disposal and efficiency. In a refactoring phase, these levels can change, allowing for an incremental evolution of the application. Remark that as a consequence of the tree of composition all Functional Entities (computations) are always leaf entities.

Although the composition is strictly hierarchical, the 'communication' (fifth C of the 5C's principle of separation of concerns) does not need to be hierarchical. Entities communicate data on the same level of (compatible) semantics, although they may reside on different depth levels, therefore crossing different composition boundaries. The common parent entity checks and connects communication channels. For example a *controller* entity communicates setpoints to the *joint1*, *joint2*, and *base* or *base'* entities presented in Box 1. As a consequence a flat or a hierarchical composition are similar from the perspective of data-flow between Functional Entities, since data is not bound to the limits of a composite.

Semantic context

Every composite forms a **semantic context**; i.e. the entities within a composite use a shared vocabulary. The support entities translate from the context of a composition to the context of its child functional entities. This concept is important for knowledge driven architectures, where this context needs to be explicit. Box 1 gives an example of the relation between the *Mobile Robot*, *Base*, and *Joint* contexts. In the running example, higher level compositions use tomato-specific semantics, such as 'a rotten tomato' while the lower level composition that generates robot motions, uses robot-specific semantics, such as 'joints'.

The semantic context also forms a 'boundary'. The support entities of a composite 'know' only about the other entities within that composite, not the composition of the parent or child Functional Entities. Moreover, the Functional Entities of a composite do not know about the support entities that *manage* them, they send and receive data and events not knowing who will use or react on them. It does not imply information hiding however: child entities can be introspected or reasoned about.

The following section details the application of the presented approach to the domain of constraint-based programming.

Applying the Composition Pattern to Constraint-Based Programming

This section explains how robots applications can use the concepts of structured application development introduced in previous section. It describes a generic division of the domain of constraint-based programming in a composition tree.

Figure 4 gives an overview of the composition tree of a constraint-based application. Each node of the shown tree represents a meta-model of a (composite) functional entity. On the one hand, it shows the relation between meta-models. On the other hand, it shows the hierarchy of composite functional entities, complementary to the composition as shown in Figure 3 (each node has the structure as shown in Figure 3). This section focuses on one branch of this tree, shown in blue, choosing a limited number of entities of a composition to detail further on.

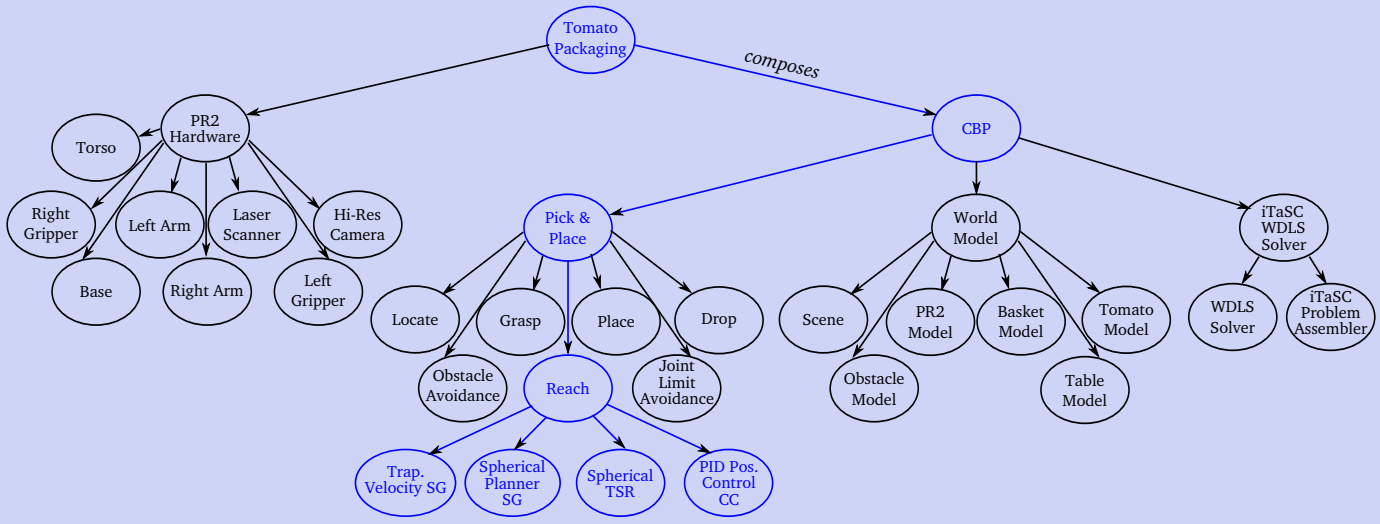
Furthermore this section applies the division represented by this tree to the running example, resulting in concrete models shown in Box 2.

We define following (composite) entities within the task specification branch of the application tree, from root to leaf: `Application`, `Constraint-Based Program`, and `Task`.

Application

An `Application` attaches a `Constraint-Based Program` to specific `Platforms` (hardware resources, which can be virtual for simulation). These `Platforms` consists of the specific robot `Actuators`, i.e. motion capabilities, and `Sensors`, i.e. sensing capabilities.

Box 2: Composition structure of the running example



The figure above presents the composition tree for the running example, using the same notation as in Figure 4 with the difference that a node is a (composite) functional entity *model*. It forms one of the possible models for the problem at hand.

The composition tree examplified above does not present the only possible hierarchy. First, the depth of the composition tree does not need to be restricted. For example a `Setpoint Generator` generating position setpoints can be implemented or replaced by a set of single DOF trajectory generators. In this case, the `Coordinator` of the now composite `Setpoint Generator` entity manages the different possible timings between the six child `Setpoint Generators`.

Moreover, intermediate composition layers can be introduced. For example an intermediate layer can be introduced between the *Pick and Place* `Task` and the different sub-`Tasks` (not shown in the figure). More concretely, the *Reach* `Task` could be replaced by a composition of the current *Reach* `Task` with an *Arm Guide* `Task`. The latter constrains the arm to move within a tight subspace when grasping a tomato in a hard-to-reach location.

Even more, entities can have multiple roles. For example the `Setpoint Generators` considered in the running example (*Trapezoidal Velocity SG* and *Spherical Planner SG*) deliver a fixed setpoint or a time-dependent stream of setpoints deduced from a motion profile. These `Setpoint Generators` get their goal from another entity, such as the *Configurator* of the *Pick and Place* `Task`, or a *Planner* (also an instance of a `Setpoint Generator`) outside or inside the scope of the *Reach* `Task`. However a `Setpoint Generator`, delivering setpoints to the `Constraint-Controller` can be of a different form, or defined outside of the scope of the `Task`. For example the haptic teleoperation scheme using iTaSC introduced by Borghesan et al. [37]. In this scheme the `Constraint-Output` of the position-coupling `Task` at the master side forms the `Setpoint Generator` of the equivalent `Task` at the slave side, and vice versa.

The running example will make use of the *PR2* and the default sensors of the platform: the tilting laser scanner and (stereo) cameras on the head.

Constraint-Based Program

A `Constraint-Based Program` (`CBP`) defines task specification and control on a robot setup. It composes a `Task` and attaches the `Task` to the world model, at certain points where we define *object frames*. The `Constraint-Based Program` also comprises a `Solver` that computes the control input to the robot as a solution of the constrained optimization problem. The `World Model` consists of the `Robot`- and `Object` (kinematic and dynamic) models placed in the `Scene`.

The `Constraint-Based Program` of the running example composes following models: (i) *Pick and Place* `Task`, (ii) a *World Model*, (iii) and a *Weighted-Damped Least-Squares Solver*. The *World Model* composes a *Scene*, a *PR2 Model*, conforming to the `Robot` meta-model, and models that conform to the `Object` meta-model: one or more *Obstacles* to avoid, the *Table* where to pick up the tomato, the *Basket* where to put the tomato, and the *Tomato*.

Task

A `Task` can compose different sub-`Tasks`, in which case an intermediate composition level is introduced. To make the distinction we will define a *Composite Task* composing different `Tasks`. The *Composite Task Coordinator* coordinates the active set of tasks. It commands different global weights and priorities as well as (abstract) goals for the tasks.

The *Pick and Place (Composite)* `Task` of the running example can be implemented using different combinations of tasks. The *Composite Task* developer chooses these `Task` entities, by (re-)using existing `Tasks` from a library, or by asking a `Task` developer to develop a model and implementation that fits the purpose. For the running example, he chooses to model the desired behavior using following `Tasks`:

- a *Locate* `Task` to actively look for the tomato, defined between the tomato-sensor hardware (a camera or laser-scanner for example) and the pick-up spot,
- a *Reach* `Task` to reach for the tomato once found, defined between the tomato and a gripper,
- a *Grasp* `Task` to grasp the tomato, also defined between the tomato and a gripper,
- a *Place* `Task` to position the tomato in the basket, defined between the tomato and the basket,

- a *Drop* `Task` to simply release the tomato, defined on the gripper,
- platform related safety `Tasks` such as *Joint Limit Avoidance*, defined on the joints of the robot (configuration space),
- and *obstacle avoidance* `Tasks`, defined between obstacles and robot parts.

Some of these tasks will be executed sequentially (locate - reach - grasp - position - drop), others in parallel. The *Pick and Place Coordinator* decides on this behavior. Remark that the resultant robot behavior emerges from the composition of the constraints of all active `Tasks`, for example a simple *Reach* `Task` will only be successful if combined with the necessary safety and obstacle avoidance tasks.

A single `Task` consists of a set of constraints on a task space representation. It is however unaware of its concrete purpose within an application. Even the object frames in between which the `Task` is defined are unknown to the `Task`. It is the parent of the `Task` that defines its purpose by coordinating, configuring, scheduling, monitoring, and composing it.

For example the *Reach* `Task` of the running example composes alignment constraints to align the gripper with the vector between the object frames in between which the *Reach* `Task` is defined, and an approach constraint that reduces the distance between these object frames. It is the *Pick and Place* `Task` that defines the object frames to be on the *Tomato* and the *Gripper* models.

At a meta-model level, a `Task` composes:

- a `Task Space Representation` (`TSR`), which defines a representation of the task space, e.g. a spherical coordinate system for the *Reach* `Task`;
- a formulation of a `Constraint-Output` equation (`CO`), which defines the output as a function of the state of the `TSR` and the joint space (*joint coordinates*), e.g. the selection of the spherical coordinates of the `TSR` forms the `CO` for the *Reach* `Task`;
- one or more `Constraint-Controllers` (`CC`), which define the controller on the output, e.g. a position controller imposes the constraints for the *Reach* `Task`;
- and one or more `Setpoint Generators` (`SG`), which define the desired values of the output at each time instance, e.g. an interpolator and a planning algorithm deliver the setpoints for the `CC` of the *Reach* `Task`.

In the running example, the *Coordinator* of the *Reach* `Task` decides when to switch between the two provided `Setpoint Generators`. Box 3 details the interaction of the *Reach* `Task` with its parent and leaf entities. Other examples of `Tasks` of the running example include a set of inequality constraints for each joint of the platform, which implements the *Joint Limit Avoidance* `Task`, and a simple open-close algorithm monitoring a 'touch' condition, which implements the *Grasp* `Task`.

The presented composition stimulates developers to make all assumptions on safety or platform specific constraints explicit, structured following Figure 4. Safety and platform specific constraints such as joint limit avoidance, center of mass requirements for humanoids etc. are introduced as `Tasks`, since they constrain the robot platform in the same way as any other `Task`. Making these `Tasks` explicit, allows human or artificial reasoning on the full active set of `Tasks`, without the need for discovering hidden assumptions. In the simplest case, the developer has to define these `Tasks` himself. However, tooling can add these safety and platform-specific `Tasks` automatically, based on the selected platforms.

Remarks

The tree of composition is a basic blue print for applications using constraint-based programming, it is a policy to use the Composition Pattern that gives definitions to guide decisions and achieve rational outcomes. However some level of flexibility remains as detailed in Box 2. Abovementioned subsection gives an example where an intermediate *Composite Task* level of composition is introduced. Moreover, the here described tree is not intended to be 'complete': entities not mentioned in the division can make part of a composite, for example `Estimators`.

Although presented in a top-down order for readability, the typical workflow will start at an intermediate composition level, composing existing (composite) entities from libraries. For the running example, the applied workflow was (i) first the development and choice of the `World Model` and `Solver` within a `Constraint-Based Program`, (ii) second the development of the *Composite Task* by selecting the `Tasks` and their interactions, (iii) and last the embedment in hardware, creating the `Application`.

Remark that each composite or functional entity gives rise to a different user perspective at a certain level of abstraction, demanding a different (level of) expertise.

Further remark that the different entities in the running example can be applied more generally, outside the scope of tomato picking. For example the approach to tomato picking can be generally applied to ball-shaped objects. However, naming of entities and events of the running example are kept within the scope of the example for readability. The following section will discuss this generality.

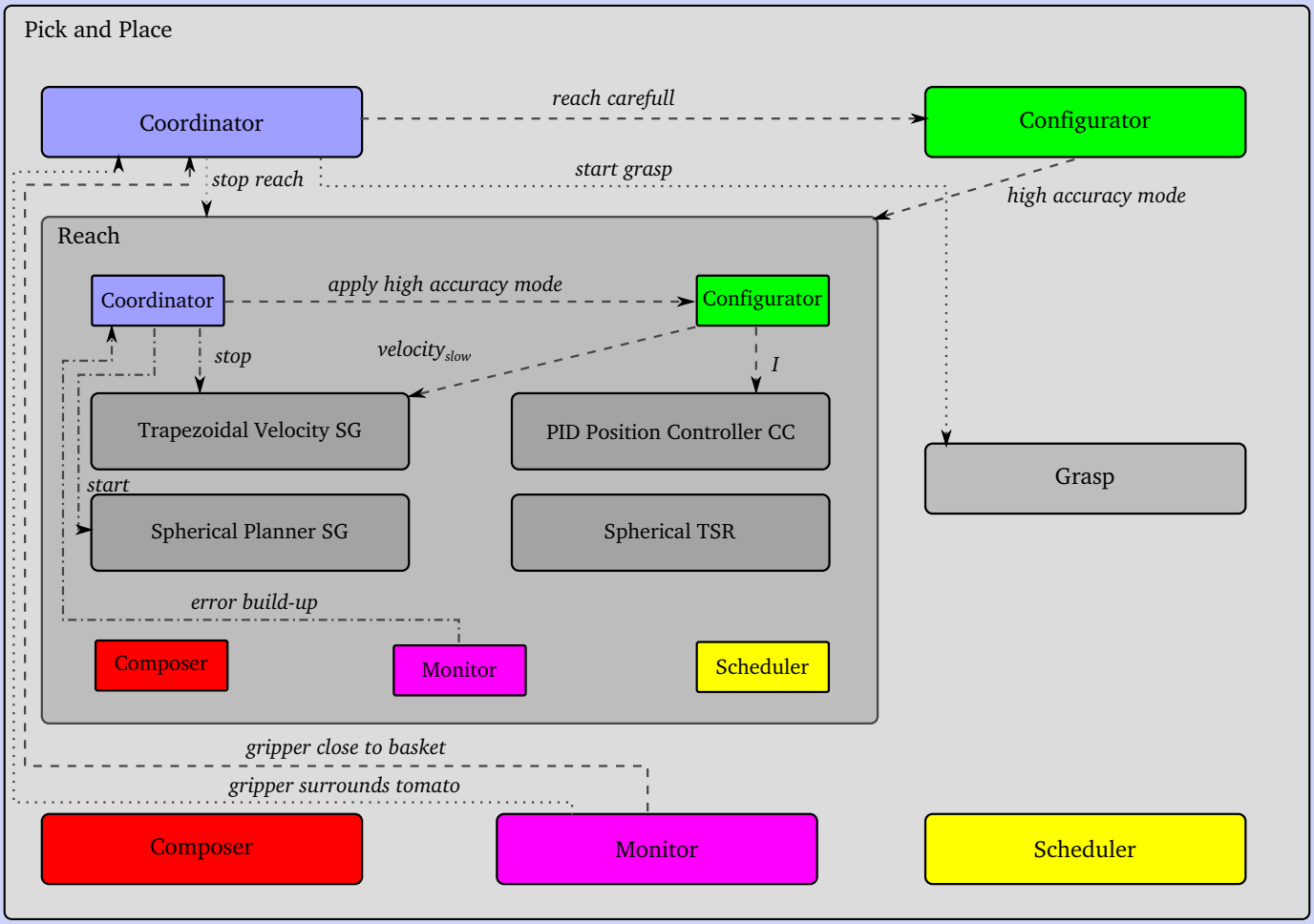
DISCUSSION

This section first discusses the implications and benefits of the Composition Pattern, secondly it discusses the role of the Composition Pattern in reuse and refactoring, lastly it discusses the relation to existing frameworks, architectures, and methodologies.

Implications and benefits of the Composition Pattern

The Composition Pattern helps the application developer to avoid following 'bad design' traits [38]: *rigidity*, i.e. when every change has its effect on too many parts of the system, *fragility*, i.e. when a change breaks unexpected parts of the system, and *immobility*, i.e. when the reuse of a piece of the system is hard since it is entangled with the application it was first designed for. These three 'bad design' traits characterize their respective opposites: flexibility, robustness, and reusability. We interpret these traits not only as a static

Box 3: Example entity interaction



The figure above details the interaction of the *Reach* `Task` with its parent, i.e. *Pick and Place*, and children, i.e. *Trapezoidal Velocity Profile* `Setpoint Generator`, *Spherical Planner SG*, *PID Position Controller CC*, and *Spherical TSR*. The figure shows entities at a deeper depth level in a darker shade of grey.

The following paragraphs elaborate the different interactions, starting from the interactions of the *Pick and Place Coordinator* with the different entities of the *Pick and Place* composite. Remark that in order to interact, each composite needs common vocabulary, which differs from the other composites. The support entities translate between the composite's own vocabulary to the vocabulary of their child (composite) functional entities, as will be exemplified in following paragraphs.

The responsability of a `Coordinator` is to change behavior by interpreting and reacting on events. First the `Coordinator` coordinates the deployment of the composite when triggered by its parent. For example, the *Pick and Place Coordinator* orders the *Pick and Place Composer* to interconnect the different `Tasks` and support entities, and the *Pick and Place Configurator* to load initial configurations to all entities, including all support entities. For example, the *Pick and Place Scheduler* is configured to schedule all concurrent `Tasks` in parallel.

Further the `Coordinator` interprets and reacts on events within the composite. For example, the *Pick and Place Monitor* monitors and signals events such as the *gripper close to Basket* condition, which triggers the *Pick and Place Coordinator* to demand the *high accuracy mode* explained in the following paragraph. The same *Pick and Place Monitor* signals *gripper surrounds tomato*, which triggers the *Pick and Place Coordinator* to transition to the *Grasp* `Task`. The dotted arrows indicate the latter. Remark that a `Monitor` signals the (non-) violation of a condition, not the expected reaction on that condition.

Furthermore the `Coordinator` triggers configuration. For example in the running example, the *Pick and Place Coordinator* commands to *reach careful when close to Basket*. The *Pick and Place Configurator* translates this command to a *high accuracy mode* configuration of the *Reach* `Task`. The *Reach Coordinator* and *Configurator* translate on their turn this mode to a lower approach speed configuration of the *Trapezoidal Velocity Profile* `Setpoint Generator`, and the activation of an integral term in the *PID Position Control*. The dashed arrows indicate these events.

The concrete translation values a `Configurator` uses, i.e. *configuration of the* `Configurator`, can be provided by different sources, including loading simple parameter lists or querying and reasoning on knowledge databases. For example, the speed configuration will depend on the controller type and the platform used.

Further a `Coordinator` has local responsibility more than the above presented translation of commands from higher levels. For example, in the running example, the *Reach Coordinator* switches from the *Trapezoidal Velocity Profile* `Setpoint Generator` to the *Spherical Planner* when the *Reach Monitor* signals that there is stall in the progress towards the goal indicated by a build-up in position error. The dash-dotted arrows indicate these signals.

architecture problem, but also as a dynamic, run-time, and behavior problem.

The Composition Pattern decreases **rigidity** since it *separates concerns* and *specifics are filled in as late as possible*. For example, configuration models what parameters to change and how to change them, avoiding coupling to other aspects, such as coordination or algorithms in functional entities. As an illustration, changing the execution rate of the running example at runtime will influence but should not alter any of its `Setpoint Generator` functional entities. Moreover, monitoring is important to decrease rigidity, since it allows an application to react on internal and external changes.

Further the Composition Pattern decreases **fragility**, because it makes the semantic *context* explicit, which keeps impact of changes *local* to a subpart of the composition tree. For example, monitoring boundary conditions, reacting (locally) on the violation of these conditions, and possibility querrying an external database for a solution in the local context, decrease fragility. As an illustration, disabling robot base movement in the running example, while the robot has to reach for the tomato outside the workspace of its arms, will cause the *Reach Monitor* to signal that no progress is made towards the goal, and the *PR2 Left Arm Monitor* to signal a stretched arm condition. The *Pick and Place Coordinator* can freeze the task execution, and signal this event. The latter will on its turn trigger operator interaction, or simply a re-activation of the base.

Moreover the Composition Pattern decreases **immobility**, because of the limited scope of a semantic *context*, the *granularity* of the composition tree hierarchy, the *modelling approach* and the *separation of concerns*. It captures a certain view of a system, making abstraction of implementation details. For example, a (composite) functional entity does not know its purpose, connectivity, or meaning within the application. Further an appropriate depth provides elementary entities, specific enough to be easily translated to code. As an illustration, the *Trapezoidal Velocity Profile* `Setpoint Generator` used in the *Reach* `Task` of the running example can be easily reused in the *Place* `Task` since its management and configuration are decoupled from its functionality. Moreover, not only leaf entities can be reused, for example the *Reach* `Task` on its own can be reused in another `Constraint-Based Program`.

Reuse and refactoring

The running example elaborated throughout the paper specifies only one possible pick-and-place robot application. Developing this example using the Composition Pattern requires more effort than an ad-hoc approach. However, common situations include the need to extend this application to multiple consecutive and/or concurrent tasks, port this task to another robot platform or object to manipulate. Applying the Composition Pattern reduces the effort of extending and revising the running example on a longer term. This need for extensibility, reusability, and adaptability forms one of the drivers to refactor robot application software. This section discusses some guidelines and examples to refactor existing applications to more reusable and adaptable systems.

Important is to **consider everything a model**, as advocated by model driven engineering, such that applications can be first modeled, analyzed and verified abstractly before code is generated, next to the advantages of conceptual simplicity, high scalability, and good flexibility [32].

A developer should detect the different forms of knowledge and expertise, and **divide** the application domain at hand: (i) in levels of abstraction, making general applicable subparts explicit, (ii) and the different forms of knowledge and expertise. This division results in the hierarchy (tree) of semantic contexts. An appropriate depth of the tree provides elementary entities specific enough for the available tools to be translated to code. It is a trade-off between *composability*, i.e. how easily an entity can be reused, and *compositionality*, i.e. the predictability of behavior of a composite knowing the behavior of its components [39]. For example, as mentioned in previous section, the models used in the running example are applicable to a wider scope of problems than tomato pick and place applications. Many parts of the *Tomato Constraint-Based Program*, such as the *Pick and Place* model, can be reused to handle for example oranges: only the *Tomato* model should be replaced by a *Orange* model, as do perception algorithms. Context dependent configuration parameters can be deduced from this altered model: the force used to grip the orange, the condition for successful approach, etc.

Each of the semantic contexts can be **separated in concerns**, more concretely the different entities of a Composition Pattern composite. More code-centric examples include: (i) If part of the code makes assumptions on *where* the data comes from, move this dependency (where) to the Composer. (ii) Replace magic numbers with configurable parameters, and move the concrete numbers to configuration. (iii) Break up if/else statements: move the condition to the Monitor, the reaction to the condition to the Coordinator, the concrete action to a Functional Entity, and timing related statements to the Scheduler. Remark that proper design is more than separation: each entity should be adaptable. For example Functional Entities should contain adaptable behavior. However, few algorithms are ready for this level of adaptability. For example the *Trapezoidal Velocity Profile* `Setpoint Generator` in the running example is a functional entity, which algorithm can be implemented in different ways. Depending on the underlying algorithm, it can assume a fixed rate of operation (sample time), can handle a change in rate of operation, or can handle asynchronous triggers (event-driven). The latter offers a higher level of adaptability.

Relation to existing architectures

The Composition Pattern generalizes concepts to which existing frameworks and architectures conform to a greater or lesser extent. In the first place the Composition Pattern delivers *structure*, not making claims on the actual behavior, nor limiting the structure to a number of tiers or levels. It provides a way to improve or create applications using for example the framework eco-systems mentioned in the Related Work section. Moreover, it provides structure beyond (coupled) coordination-configuration using hierarchical state machines or flow charts frameworks.

The Composition Pattern stimulates context structuring, but does not impose information hiding. It presents a (composite) functional entity as first-class entity, which can be inspected and reasoned upon, compatible with knowledge-driven approaches such as CRAM. Moreover, in future applications we want to integrate reasoning on all composites, on all tiers, as presented in the Composition Pattern, and in contrast to 2- or 3-Tier architectures of the Related Work section.

One consequence, together with the non-strict hierarchical communication, is local reaction on events. A powerful feature, which needs to be used wisely to avoid immobility and fragility. For example a 'motor broken' event can trigger the immediate deactivation of a task, without the need to trickle through hierarchical layers, such as in strictly hierarchical architectures e.g. JAUS [40].

Remark that the Composition Pattern applied to constraint-based programming resulted in a hierarchy of entities. Other task specification approaches use different forms of hierarchy. Certain frameworks, such as TaskNets [14], use hierarchies of a single type of entity, comparable to the relation of the `Task` entities and the *Composite Task* in the running example. Other frameworks, such as the High-Level Mission Specification [21], transform high level descriptions to low level descriptions, i.e. a reduction of system complexity through abstraction along the task dimension. However, the here presented hierarchy corresponds to higher levels of platform coupling, next to the task dimension within the task tree: the application level couples the 'abstract' program to a specific hardware, while a task is the abstraction of a set of constraints. Hence the hierarchy of the high-level mission is complementary with the here presented approach, and is topic of ongoing research.

CONCLUSIONS

This paper provides a methodology for systematic robot application development (integrating structure and behavior), formalized as the Composition Pattern. It does not limit itself to only 'bringing functionality together', but adds the important application design concepts of (i) metamodeling, (ii) composition (Coordinator, Composer, Configurator, Scheduler, and Monitor), (iii) hierarchy, and (iv) semantic context.

As strongest point, the Composition Pattern enables developers to deal with the increasing scale and complexity of robotic applications, as well as the resulting need for flexible, reusable, and adaptable software. Moreover, the methodology can be applied to the developers' framework eco-system of choice.

The paper shows how the methodology decreases the design pitfalls of rigidity, fragility, and immobility and gives concrete guidelines on reuse and refactoring. However, it does not provide the final answer on *how* to best apply the methodology to any new application.

Hence a lot more work is required to provide a broader set of structural and behavioral *models* within the robotics community, and the development of *tooling* to aid developers at creating applications; a wide and consistent application of the Composition Pattern might be a significant driver to accelerate these developments.

ACKNOWLEDGMENT

All authors gratefully acknowledge the financial support by the Flemish FWO project G040410N, KU Leuven's Concerted Research Action GOA/2010/011, KU LeuvenBOF PFV/10/002 Center-of-Excellence Optimization in Engineering (OPTEC), and European FP7 projects RoboHow (FP7-ICT-288533), BRICS (FP7-ICT-231940), Rosetta (FP7-ICT-230902), Pick-n-Pack (FP7-NMP-311987), and Sherpa (FP7-ICT-600958).

REFERENCES

[1] DARPA, "DARPA robotics challenge," http://www.theroboticschallenge.org, 2013, last visited May 2014.

[2] Robocup, "Robocup," http://www.robocup2014.org, 2014, last visited May 2014.

[3] D. Vanthienen, M. Klotzbücher, and H. Bruyninckx, "The 5C-based architectural Composition Pattern: lessons learned from re-developing the iTaSC framework for constraint-based robot programming," *J. Software Engin Robotics*, vol. 5, no. 1, pp. 17–35, 2014.

[4] D. Vanthienen, T. De Laet, R. Smits, and H. Bruyninckx, "itasc software," http://www.orocos.org/itasc, 2011, last visited May 2014.

[5] Willow Garage, "Willow Garage," http://www.willowgarage.com/, 2011, last visited November 2013.

[6] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *IJRR*, vol. 26, no. 5, pp. 433–455, 2007.

[7] W. Decré, R. Smits, H. Bruyninckx, and J. De Schutter, "Extending iTaSC to support inequality constraints and non-instantaneous task specification," in *Int. Conf. Robotics and Automation*, Kobe, Japan, 2009, pp. 964–971.

[8] W. Decré, , H. Bruyninckx, and J. De Schutter, "Extending the Itasc constraint-based robot task specification framework to time-independent trajectories and user-configurable task horizons," in *Int. Conf. Robotics and Automation*, Karlsruhe, Germany, 2013, pp. 1933–1940.

[9] D. Vanthienen, M. Klotzbuecher, T. De Laet, J. De Schutter, and H. Bruyninckx, "Rapid application development of constrained-based task modelling and execution using domain specific languages," in *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, Tokyo, Japan, 2013, pp. 1860–1866.

[10] N. Mansard, O. Stasse, P. Evrard, and A. Kheddar, "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks," in *Int. Conf. Advanced Robotics*, Munich,Germany, 2009.

[11] L. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical conrol of behavioral primitives," *Int. J. Hum. Rob.*, vol. 2, no. 4, pp. 505–518, 2005.

[12] D. Kortenkamp and R. G. Simmons, "Robotic systems architectures and programming." in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 187–206.

[13] N. J. Nilsson, "Hierarchical robot planning and execution system," Stanford Research Institute, Tech. Rep., 1973.

[14] U. Thomas, B. Finkemeyer, T. Kröger, and F. M. Wahl, "Error-tolerant execution of complex robot tasks based on skill primitives," in *Int. Conf. Robotics and Automation*, Taipeh, Taiwan, 2003, pp. 3069–3075.

[15] B. Finkemeyer, T. Kröger, and F. M. Wahl, "Executing assembly tasks specified by manipulation primitive nets," *Advanced Robotics*, vol. 19, no. 5, pp. 591–611, 2005.

[16] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using uml/p statecharts," in *Int. Conf. Robotics and Automation*, Karlsruhe, Germany, 2013, pp. 461–466.

[17] J. Bohren, R. B. Rusu, E. G. Jones, E. Marder-Eppstein, C. Pantofaru, M. Wise, L. Mosenlechner, W. Meeussen, and S. Holzer, "Towards autonomous robotic butlers: Lessons learned with the PR2," in *Int. Conf. Robotics and Automation*, Shangai, China, 2011, pp. 5568–5575.

[18] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. Kemp, "ROS Commander (ROSCo): Behavior creation for home robots," in *Int. Conf. Robotics and Automation*, Karlsruhe, Germany, 2013, pp. 467–474.

[19] R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237–256, 1995.

[20] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics API: Object-oriented software development for industrial robots," *J. Software Engin Robotics*, vol. 4, no. 1, pp. 1–22, 2013.
[21] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 1, no. 1, pp. 75–119, 2013.
[22] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM—A cognitive robot abstract machine for everyday manipulation in human environments," in *Int. Conf. Advanced Robotics*, 2010, pp. 1012–1017.
[23] M. Beetz, U. Klank, A. Maldonado, L. Mösenlechner, D. Pangercic, T. Rühr, and M. Tenorth, "Robotic roommates making pancakes," in *11th IEEE-RAS Int. Conf. on Humanoid Robots*, Bled, Slovenia, October 2011, pp. 529 – 536.
[24] NI, "LabVIEW," http://www.ni.com/labview/.
[25] The MathWorks, "Simulation and model-based design by The MathWorks," http://www.mathworks.com/products/simulink/.
[26] Willow Garage, "Robot Operating System (ROS)," http://www.ros.org.
[27] H. Bruyninckx and P. Soetens, "Open RObot COntrol Software (OROCOS)," http://www.orocos.org/, 2001, last visited November 2013.
[28] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," Robot Standards project, Tech. Rep., 2009, http://www.robot-standards.eu//Documents_RoSta_wiki/whitepaper_reuse.pdf.
[29] M. Klotzbücher and H. Bruyninckx, "Coordinating robotic tasks and systems with rFSM Statecharts," *J. Software Engin Robotics*, vol. 3, no. 1, pp. 28–56, 2012.
[30] The MathWorks, "Design and simulate state charts by The Mathworks," www.mathworks.de/products/stateflow/.
[31] Object Management Group, "OMG," http://www.omg.org.
[32] J. Bézivin, "On the unification power of models," *Software and Systems Modeling*, vol. 4, no. 2, pp. 171–188, 2005.
[33] T. De Laet, S. Bellens, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies (part 2): from semantics to software," *IEEE Rob. Autom. Mag.*, vol. 20, no. 2, pp. 91–102, 2013.
[34] R. Smits, H. Bruyninckx, and E. Aertbeliën, "KDL: Kinematics and Dynamics Library," http://www.orocos.org/kdl, 2001, last visited August 2012.
[35] T. Foote, "Robot Operating System (ROS) geometry stack," http://ros.org/wiki/geometry.
[36] M. Klotzbücher, G. Biggs, and H. Bruyninckx, "Pure coordination using the coordinator–configurator pattern," in *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems*, 2012.
[37] G. Borghesan, B. Willaert, T. De Laet, and J. De Schutter, "Teleoperation in presence of uncertainties: a constraint-based approach," in *10th IFAC Symposium on Robot Control (SYROCO)*, vol. 10, Dubrovnik, Croatia, September, 5–7 2012.
[38] R. C. Martin, "The dependency inversion principle," 1996, pp. 1–12, http://www.objectmentor.com/resources/articles/dip.pdf.
[39] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The BRICS Component Model: A model-based development paradigm for complex robotics software systems," in *28th ACM Symposium On Applied Computing*, 2013, pp. 1758–1764.
[40] JAUS, "JAUS toolset," http://jaustoolset.org/.

Journal of Software Engineering for Robotics 5(1), May 2014, 17-35
ISSN: 2035-3928

The *5C*-based architectural *Composition Pattern*: *lessons learned* from re-developing the *iTaSC* framework for constraint-based robot programming

Dominick VANTHIENEN Markus KLOTZBÜCHER Herman BRUYNINCKX
Department of Mechanical Engineering, University of Leuven, Belgium

Abstract—The authors are part of a research group that had the opportunity (i) to develop a large software framework (±5 person year effort), (ii) to use that framework ("*iTaSC*") on several dozen research applications in the context of the specification and execution of a wide spectrum of mobile manipulator tasks, (iii) to analyse not only the functionality and the performance of the software but also its readiness for reuse, composition and model-driven code generation, and, finally, (iv) to spend another 5 person years on re-design and refactoring.
This paper presents our major *lessons learned*, in the form of two best practices that we identified, and are since then bringing into practice in any new software development: (i) the *5C meta model* to realise *separation of concerns* (the concerns being Communication, Computation, Coordination, Configuration, and Composition), and (ii) the *Composition Pattern* as an architectural meta model supporting the methodological coupling of components developed along the lines of the 5Cs.
These generic results are illustrated, grounded and motivated by what we learned from the huge efforts to refactor the *iTaSC* software, and are now behind all our other software development efforts, without any exception. In the concrete *iTaSC* case, the Composition Pattern is applied at three levels of (modelling) hierarchy: application, iTaSC, and task level, each of which consist itself of several components structured in conformance with the pattern.

Index Terms—Software pattern, architecture, composition, robot programming, task specification

1 INTRODUCTION

ROBOTICS has evolved from a single manipulator arm to a broad field of fixed, driving, crawling, diving, sailing and flying robots with many, redundant degrees-of-freedom (DOF). Each of them equipped with a wide range of sensors, from simple encoders to point cloud generating laser scanners. Moreover, more and more different, concurrently active tasks are integrated on these platforms in ever more demanding scenarios, such as human-robot co-manipulation.

One of our research priorities is the development of a methodology to program such complex tasks—i.c. the *instantaneous Task Specification and estimation using Constraints* (iTaSC) [1]—and to provide developers with appropriate software support to facilitate reuse [2]. This paper focuses on what we learned along the way, as "best practices", to realise such large-scale software frameworks; these insights have been re-applied to the iTaSC software support context, which we use as a concrete application domain in this document, to make the generic, application-independent "best practices" more tangible, and the discussion about its pros and cons more concrete.

The focus of this paper is not on discussing the *functionalities* offered by the iTaSC framework or any of the frameworks mentioned in the related work section; nor on discussing their relative merits, but on their software engineering design. The outcome is a set of "best practices" on how to tackle future labour intensive software development efforts, such that they could be developed with less pain, and integrated better with

Regular paper – *Manuscript received November 14, 2013; revised April 28, 2014.*

- *This work was supported by the Flemish FWO project G040410N, KU Leuven's* Concerted Research Action *GOA/2010/011, KU LeuvenBOF PFV/10/002* Center-of-Excellence Optimization in Engineering *(OPTEC), and European FP7 projects* RoboHow *(FP7-ICT-288533),* BRICS *(FP7-ICT-231940),* Rosetta *(FP7-ICT-230902),* Pick-n-Pack *(FP7-NMP-311987) and* euRobotics *(FP7-ICT-248552). We are extremely grateful for the many constructively critical interactions with our partners in the BRICS, Rosetta and RoboHow projects; they added another two decades of experiences with large software design and development projects, providing ample material for the synthesis of many "best and worst practices". Further, the authors would like to thank the anonymous reviewers for their insightful comments on the paper, which contributed to the improvement of this work.*

other frameworks.

One of the *major lessons learned* by the authors, is that integration should *not* start at the level of the software code, but at the level of *models* of the provided functionality. In other words, the essential role of formal *Domain Specific Languages* (DSLs) will be stressed and illustrated at several occasions in the document. Remark that a transformation between models, and the generation of code from a model does not imply a "one to one" mapping; it can include optimizations based on "reasoning" on the model. A well known example of this principle are software compiler optimizations. In general such "model-to-x" transformations are a far from resolved problem, beyond the scope of this paper.

While this work refrains from introducing "the" best system architectures, it does propose an *architectural pattern* (or "meta architecture") that has proven to be a "best practice" to help developers in finding and expressing the (most often rather complex) system architecture that fits best to their application's particularities.

1.1 The 5Cs

The software pattern introduced in this paper builds on the *5C's principle of separation of concerns* [3], [4] separating the **c**ommunication, **c**omputation, **c**oordination, **c**onfiguration, and **c**omposition aspects in the overall software functionality. This earlier work reflects our insights, or "analysis" of the design problem, while this paper introduces our solution, or "synthesis", of how to provide constructive guidelines to system and component developers.

The authors consider the 5C's as their most often proven "best practice" in robotics software development, since it (gradually) emerged during the huge accumulated software development experience (section 2.4), and was applied to dozens and dozens of new software developments. Since two years, it is even the core of a course on *Embedded Control Systems* for first-year Master students in Mechanical Engineering, where it has proven essential to let them grasp, quickly and thoroughly, the high-level design challenges of a complex *system-of-systems*.

1.2 Outline and notation

Section 2 cites the related work and introduces the application domain. Sections 3–7 elaborate each of the five "5C" concerns, with a sub-section devoted to *modelling*, one on the *implementation*, one on discussion and lessons learned, and one on how to compose that concern in a bigger architecture. Section 8 states the conclusions of this paper.

The paper emphasizes entity[1] type names using `teletype font`, and instance names with *italic font*; names of events are emphasized using `teletype font` and begin with `e_`.

1. Entities, or components, agents, objects, modules, processes, activities… The concrete name has no real importance in the context of this paper.

2 Related Work

This section gives an overview of related work and introduces the application domain. It further states the experience that led to the formulation of the Composition Pattern.

2.1 Robot Systems Architectures and Frameworks

Different *architectures* and *frameworks* have been proposed to create large and complex robot systems, an overview can be found in the book chapter by Kortenkamp and Simmons [5]. This section discusses some relevant and more recent work.

A first set of frameworks use hierarchical (concurrent) flow charts or state machines to create large and complex robot systems [6]. Recent examples include *ROSCo* [7] and *LightRocks* [8]. The latter focuses on task specification and will be discussed in section 2.2.

Many robotic frameworks start from a multi-tiered architecture [9]. A recent two-tiered architecture, *robAPI* [10] aims at industrial robot applications. The first tier provides a real-time dataflow, and the second tier provides an object-oriented robotics API making abstraction of the real-time aspects, and dividing an application in actuators, actions, sensors, and state. Another example is the *BIP* (behavior, Interaction, Priority) framework [11], [12], which has a three-tiered architecture. It provides *formal models* for the discrete behavior, which allows for *Validation and Verification* of those parts of the robot task.

Recently, cognition-enabled approaches have gained more attention. For example *CRAM* [13], a light-weight reasoning mechanism that can infer control decisions. It is a two-tiered architecture, merging the planning and sequencing layers of 3T architectures [9]. Another example of a cognition-enabled approach is the formal framework and agent-based software architecture by Doherty et al. [14].

2.2 Application Domain: Task Specification

This subsection introduces the basic primitives of the application domain—specification and execution of complex robot tasks—that was chosen in this paper to illustrate the best practices in software development for large-scale robotics software frameworks. This introduction is not meant to be self-contained or exhaustive, hence the reader is referred to the references for further details.

Traditionally, robot programming methods specify the robot motion in either joint space or Cartesian space. In joint space the motion trajectory is directly imposed on the individual robot joints, and is often used for programming fast point-to-point motions. In Cartesian space, for example used for tool trajectory tracking, the robot motion is specified in a *compliance frame* [15], or *task frame* [16] (typically either a tool centre point (TCP) frame or a base frame). Besides motion-based control, also joint-specific, Cartesian wrench (i.e. force and torque), and impedance control schemes are often used in practice [17].

This approach has proven its effectiveness for (geometrically) simple tasks, however, it scales poorly to more complex tasks that involve multiple frames and multiple partial motion specifications, [16].

Constraint-based programming on the other hand does not consider the robot joints nor the single task frame as the central primitives in the specification. Instead, the core idea is to describe a *robot task as a set of constraints* (in *various* frames on the robot, in joint space as wel as in Cartesian or sensor space), and *one or more objective functions to optimize*. Samson et al. [18] presents this approach in a generic way, and De Schutter et al. [1] were the first to turn these generic ideas into a publicly available software framework. The latter, named **i**nstantaneous **Ta**sk **S**pecification using **C**onstraints (iTaSC), introduces particular sets of auxiliary coordinates to model uncertainty and to express task constraints. These task constraints are defined between *object frames* defined on robots and objects involved in an application. These object frames have, preferably, *semantic* meaning in the context of the task, for example the *point of a pencil*. Decré et al. [19] extended the framework to support inequality constraints.

A general iTaSC task is the *composition* of multiple sub-tasks, involving possible multiple *robots*, *sensors* and *objects*, and at the level of that composite task, *weights* and/or *priorities* between the sub-tasks can be introduced by the task programmer. This specification is then turned into a numerical *constrained optimization problem*, from which a *solver algorithm* computes the instantaneously best joint setpoints (e.g., joint velocities or accelerations) for the robot(s) at each moment in time, which are then sent to the lower-level actuator hardware controllers.

The key advantages of the "iTaSC paradigm" are: (i) a *systematic workflow* to define task constraint expressions [20]; (ii) the *composability of constraints*, since not only can multiple constraints be combined, but each of them can also be *partial*, that is, not constraining the full set of degrees-of-freedom (DOF) of the robot system or of the task space; (iii) *reusability of constraints*, since the (recent) DSL support allows to specify relation between object frames in symbolic form, hence with (potentially) more semantic and hence higher and more context-specific reusability; (iv) *derivation of the control solution*: the iTaSC methodology systematically evaluates the task constraint expressions at run time and generate setpoints for a low-level controller; (v) *modelling of uncertainty*: it provides a systematic approach to model and estimate uncertainties.

iTaSC is not the only software framework available for complex robot task specification. Three similar frameworks (developed independently and during overlapping periods in time) are known to the authors:

- *TaskNets*: Finkemeyer et al. [21] developed a control architecture and a software framework for the execution of Manipulation Primitive nets, including the integration of on-line trajectory generation [22]. Recently Thomas et al. provided the LightRocks [8] DSL for skill based robot programming.
- The *Stack of Tasks* (SoT) [23] framework provides a dataflow approach to the "Generalized Inverted *Kinematics*" computations required in complex compositions of several sub-tasks for the robot, in which the relative contributions of each sub-task can be prioritized with respect to the others.
- the *Stanford Whole-Body Control* framework (SWBC) [24] implements a hierarchical control structure, on the basis of *full-dynamics* "solvers". Also SWBC allows to establish priorities among several sub-tasks.

The single underlying paradigm of all these frameworks is that they rely on *a set* of *compliance frames* or *task frames*. Each of the task frames represents part of the overall task specification (which we call `Tasks` in the remainder of this text), and adds a set of *objective functions* and *constraints* to a *solver* that then has to compute the "optimal" solution to the (possibly overconstrained or underconstrained) overall constrained optimization problem.

In contract to SoT and SWBC, iTaSC and TaskNets introduce some extra software in their framework, namely *Finite State Machines*, to specify and execute also the *discrete behavior*, that is, the the sequencing of particular sets of sub-tasks (each of which specifies a continuous time/space behavior).

2.3 Relation to the paper

All of the frameworks mentioned in section 2 *have* paid attention to the *integration* challenge, but, invariably, this is still limited to "adding extra functionalities into our own framework", but not (yet) "integration of selected functionalities from different frameworks into the same application". Hence, the ambition of this paper is to explain how to (re)design software frameworks, such that the latter type of real integration can be supported in a more maintainable way; here, the "maintainability" context is that of independent "third parties", and not that of the original developers of the framework. "Real integration" also means that the provided functionality can be used as building blocks in *any other* system architecture than the one(s) used by the original developers.

Many of the architectures discussed in section 2 conform to a certain degree to the architectural Composition Pattern. However, none of these architectures is known to incorporate or separate all aspects of the pattern, explained in following sections, explicitly.

Moreover, the Composition Pattern does not limit the composition hierarchy to a fixed number of layers or tiers, nor to a hierarchy of "general to specific" layers of abstraction.

2.4 Lessons learned from refactoring the iTaSC framework

As mentioned before, the authors get their "best practice" insights mainly, but by far not exclusively, from the long-

term development efforts of the *iTaSC* framework, [1], whose functionality is summarized in Sec. 2.2.

The first *iTaSC* software was developed by Ruben Smits [25], influenced heavily by the features available at that time in our other large-scale software framework *Orocos* [26]. Both frameworks were, in themselves, already improved (and "decoupled") versions of our previous-generation (too) highly integrated robot specification and control software framework *COMRADE* that dates back to the early 1990s. [27], [28]. Recently, Vanthienen et al. [29] created a second-generation iTaSC implementation, profiting from the "best practices" presented in this paper; the major difference with the first generation is the higher degree of formalization and structure of the iTaSC paradigm, supported by a formal *Domain Specific Language*. Hence, a developer can create an iTaSC *model* of an application (instead of directly having to write the *code*), and that model is parsed, transformed into structured code templates, and then executed by a running instance of the *code framework* presented in this paper. The higher degree of formalization, separation of concerns, and the accompanying structure, enable developers to reuse tasks specified and implemented before in combination with other tasks to form a new application, and on any robot that can be represented by a kinematic tree. Moreover, it allows reuse on the even more fine-grained level of *only* the statechart ("Coordination", that is, the *discrete* behavior of a task). All this can now happen with much smaller configuration files that have to be changed during the reuse, compared to the first-generation version.

Examples of concrete limitations for reuse, adaptability, and extensibility, encountered in earlier work, which where solved using the "best practices" introduced in this paper, include: (i) conditional statements (if-then-else) in components that in fact do scheduling or coordination of the component, for example combining the procedure to bring a robot to a running state, with the (general applicable) kinematic algorithms to calculate end-effector positions; (ii) interfaces that communicate data, which are in fact events; (iii) the coupling of application specific configuration and monitoring with the functional behavior, inside a component.

In summary, the presented "best practices" are grounded in the accumulated software development experiences of several dozen researchers spanning more than 20 years of very focused framework developments, and several generations and types of computational and robotics hardware.

3 Composition

Composition is the first one of the *5C's* to be discussed. It models the *structure* of the coupling between the entities of the other concerns; those other concerns (Computation, Configuration, Coordination and Communication) model four complementary kinds of *behavior* in a system. The structural model in the Composition deals with two aspects: on the one hand, it groups entities together in *composites*, supporting

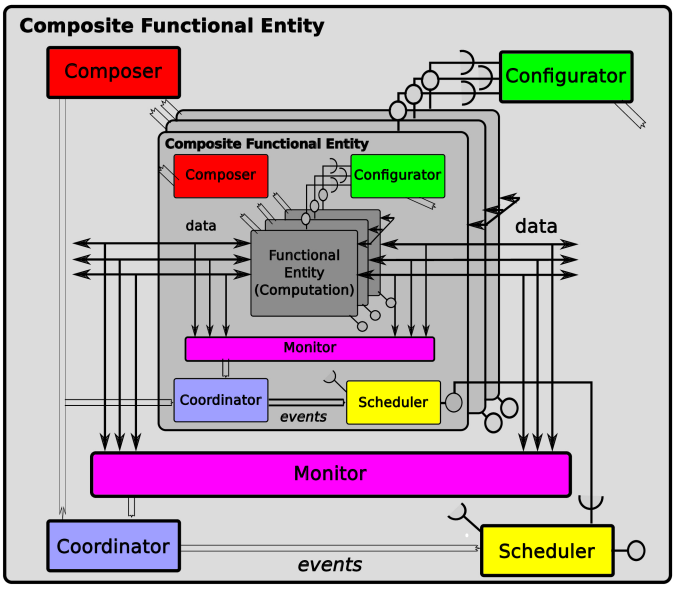


Fig. 1: Pattern of composition. Each block represents an entity, arrows indicate data communication, double lines indicate event communication, and a line with the lollipop-socket indicate event or service providing-requesting. The `Composer` (red), `Coordinator` (blue) and `Scheduler` (yellow) are "singletons" within a Composite `Functional Entity` (grey) because they all are "master" of the (possible multiple) (Composite) `Functional Entities`, `Monitors` (purple) and `Configurators` (green), at different phases in the composite component's life cycle. Each `Functional Entity` can be (replaced by) a Composite `Functional Entity`, which leads to hierarchy of compositions. A hierarchy with a depth of three is shown in the figure; a darker shade of grey indicates a (Composite) `Functional Entity` at a deeper depth level within the hierarchy.

hierarchy, and on the other hand, it models the **interactions** between the system entities. Composition (or "architecture") is a trade-off between *composability*, i.e. the property of an entity to be easily reused in a composition, and *compositionality*, i.e. the property of a composite to have predictable behavior knowing the behavior of its components [4]. To the best of the authors' knowledge, no scientific insights are known about how to optimize the architecture of complex systems; hence, Composition remains much of an art, while for the other C's described below, some more concrete design insights and guidelines do exist.

3.1 Modelling

Figure 1 shows the **pattern of composition**, one of the two major "best practices" presented in this paper (together with the "5C's"). The pattern forces developers to consider *any*

composite entity as consisting of following entities:[2]

- *Functional Entities* (Computations) deliver the functional, algorithmic part of a system, that is, the *continuous time and space behavior*. A `Functional Entity` can be a composite entity in itself, following the same pattern of composition. Section 4 elaborates on (Composite) `Functional Entities`.
- A *Coordinator* to select the *discrete behavior* of the entities within its own level of composition, that is, to determine which continuous behavior each of the Functional Entities in the composite must have at each moment in time. Section 6 elaborates on `Coordinators`.
- A *Scheduler* handles the *order of execution of the Functional Entities (computations)* within the entity (including access to shared data), required for correct overall behavior of the composite. Section 6 elaborates on `Schedulers`.
- *Functional data Communication* handles the *data exchange behavior* between Functional Entities, elaborated in Section 7. Note that data communication is, in general *bi-directional*, in contrast to the popular mainstream "publish-subscribe" tradition.
- *Event data Communication* handles communication between all entities and the `Coordinator`, elaborated in Section 7.
- A *Monitor* compares the *actually* received and sent out data with the *expected* data, and fires events depending on a configurable set of constraint conditions that must be monitored for a robust execution of the composite. Section 4 elaborates on `Monitors`.
- A *Configurator* configures the entities within a level of composition. Section 5 elaborates on `Configurators`.
- A *Composer* constructs a composition by grouping and connecting entities. This section further elaborates on `Composers`.

The composition pattern is recursively applicable (as suggested by Fig. 1), with each `Functional Entity` in each hierarchical level following the same composition structure. This gives the possibility of creating a hierarchy of large numbers of composite entities, without having to learn any new architectural design primitives, or adapt one's design trade-off insights. In other words, the authors' "best practice" suggests to use this composite pattern as the *smallest architectural building block*, which is in strong contrast to the more mainstream belief that the single entities (or "component") themselves are *the* most appropriate system primitives for composition or reuse. The impact of this difference on overall system architecture can not be overestimated, and hence it is a very important point for discussion and/or review. Again, this "best practice" has grown out, step by step, from the above-mentioned large body of software systems that have been built by the authors' research group, in isolation or in close cooperation with international partners. That means that the role of *each* of the parts in the pattern is motivated by several concrete use cases, in a multitude of application scenarios and contexts.

One successful, independently created instance of (a large part of) this composition pattern is realised in the *RObot Construction Kit* (ROCK) [30]. It was the first publicly available software project to introduce what this paper calls the `Composer`, as a necessary entity within any composite. Its role is to group and connect all other entities, on the basis of a *model* of the architecture. Its first responsibility is the deployment of the entities within a Composite Functional Entity, when the system is brought alive for the first time. However, the `Composer` is *active throughout the whole lifetime* of a Composite Functional Entity, and responsible for *run-time* changes in the system architecture. A `Composer` as an entity in its own right allows the `Coordinator` to trigger a (re-)composition of the Composite Functional Entity or a gradual composition, intermittent with configuration steps for the composed entities. This acknowledges the `Composer` as a real "activity" and not a static data structure.

The interaction between `Composer` and `Coordinator` follows a **`Coordinator-Composer` pattern**, a specialisation of the `Coordinator-Configurator` pattern introduced by Klotzbücher et al. [31]. In the `Coordinator-Composer` pattern, a `Composer` holds a set of composition steps. Each composition step has a unique ID and can be implementation or software specific. The `Coordinator` *commands* the composition steps to be executed, the `Composer` *executes* the commanded composition steps, that is, it is configuring the *structural* model of the composition; the `Configurator` in a composite, on the other hand, is changing the *behavior* of the composite but not its structure. Of course, changing the composite's structure most often implies that first a change in the composite's behavior must be realised, in order to bring the composite to a behavior that allows the restructuring.

This `Coordinator` commands in the form of raising events, on which a `Composer` reacts when the event matches a composition step ID. A status event communicates success or failure of the composition step back to the `Coordinator`, allowing a befitting reaction. Section 6 details the interaction between the `Coordinator`, `Composer`, and `Configurator`.

The `Functional Entities` take a special position within the pattern of composition: (i) there can be multiple `Functional Entities` within a composition, and (ii) a `Functional Entity` can be a Composite Functional Entity in itself, following the pattern of composition of Figure 1, resulting in a hierarchy of composites.

`Functional Entities` take this special position since they form the core functionality of a system: without them the other entities have no meaning nor use. Moreover, the other

2. In some cases, it might make sense to eliminate one or more of these entities, but then, at least, the developer has a motivated reason to do so.

entities exist *only because* the behavior and interaction structure of multiple `Functional Entities` need extensive "bookkeeping" support.

The `Functional Entities` described in Section 4 are grouped in a hierarchy of composites, further referred to as *levels of composition*. A *higher* level of composition, the parent, consist of a composition of *lower* level components, the children. Section 4.2 elaborates on these levels of hierarchy.

The presented hierarchy of composition has the semantic content of a **boundary of knowledge**. The entities within a Composite Functional Entity only know about the presence of the other entities within that composite. This does not hold for the `Functional Entities`: each of them *should not know* about any of the other entities in the composite, since everything that has to be known is already covered in the other entities. Hence, the `Functional Entities` *broadcast* their data and events, not having to know who will react or use them. It is the authors' belief that this pattern represents the most strict decoupling between entities that still results in a manageable and comprehensive entity, composite and system design.

Figure 3 gives an example of this concept applied to an example `Task` in the context of the *iTaSC* framework. The `Coordinator` raises an `e_CC_PID_connect` event in its *ConnectEntities* sub-state. The `Composer` reacts to this event by creating, amongst others, a connection between the *Chif* ports of the `Functional Entities` *VKC_Cartesian* and *CC_PID*. Sections 4, 5, and 6 will further elaborate on this example.

3.2 Implementation

Composition as a concept *composes* entities of all the other 5C concerns; details of the latter will be given in their respective Sections.

The current implementation of `Composer` is a Lua [32] script using the RTT-Lua extension libraries [33]. The `Composer` scripts are loaded in an Orocos-Lua component [33]. An Orocos-Lua component provides a Lua based execution environment for constructing real-time safe robotic domain specific languages. It gives the features of an Orocos component, such as Communication and Configuration infrastructure (ports, property marshalling) to a `Composer`.

The RTT-Lua extension libraries provide the software framework specific information to create and connect entities, in this case *deploying Orocos components* and *connecting Orocos ports*. As will be elaborated in dedicated sections, all entities will be deployed in an Orocos component.

The implementation provides a boiler plate script for the `Composer` for the default compositions made in iTaSC (see Section 4.2), and this is possible because of the very fixed structural model to which the involved implementations of the entities conform. Future work will create a Domain Specific Language for the `Composer` in line with the `Coordinator` DSL [31], Figure 3 hints at such an implementation.

The reference iTaSC framework implementation groups code related to a Composite Functional Entity in a ROS package. For example such package contains the C++ code for the `Functional Entities` and `Monitors`, rFSM/RTT-Lua Lua scripts for the `Coordinators`, `Configurators`, `Composers` and `Schedulers`, XML property files for the `Configurators` and references (e.g. ROS dependencies) to leaf composite entities.

3.3 Discussion and lessons learned

In a first implementation the `Configurator`, `Coordinator` and `Composer` were loaded in a single Orocos-Lua component. The advantage of this approach was the shared activity (thread) and memory, reducing the need for event communication; also the human factor was important: at that time, we worked in a context where typically one single developer was responsible for most phases in the development process, so it was the easiest solution for this single developer to put all configurations, deployments and coordinations into one single file.

This simple approach turned out to have severe disadvantages in the longer term: the sharing of activity and memory implicitly also causes the coupling of these entities, because the blocking of an operation in a `Coordinator` or `Composer`, causes the thread to block, leaving possible identification and reaction only to the higher level `Coordinator`. The latter typically can do no more than identify that the whole Composite Functional Entity has stalled. The current separation of the entities as single Orocos components conforms better to the separation of concerns advocated in this paper.

Another "lesson learned" in this context was about the human factor: large configuration files make it *extremely difficult* for new developers (i) to understand the whole file, and, hence, (ii) to be confident that they understand the implications of whatever small change they would like to make to the configuration file. In practice, this had led to very poor reuse of existing code, and even too close to zero incremental improvement of the existing code.

From the "component" framework point of view, we learned that it is impossible to create something like a generic default script for a `Composer`, since Orocos-RTT (or ROS, or any other "component" framework) lacks an explicit, let alone formal, model of components and of how they can be composed. However the above-mentioned *ROCK* project [30], which builds on Orocos-RTT, has made very good steps at bringing in such formal modelling for Orocos components and their composites, via its *Syskit* sub-project.

From the "task specification" framework point of view, none of the framework mentioned in the Introduction provides hierarchy for their *software entities*; many do offer hierarchy for their *task specification* primitives, but this hierarchy has very different purposes. A task specification (in a model-driven engineering context) is a formal description (model) of what

the robot system should do. Hierarchy has been introduced in that context since basically the beginnings (early 80s), in the form of more or less detail in the task description; for example the task of navigating from Room_A to Room_B in a building is hierarchically decomposed into the subtasks of navigating (i) within Room_A from the robot's current position to the door of Room_A with Corridor_1, (ii) through Corridor_1 to Corridor_2, (iii) inside Corridor_2 to the door of Room_B, (iv) from the door of Room_B to the desired end location inside Room_B. And each of these sub-tasks can be hierarchically decomposed in more fine-grained sub-sub-tasks, such as (i) moving the robot arm to the handle of the door in Room_A, (ii) grasping the door handle, (iii) turning the door handle crank, (iv) turning the door around its hinge, (v) releasing the handle grasp, (vi) moving the arm in a minimal-width configuration, (vii) moving through the door opening into Corridor_1. Etc. The hierarchy described above is 'orthogonal' to the software architecture hierarchy which is the focus of this paper.

4 COMPUTATION

Computation (a `Functional Entity`) delivers the useful **functionality** of a system, i.e., the algorithmic part of an application. As mentioned above, applications typically involve *many* different `Functional Entities`.

4.1 Modelling

A task specification application, based on constraint-based programming according to the iTaSC methodology, consists of the following `Functional Entities`:

- *Setpoint generators* deliver desired values for the controllers of a `Constraint-Based Program`. `Setpoint Generators` can provide *fixed values*, but also more *complex data structures*, or even full trajectory generating or planning *functions*.
- *Sensors* deliver *feature measurements* derived from raw sensor data, e.g., distance information, force-torque data, or point clouds.
- *Robots and Objects* calculate the state of robots and objects involved in an application based on their kinematic and dynamic models. `Robots` have controllable degrees-of-freedom (DOF), whose state is denoted with coordinates $\boldsymbol{q}$. Unlike `Robots, Objects` have no controllable DOF; their models comprise definitions of *object frames* as reference frames for state calculations such as the pose or twist between two object frames. Computations by `Robots` and `Objects` include forward and inverse kinematics and dynamics solvers, as implemented by for example the Orocos KDL library [34].
- *Drivers* deliver hardware interfaces for `Robots` and `Objects`, communicating proprioceptive information, desired low-level controller setpoints, and sensor or estimator information. Examples include the Kuka FRI interface [35] or an interface to a controller provided by the pr2_controller_manager on a PR2 robot [36].
- A *Scene* (or *World Model*) keeps track of the position of the robots and objects in the world, and between which object frames tasks are defined. It transforms data to be composable conforming to geometric semantics [37], [38], e.g., common reference frame and point, as well as object and reference object on which these are defined for the sum of poses.
- A *Solver* calculates the desired values for the low-level robot controllers as the result of the *constrained optimization problem* that results from the methodological composition of task constraints and objective functions. Examples include mathematical optimization algorithms such as frequently used weighted-damped least-squares, or more complex algorithms provided by general-purpose numerical solver toolkits, e.g., ACADO [39].
- A *Virtual Kinematic Chain (VKC)* calculates the state of the task space or feature space defined between object frames of the robots and objects. It uses a kinematic model of this task space using the auxiliary feature frames. In its explicit form it can be regarded as a virtual kinematic chain which state is represented by the *feature coordinates* $\boldsymbol{\chi_f}$ ("*Chi-f*"). Computations by `VKCs` include forward and inverse kinematics and dynamics solvers.
- A *Constraint-Output (CO)* calculates the output equation $\boldsymbol{y} = f(\boldsymbol{\chi_f}, \boldsymbol{q})$. The output can serve as input for controllers, estimators, monitors etc.
- A *Constraint-Controller (CC)* calculates the control law that enforces a desired setpoint on an output, resulting in the desired output in task space, e.g. $\dot{\boldsymbol{y}}_d^\circ$ for the velocity resolved case. Examples of `Constraint-Controllers` include the commonly used PID controller or impedance controllers.
- An *Estimator* observes or estimates the (internal) state of a system, based on a model, and the input and output of the system under observation. `Estimators` are commonly referred to as *state observers* in control theory, or an implementation of *adaptation* in computer science.

The *Composition Pattern* discussed in Section 3 introduces the `Monitor` as an essential, special Functional Entity. It compares the actual data flow between the `Functional Entities` to the actual one, and raises an event when a configured set of conditions is met. For example, the `Monitor` on an `Estimator` that outputs the uncertainty on an estimated parameter, can raise an event to indicate that the uncertainty has risen above a maximum value; the composite's `Coordinator` can then react to that event by, for example. slowing down the current movement of the robot. (Event processing is discussed in detail in Section 6.)

Figure 3 shows an example of the interactions of a `Monitor` of a `Task` Composite

Functional Entity. The Coordinator raises an e_monitor_max_position_error event that triggers the Monitor to monitor the position error. The Monitor has a connection to the data flow of the Functional Entity *CC_PID* that outputs this error. The Monitor raises the e_max_pos_tracking_error_exc event to indicate that the maximum allowed position tracking error has exceeded.

The *separation* of Functional Entities from their Monitors *decouples* Functional Entities and application specific monitoring conditions, resulting in higher reusability. Obviously, Functional Entities should also raise additional events themselves, based on *internal* monitoring conditions, such as the completion of a certain algorithm or the reaching of a maximum number of iterations.

4.2 Composition of Functional Entities

The following paragraphs describe the *levels of composition* for the use case of constraint-based optimization. We restrict ourselves to three levels of composition, *Application*, *Constraint-based program* and *Task*. Higher or lower level composites are definitely possible, for example the higher level of a *Mission* that incorporates multiple applications, deployed simultaneously or serially on multiple robots. At each of the three levels we focus on, we regard the most important example of a composite, which also gives its name to the level. This does not limit the other Functional Entities to be composites following the pattern of composition. For example a Sensor can be a composite of a Driver, a Filter, and other algorithms on the sensor data, possibly divided over multiple levels of composition. Or a Constraint-Controller can consist of a composition of atomic controllers for each of the degrees-of-freedom of the output equation. The structure of the Communication, Configuration and Coordination on each level will be discussed in their respectively sections.

- A *Task* forms a first composite, delivering a set of constraints to the optimization problem that are related to the same task space. In case of the explicit formulation of iTaSC, a Task composes the Virtual Kinematic Chain (VKC), a Constraint-Controller (CC), the Constraint-Output (CO), and a Setpoint Generator as shown in Figure 2. This composite contains all functionality needed to define and execute a task. This Task is however agnostic of its concrete role in the whole application. For example, it is unaware of the role or context of the object frames between which it is acting. That semantic meaning is (or rather, should...) be given by the parent composite. The current discussion limits itself to one entity of each type, however multiple entities can be present to be able to switch between Constraint-Controllers or Setpoint Generators within one Task, or entities can be brought out of the Task composite. This discussion is beyond the scope of the current document.
- A *Constraint-based Program* forms a second composite, delivering task specification and control on a set of involved robots and objects. This Composite Functional Entity composes all elements related to the *Scene graph* and how it is used to generate setpoints for the low-level (motor) commands. It composes ("couples") the Robots and Objects in a Scene, constrained and linked by Tasks which encompasses the task space formulation and resolved by a Solver.
- The *Application* forms our third composite, composing ("coupling") the Constraint-Based Program with the application-specific "hardware" (Drivers and Sensors), as shown in Figure 2. Separating the hardware from the program allows developers to reuse the same Constraint-Based Program in simulation or on the real robot by just changing the Driver and Sensors, and offers flexibility with respect to the hardware used (multi-vendor).

As mentioned in Section 3, a composition forms a boundary of knowledge. The following example explains this concept; the Configurator named *iTaSC_Configurator* of a Constraint-Based Program named *iTaSC* needs to know which Tasks to configure, and the Coordinator named *iTaSC_Coordinator* needs to know which Tasks to expect events from. The Tasks however present their data on a data flow port, not knowing who is using the data. It is the composition of *iTaSC* that determines who is listening and reacting. For example the Monitor named *iTaSC_Monitor* that monitors the data of a specific Task named *ApproachObject*. In addition to the *(functional) data*, the *ApproachObject* also broadcasts *events*, and it is the *iTaSC_Coordinator* that expects and reacts on events from the *ApproachObject*.

4.3 Implementation

The model provided above is implementable with various software component frameworks or their combination, such as OpenRTM [41], [42], Orca [43], GenoM [44] or ROS [45]. Strictly speaking, the Composition Pattern requires only the following primitives to be provided by software frameworks: Component, Port, DataFlow, Event, FiniteStateMachine. All these primitives are provided by many frameworks, but no framework provides them all; except for ROS or OpenRTM, when the definition of *framework* is taken in the broader sense of *original framework and the ecosystem that grew around it*. However, it is not at all necessary that an implementation of the Composition Pattern has to be realised in one single framework; on the contrary, the 'best' implementation will most often consist of a selection of features from different frameworks. Of course, 'best' is an application-specific objective function, and sometimes 'real-time performance' will be part of that objective function (making the Orocos framework more appropriate than ROS, for example), while another application gives less weight to real-time performance than to the desire to reuse already existing ROS node implementations,

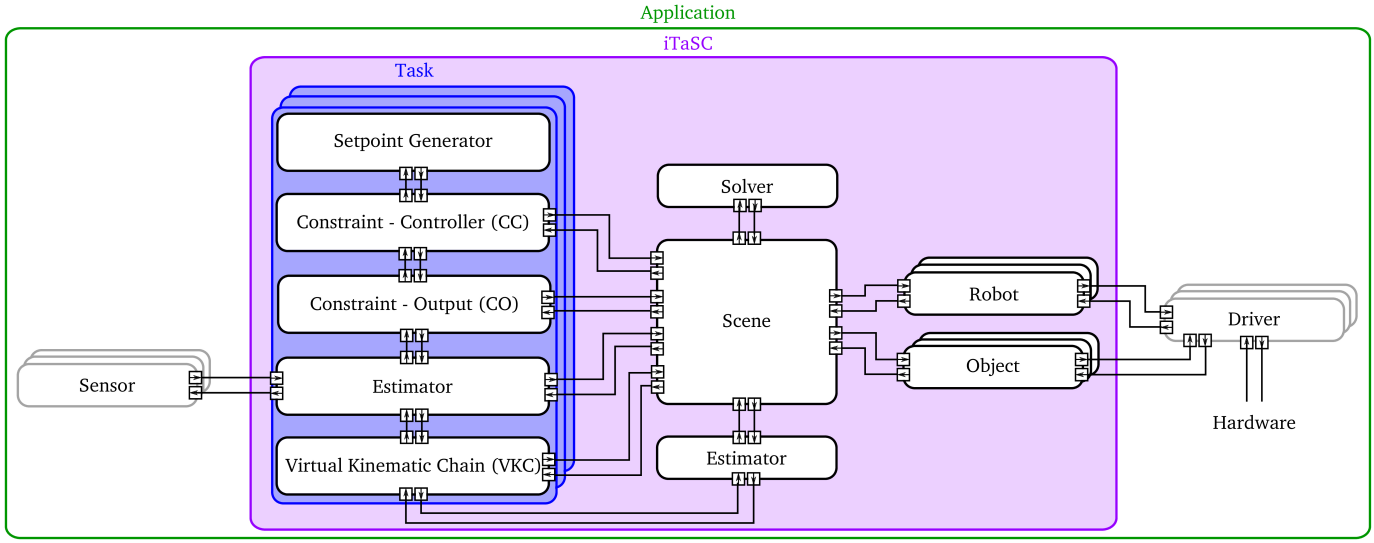


Fig. 2: Detail of the composition of computation (Functional Entities) for the explicit formulation of iTaSC using sysML flow ports [40]. The composition levels are the *Application* context, the *iTaSC* program, and the *Task* specification. Stacked boxes refer to the possibility of having multiple entities of a specific type.

Our reference implementation provides two different approaches to implement `Functional Entities`. The first and most often used approach is applied throughout the core of the implementation and uses the Orocos component framework for real-time control [26], [46], [47] to provide an infrastructure for `Functional Entities`.

The model of an Orocos component has three primitives: Operation, Property, and Data Port. That means that in the (semantically rather restricted) context of component frameworks, it is the component that provides the basic unit of computational functionality. Data needed for calculations and the resulting data of a component's calculations is communicated using Data Ports.

The advantages of the Orocos components as `Functional Entities` include: 1) the real-time capabilities, 2) thread-safe time determinism, 3) lock free inter-component communication in a single process, 4) synchronous and asynchronous communication possibilities, 5) reflection capabilities and interfaces to other frameworks such as ROS. Their disadvantage is that most developers (implicitly and incorrectly!) assume that each component has to be deployed in its own operating system process, but this policy of composition introduces many *context switches*, most of which are functionally superfluous.

The Orocos component framework does not *explicitly* provide composite components. However, since the software patterns presented in this paper offer *composition by infrastructure* (Section 3), this lack of explicit Orocos composite components is not a fundamental problem to the formalization of `Functional Entities` as composite entities.

In order to ensure modularity and reusability of the components as instances of `Functional Entities`, they have to provide a well defined Data Port interface: what data should be communicated and in which form. (Section 7 elaborates on the communication aspects of these issues.) Therefore the reference implementation offers a template component for each type of `Functional Entity`, in the form of a C++ class. More specialised components inherit from this template.

For example a `PID` or `impedance_control` component inherits from the `Constraint-Controller` template, implementing a PID controller and impedance controller respectively. Both components are however still *general* in the sense that their behavior will depend on

- their composition and communication that determines who delivers setpoints and state information,
- their configuration that determines which gains to use,
- their coordination that determines when they are active.

A component can also serve as an interface to other parts of software or hardware, for example a `Driver` that interfaces with a KUKA robot over an FRI connection [35].

In addition to Orocos components that inherit from a template, the reference implementation provides a second, more general way of introducing `Functional Entities` by adding meta-data to an implementation of a `Functional Entity`. This meta-data models the interface of the `Functional Entity` and contains the necessary information for other entities to interact with the entity. Listing 1 gives an example of meta-data of the Data Ports of a `Functional Entity` using the *Lua* language [32]. It contains an entry in the Lua table for each Data Port by its name with following tags:

- type: the type of the port that defines what general kind of information the port delivers or requests,

- rtt_type: the type of the data specific to different platforms, in this case Orocos RTT,
- semantics: the (geometric) semantics meaning of data, the importance which will discussed in section 7,
- id: detailed identification of the port, this could refer to for example ROS topic information,
- direction: the direction of the Data Flow with respect to the entity,
- fw: framework in which the entity is implemented, which will define how to interpret the other tags such as the id.

The reference implementation uses this meta-data approach for example to provide a `Driver` for the KUKA Youbot using the existing open-source ROS nodes provided by *youbot_description*. [48].

Listing 1: Example of meta-data

```
ports={
  my_port_a={rtt_type='/motion_control_msgs/↩
      JointVelocities',
    type='driver',
    semantics='JointVelocitySemantics(ee,base)',
    id='/JointVelocitiesCommand',
    direction='input',
    fw='ros'}}
```

Our reference implementation implements `Monitors` for example using the service plugin feature of Orocos. It allows pluging in extra functionality in an existing Orocos component. Future work will formulate a DSL for `Monitors`, as hinted at in the `Monitor` entity shown in Figure 3.

4.4 Discussion and lessons learned

Majority of Functional Entities (computations) in the current implementation are encapsulated in the Orocos components, which mirrors the proposed `Functional Entity` model. However, this structure of different components with (inter-process) communication is also very rigid with respect to optimization of the computational efficiency. Nevertheless, models can be deployed in different ways. For example, certain Functional Entities could be grouped at run-time, reducing communication needs. An example of such composition can be found in the GenoM project, that makes use of codels [44] as the smallest unit of execution that can be easily composed to larger Functional Entities without inter-process communication, for example using shared memory.

The approach to attach a model to an implementation gives more versatility. The current implementation gives only a limited example of such an approach.

5 Configuration

Configuration influences the behavior of entities of the other concerns by changing its **settings**. Examples include control gains and communication buffer sizes.

5.1 Modelling

Configuration is enforced by a `Configurator` entity, separating it from coordination by the `Coordinator-Configurator` Pattern [31]. A `Configurator` holds a set of configurations.

A configuration consists of a set of parameters of another entity that are exposed to be configurable. It has a unique name and can be implementation-, hard- or software specific.

The `Coordinator` *commands* the configurations to be loaded in an entity, the `Configurator` *executes* the commanded configuration. A `Configurator` applies a configuration with a certain name when receiving an event from the `Coordinator` with a matching ID. A status event communicates success or failure of the configuration action back to the `Coordinator`, allowing a befitting reaction.

Figure 3 gives an example of the `Coordinator-Configurator` interaction for an example `Task` Composite Functional Entity. The `Coordinator` commands a high tracking accuracy of a controller by raising an event *e_high_accuracy_control*, on which the `Configurator` reacts with adapting the gains of the `Functional Entity` *CC_PID* to a preset value.

Another example is the configuration of a `Monitor`, as also shown in figure 3. The `Configurator` configures the `Monitor` with the concrete error level to react on, and the resulting events to raise, in this example *e_max_pos_tracking_error_exc*.

The `Configurator` needs to be configured itself which seems a contradiction at first glance. It is however the hierarchy provided by the composition that allows the configuration of the `Configurator`: The `Configurator` of the level of composition higher will configure the `Functional Entity` to which this `Configurator` belongs to. This configuration includes the configuration of this `Configurator`. For example the `Configurator` *iTaSC_Configurator* of a `Constraint-Based Program` *iTaSC* configures a `Task` *ApproachObject*, hence configuring its `Configurator` *ApproachObject_Configurator*. A bootstrap ensures the configuration of the `Configurator` of the highest level Composite Functional Entity. Section 6 details the bootstrap to bring up the system.

5.2 Implementation

Since the reference implementation mainly uses Orocos, its Property infrastructure is used for configuration. Orocos Properties [47] provide an interface to adapt at run-time parameters that are made publicly available. Services to read and write these properties to XML, RTT-Lua or other formats are available for the Orocos platform. Configuration specified in the iTaSC DSL or deduced from it can be set accordingly.

The `Configurator` implementation uses the reference implementation of the Coordinator-Configurator pattern in Lua. Its extension with the RTT-Lua libraries provides the

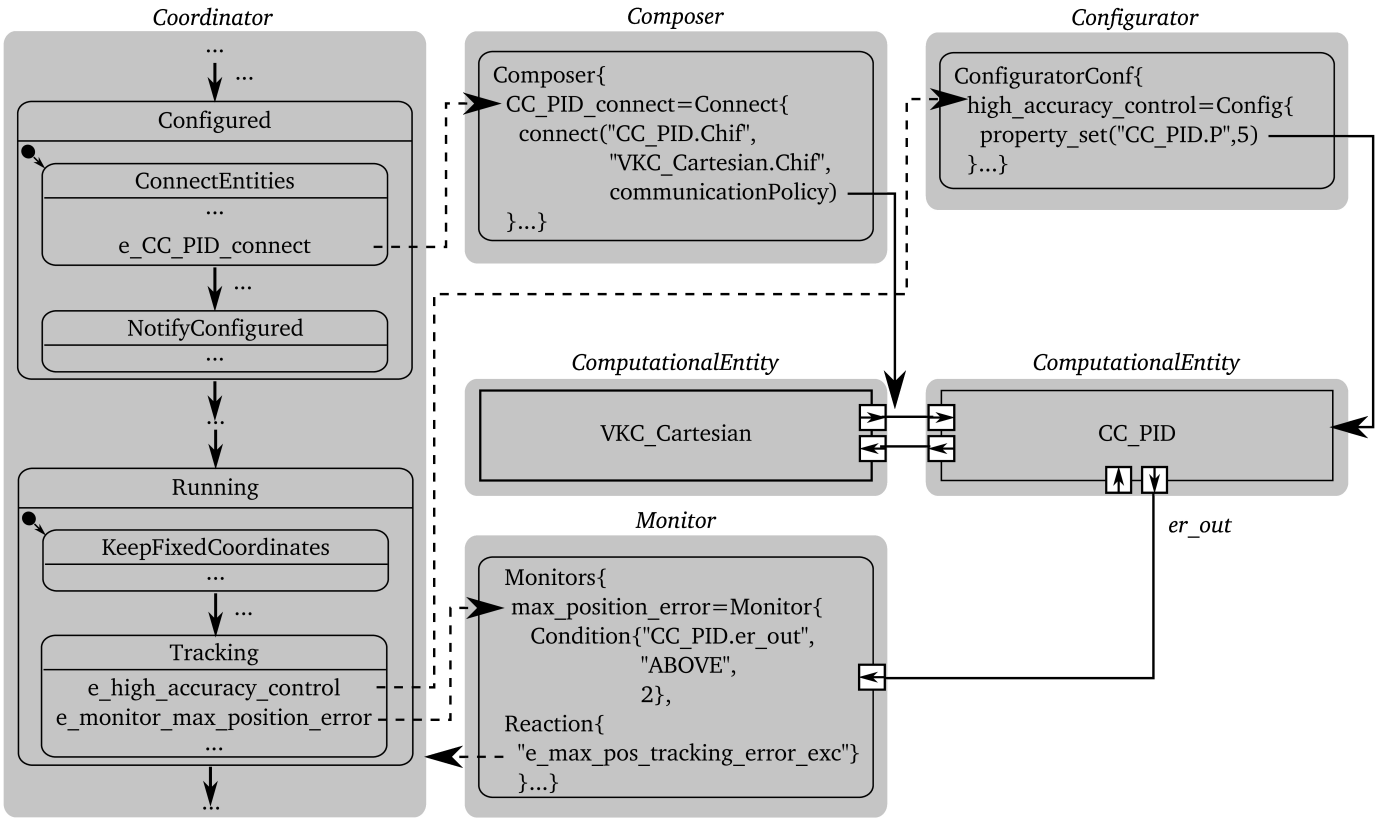


Fig. 3: Example of the interaction of the `Coordinator`, the `Composer`, the `Configurator`, and `Functional Entities` of an example `Task` Composite Functional Entity. Dashed arrows indicate how events trigger actions, black arrows indicate how entities act on other entities. Only the parts relevant for the example are shown, the `Scheduler` and other `Functional Entities` are left out. Three dots indicate left out parts within an entity.

software framework specific information to configure Orocos components.

As for the `Composer`, the implementation provides a boiler plate script for the `Configurator` for the default compositions made in iTaSC. The configuration of the `Configurator` can load different sets of configuration. For example the configuration of a `Configurator` of a `Constraint-Controller` comprises the loading of the gains stated in the iTaSC DSL model (the configuration) into the `Configurator`, which applies the correct set of gains on the instance of the `Constraint-Controller` on receiving a command from the `Coordinator`.

5.3 Discussion and lessons learned

As for the `Composer` detailed in section 3.3, separating the `Configurator` from the `Coordinator`, relieves the `Coordinator` from software platform specific actions and decouples execution and hence failure of `Coordinator` and `Configurator`.

6 Coordination

Coordination determines how the entities of all concerns work together, by selecting in each a certain behavior. It provides the **discrete behavior** of entities and their composites.

6.1 Modelling

Each Composite Functional Entity has one `Coordinator` that interacts with entities of other concerns by events. The model of the `Coordinator` is a rFSM statechart, introduced by Klotzbücher and Bruyninckx [49]. Statecharts have the advantage to be composable, moreover rFSM statecharts are able to satisfy real-time constraints. The extended version of rFSM includes event memory, the use of which will be explained further on.

The model of the `Coordinator` follows the best practice of *pure coordination* [49]. Pure `Coordinators` are *event processors*, that have determining state based on events and sending out events as only functionality. Pure coordination avoids dependencies on platform specific actions, and avoids blocking invocations of operations. The events originate from the other entities of a Composite Functional Entity or from a `Coordinator` of a parent or leaf entity.

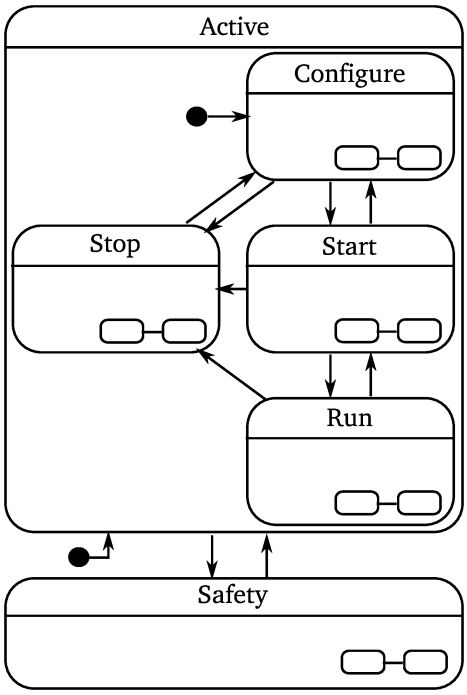


Fig. 4: Life-cycle coordination pattern. The Active state consists of a Configure, Start, Run, and Stop state. The Safety state next to the Active state allows transition to this Safety state at highest priority. Each state can be a state machine of its own indicated with the two connected ovals in the right corner of a state. Figure 5 gives an example of the sub-states. The arrows indicate a state transition which is triggered by an event, the filled black circle indicates an initial connector.

A `Coordinator` conforms to the **life-cycle FSM** of a Composite Functional Entity, as represented in figure 4. Each of the states of the life-cycle FSM can be a state machine on its own, hence a `Coordinator` is a *hierarchical FSM*. The following states make part of the life-cycle FSM:

- The *Active state* which consists of the Configure, Start, Run and Stop state. This state is the initial state when a `Coordinator` is brought up, indicated by the initial connector in figure 4.
- The *Configure state* coordinates the composition and configuration of the Composite Functional Entity. In the Configure state the `Coordinator` triggers the `Composer` and the `Configurator`. The `Composer` composes the entities of the Composite Functional Entity by creating entities (deployment) and connecting communication channels between entities, as explained in section 3 and 7. The `Configurator` loads and executes the configuration of all entities of the Composite Functional Entity, following the `Coordinator-Configurator` pattern [31] as explained in section 5. The `Coordinator` triggers the `Composer` and the `Configurator` intermittent, since some steps of the composition need prior configuration. For example the creation of communication ports of the `Scene` dependent of the number of `Tasks` (configuration step), which can only be connected after their creation (composition step). This sub-state is the initial sub-state when a `Coordinator` is brought up, indicated by the initial connector in figure 4.
- The *Start state* coordinates the preparation of the entities of the Composite Functional Entity for nominal operation. In the Start state the `Coordinator` triggers the `Scheduler` to initialize, and Functional Entities to start computation and data exchange.
- The *Run state* is the state of nominal operation of the Composite Functional Entity. On the one hand, the `Coordinator` triggers when entering this state the activation of the `Scheduler`. On the other hand, it influences the run-time behavior of the Composite Functional Entity. This run-time behavior consists of altering the active set and configuration of `Functional Entities`, based on incoming events fired by for example the `Monitor`. For example the configuration of the `Constraint-Based Program` consists of amongst others, the set of active tasks, the involved objects and (parts of) robots, and the task weights and priorities.
- The *Stop state* coordinates the termination and destruction of the entities of a Composite Functional Entity. In this state, the `Coordinator` triggers the `Configurator` to do the 'opposite' of the actions during the Configure state.
- The *Safety state* brings the composite state in a safe mode, which does not necessarily correspond with the Stop state. Since the Safety state is located on a higher level in the state machine hierarchy, events triggering a transition to the Safety state will have always priority, independent of the current sub-state within the Active state. For example blocking the motors of a robot in an application in which the robot has to handle dangerous materials, or on the contrary, bringing the robot to gravity compensation mode when working close to humans. As the initial connector indicates, recovering from a Safety state requires a reconfiguration.

The different `Coordinators` over the different composition levels interact by events, forming a *hierarchy of concurrently executed (hierarchical) FSMs* in which the higher level `Coordinator` coordinates the lower level `Coordinators`. Remark that not all `Coordinators` need to be in the same state, for example when a new `Task` is added to an existing `Constraint-Based Program` in the Run state and needs to go through the life-cycle until the Run state.

The life-cycle FSM takes part in the deployment of the system. A bootstrap brings up the highest level `Composer` that deploys the `Coordinator` and communication between them. Further this bootstrap ensures the configuration of the highest level `Configurator`. The `Coordinator` brings

up the remainder of the Composite Functional Entity by a coordination of a series of composition and configuration steps, using the `Coordinator-Configurator` [31] and `Coordinator-Composer` pattern, explained in section 3 and 5.

The advantages of this approach of deployment are (i) the systematic approach to bring up a system, (ii) the reduction of the actual phase of bringing up the system to a 'minimal' bootstrap, by using the structure of the composition, (iii) the predictability of the deployment procedure and its possible errors.

In addition to the `Coordinator`, the `Scheduler` forms part of the coordination. The `Scheduler` handles the order of the computations by the `Functional Entities`. However it forms not part of the `Coordinator`, since (i) a `Scheduler` uses service calls or events, therefore it is not a pure event processor, (ii) a `Scheduler` forms a periodic (time-triggered) process with respect to the (mostly) aperiodically, event triggered behavior of the `Coordinator`, (iii) a `Scheduler` depends on the implementation of the `Functional Entities`, (iv) a `Scheduler` must be fast and efficient, and can therefore be optimized using specialized routines. `Scheduler` and `Coordinator` are separately triggered by their counterpart at a higher level of composition, hence the `Schedulers` of each Composite Functional Entity form also a *hierarchy of concurrently executed (hierarchical) FSMs*. This separation avoids coupling of the timing of `Scheduler` and `Coordinator`, that could cause delays in the scheduling. For example the `Scheduler` *iTaSC_scheduler* of a `Constraint-Based Program` *iTaSC* triggers a `Functional Entity` *Task* that itself is a Composite Functional Entity, this *Task* should immediately execute the algorithm, hence trigger its own `Scheduler` *Task_scheduler* and not wait for its own `Coordinator` *Task_coordinator* to command the *Task_scheduler* to do so. This avoids the situations where 1) the *Task_coordinator* has to react on both an event causing a behavior change and the trigger from the *iTaSC_scheduler*, 2) and the situation where the *Task_coordinator* only reacts on the trigger from the *iTaSC_scheduler* in a next timestep (section 6.2).

6.1.1 Concrete model of the life-cycle FSM

Figure 5 shows the details of the sub-states of the Active state of the life-cycle FSM. The grey boxes on figure 5 indicate these sub-states: Configure, Start, Run and Stop. They have each two sub-states: one with a name ending on *-ing* and one with a name ending on *-ed*, with exception of the Run state which has a PreRunning and a Running sub-state for linguistical reasons.

When in a *-ing* state, composite entities are coordinated, before triggering the `Coordinators` of its child entities. When in a *-ed* state, composite entities are coordinated after the `Coordinators` of the child entities are triggered but

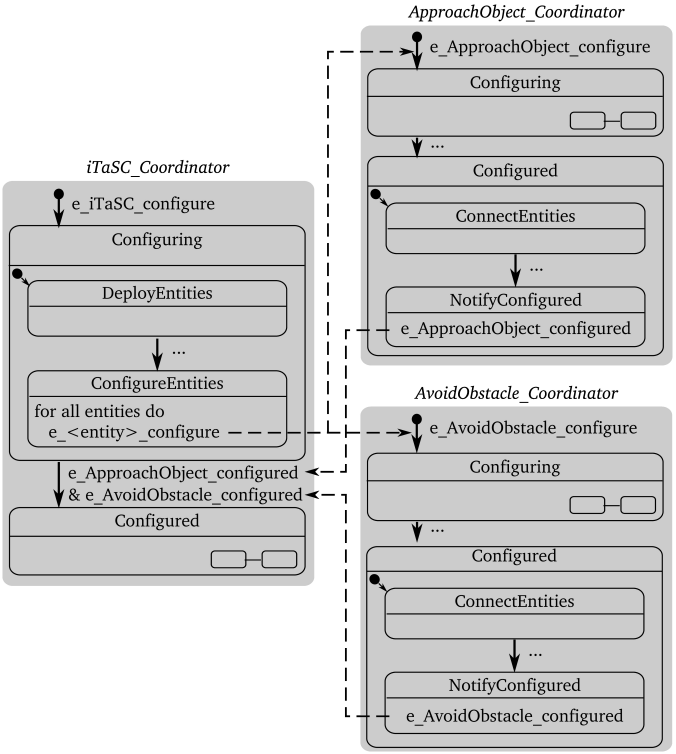


Fig. 6: Example of the interaction between `Coordinators` at different levels of composition. The dashed arrows indicate how the raised event triggers a transition.

before the parent `Coordinators` are notified with an event. The Composite Functional Entity will be further on referred to as the composite.

A parent `Coordinator` triggers the transition to an *-ing* state, hence the name of the composite that the `Coordinator` belongs to is in the event name. A `Coordinator` transitions from an *-ing* state to an *-ed* state when triggered by events from the child `Coordinators`. Due to this hierarchy the Active sub-states consist of exactly two states.

For example within the Configure state of a `Constraint-Based Program` *iTaSC* that has two composite child entities: the `Tasks` *ApproachObject* and *AvoidObstacle*, as shown in figure 6. The events `e_ApproachObject_configured` and `e_ApproachObject_configured` raised by their respectively `Tasks` trigger the transition from the Configuring state of the `Constraint-Based Program` *iTaSC* to its Configured state.

Another example is shown in figure 3 and was detailed in previous sections.

Transitions that require events from multiple child `Coordinators` require the event memory extention of rFSM in order to avoid synchronization problems. An event is in the rFSM model an edge triggered event that lives only at that

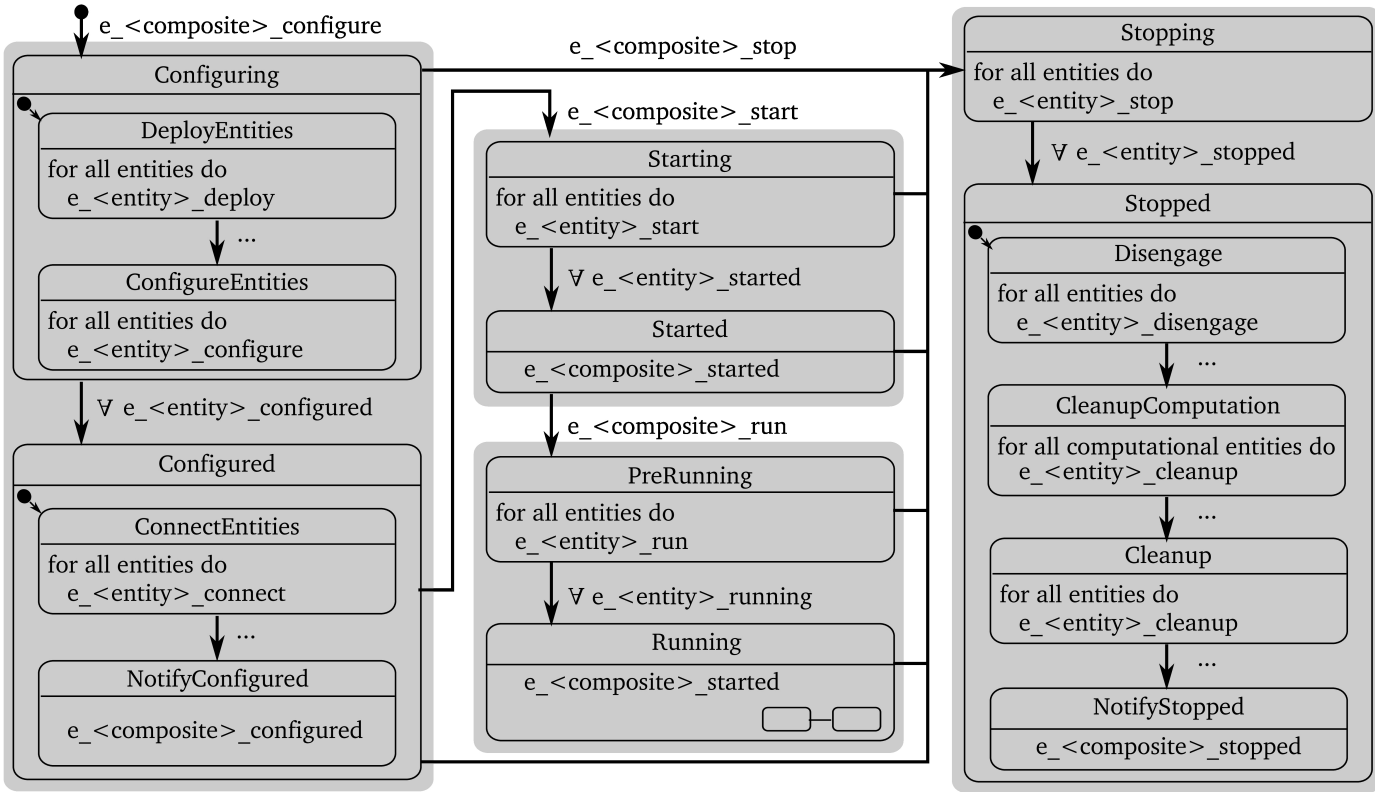


Fig. 5: Detail of the Active state of the life-cycle FSM with example events. The arrows indicate a state transition which is triggered by an event, the filled black circle indicates an initial connector. Names starting with 'e_' denote events. Events next to arrows indicate the events that a state is waiting for to make that transition, events within a state indicate the events sent out by the state. The ∀ symbol denotes that all events of that type need to be raised to make that transition. $< entity >$ denotes a name of an entity within the composite, $< composite >$ denotes the name of the Composite Functional Entity this `Coordinator` belongs to. The grey background denotes the sub-states of the Active state as shown in figure 4. Events that trigger the lowest level transitions are replaced by ...for readability. Also returning transitions such as from PreRunning to Started are left out for readability.

time instant. The event memory extension registers all events that could trigger a transition from the current state, starting from the moment the state was entered. In other words the event memory is cleared with every new state that is entered.

The following paragraphs give an overview of the function of each sub-state:

- The Configuring state consists of two sub-states: DeployEntities and ConfigureEntities. The first triggers the `Composer` to create the entities within the composite. The `Composer` will also enable event flow between the entities. The creation of child composite entities consists of the creation of its `Coordinator`, `Configurator`, and `Composer`, similar to the execution of the bootstrap to bring up the root Composite Functional Entity as mentioned in 6.1. A status event from the `Composer` triggers the transition to the second sub-state. The ConfigureEntities sub-state triggers the `Configurators` to configure the entities within the composite and the `Coordinators` of child composite entities to transition to their Configuring state.
- The Configured state consists also of two sub-states: ConnectEntities and NotifyConfigured. The first connects the data flow between the entities of the composite. This connection is made after the configuration of the child composite entities, since connections can be configuration dependent. The NotifyConfigured sub-state notifies the completion of the configuration step to the `Coordinator` of the parent Composite Functional Entity.
- The Starting state triggers the `Scheduler` to initialize, `Functional Entities` to start computation and data exchange, and triggers the `Coordinators` of child composite entities to transition to their Starting state.
- The Started state has as only function the notification to the `Coordinator` of the parent Composite Functional Entity.
- The PreRunning state triggers the `Coordinators` of the child composite entities to transition to the Pre-

Running state. It further triggers the activation of the `Scheduler`.

- The Running state notifies the `Coordinator` of the parent Composite Functional Entity and coordinates the run time behavior as explained in section 6.1.
- The Stopping state has as only function the triggering of the `Coordinators` of child composite entities to go to the Stopping state.
- The Stopped state consists of four sub-states: Disengage, CleanupComputation, Cleanup, and NotifyStopped. The disengage sub-state triggers shutdown procedures, for example locking robot axes. The cleanup phase consists of two steps: CleanupComputation and Cleanup. The CleanupComputation state triggers the destruction of `Functional Entities`, including child composite entities. The Cleanup state triggers the destruction of the other entities within a composite. This distinction of two states allows the `Coordinator` to react on problems when destroying the `Functional Entities` for which it needs the other entities within a composite. In the last sub-state, NotifyStopped, the completion of the stopping is notified to the `Coordinator` of the parent Composite Functional Entity.

The execution of this pattern of coordination requires a model that provides the information of all separate parts and their relations. The iTaSC DSL [29] is an example that provides such a model.

The interaction of `Coordinators` outlined in previous paragraphs, details interaction in case of the existence of a parent Composite Functional Entity to the composite under consideration. The `Coordinator` of the root Composite Functional Entity will transition from a *-ed* to *-ing* state after completion of the latter, not triggered by an event of a parent `Coordinator`.

The same structure applies to all entities, also for example to the `Driver` and its sub-entities that coordinates robot hardware co-operation when composing different hardware.

6.2 Implementation

The iTaSC software framework uses the Lua reference implementation of the rFSM DSL for the `Coordinator`, that conform to the models presented in previous sub-sections. These rFSM models are loaded in an Orocos-Lua component as for the `Composer`, `Configurator` and `Scheduler`, providing Communication and Configuration infrastructure to the entity. This component will be named *Supervisor* further on.

A *Supervisor* exposes the events raised within a `Coordinator` to an Orocos port, this port is connected to the other entities within a composite by the `Composer`. Through other Orocos ports, the *Supervisor* and hence `Coordinator` receives events.

The implementation considers two types of events, related to the state machine progression of the `Coordinator`: 1) *common events*, which are processed at each update of the `Coordinator`, 2) *priority events*, which are processed upon receiving them.

Most events are common events. Priority events are mainly used for 1) timer events, sent out by a periodic *Timer* to the root Composite Functional Entity, 2) events sent out by a `Scheduler` to trigger a `Functional Entity`, or its child `Scheduler` when that `Functional Entity` is a composite, 3) events that signal a fatal error, such as an `e_emergency` event. Hence a `Coordinator` is a hybrid event-triggered and time-triggered system.

The *Timer* triggers the `Scheduler` of the root Composite Functional Entity, which triggers his leaf `Schedulers`, which on their turn trigger their leaf `Schedulers` etc.

The implementation provides a boiler plate script for the life-cycle FSM, which is a general model and allows 'plugging in' the application specific part of the Running sub-state machine. These application specific parts can be developed and saved as separate rFSM models and hence files.

As mentioned in the modelling Section, a `Coordinator` knows the other entities within a composite, this knowledge is provided by the configuration of the `Coordinator`, derived for example from the iTaSC DSL model.

The current implementation provides a basic `Scheduler`, that requests operations on Orocos components in an algorithmic correct order with respect to the iTaSC concept.

6.3 Discussion and lessons learned

As detailed in previous sections, separating the `Configurator` and `Composer` from the `Coordinator`, leaves the `Coordinator` with no software platform specific actions, and is hence reusable with any other framework.

Remark that in the proposed life-cycle FSM a state triggers the execution of 'actions' by other entities. These actions happen when being in a state, while transitions are light weight event based transitions. This forms a difference with the life cycle FSM of Orocos, where the actions, i.e. the execution of configuration etc., happen in between states. The advantages are that 1) the life-cycle FSM can react on errors when executing these actions, 2) a state of the life-cycle FSM can be divided in sub-FSM to coordinate this transition to the level of desired granularity.

7 Communication

Communication relates to the **exchange of data** [50], [4]. Different communication mechanisms are possible, for example data flow, events, and service calls.

7.1 Modelling

The communication follows the commonly used connector design pattern [47], [51] that decouples dataflow between entities by abstracting the locality of the entities. It enforces a

communication protocol. Figure 1 shows the different communication mechanism within a Composite `Functional Entity`. `Functional Entities` exchange *data flow*. `Monitors` monitor this data flow and communicate events. `Coordinators` exchange *events* with all entities of a Composite Functional Entity, as well as with the `Coordinators` of a higher and lower level of composition. `Schedulers` interact with `Functional Entities`, and `Schedulers` of the higher and lower composition levels by *service calls* or events. In addition they exchange events with the `Coordinator`.

7.2 Implementation

The reference implementation uses mainly the Orocos port infrastructure, with connections between them. Orocos provides lock free, thread-safe communication and integrates with ROS topics or middleware such as CORBA. The `Composer` creates these connections.

An important setting for the communication of events is the buffer of the connection. Since multiple (common) events can occur at any time, and `Coordinators` advance when (time-)triggered, multiple events can accumulate between two executions of a `Coordinator`. Moreover an entity has multiple event sources. A buffer must be used to avoid the loss of events. An entity has to empty this buffer when reading from the port receiving events.

A major drawback in communication are the many data types available to represent the same content. Moreover, majority of these data types are general and have no specific semantic meaning. In the reference implementation a tag is provided to all entities that communicate data to specify this semantic meaning.

The `Composer` uses these tags, together with model information from for example an iTaSC DSL model, to automatically resolve connections between entities.

7.3 Discussion and lessons learned

The Composite `Functional Entity` as boundary of knowledge helps to reduce the number of events communicated throughout the levels of composition. Events of entities other than the `Coordinator` or `Scheduler` can be configured to remain within that boundary.

As mentioned are many data types available to represent the same content, and they mostly lack a semantic specification. A promising approach is standardisation of notations and specific models of these semantics. An example is the work by De Laet et al. [37], [38], to standardise semantics for geometric relations. They also provide software support to enhance common data types for geometry with these semantics. The following workflow shows how geometric relation semantics integrates in the presented approach:

- each Port should get a model of the data it makes available;
- that model should be in a standardized semantic format;
- when the `Composer` is making the interconnection between components, it should check whether the semantic model (and meta-model) of both Ports are the same;
- in case both Ports have different implementations of the model, transformation code could be added automatically (if such code is available in the binaries of the system).

The implications on the overall design are: (i) the Communication and `Composer` activities must be made aware of the semantic models, and (ii) they must have access to implementations that support the model checking and transformations. These implications are almost trivial, conceptually speaking, but horrendously huge for the design and implementation of code. Currently, the authors are not aware of one single software project that supports even the simplest form of such semantic awareness.

8 Conclusions

This paper introduces, motivates and illustrates two major "best practices" that resulted from the accumulated experience of dozens of person years of robotic software framework development at the authors' research group. The first "best practice" is that of the 5C's principle of separation of concerns [3], [49]: the **c**ommunication, **c**omputation, **c**oordination, and **c**onfiguration aspects of any software project should be kept fully separated, but ready to be integrated into a **c**omposition architecture. For the latter, we introduce a second "best practice", the Composition Pattern, that has proven to be very helpful as the basic building block in the design of application-specific, complex system architectures. (A third, derived, "best practice" might be the insight that starting a complex system development process with imposing a specific system architecture from the start is a recipe for failure in the long term.)

The paper illustrates the general best practices by means of the recent intensive refactoring of our *iTaSC* software framework, a generalized constraint-based programming approach [1] (Section 2.2), because (i) it was the application in which the authors first encountered the fundamental deficiencies of former design "guidelines", and (ii) task specification, execution and monitoring involves "planning", "sensing", "control", and "world modelling" functionalities, hence it is a primary example of a robotics system. It is also that broad system integration context and challenge that is the major difference between robotics and other software developments for engineering systems.

The reference implementation uses, in itself, two other large-scale software frameworks, *Orocos* [26] and rFSM [49]; all of them are available under open-source licenses, so readers have access to all details about to what extent exactly we have succeeded in realising the documented best practices in the actual code.

Our search for (i) a systematic way of *describing tasks* in *iTaSC*, together with (ii) the *reusability* driver in the *software implementation* of the iTaSC software framework, drove our software development approach strongly towards a *formalization* of our functionalities and software by means of *Domain Specific Languages* (DSLs); the result in the context of *iTaSC* can be seen from Vanthienen et al. [29].

More concretely, we here enclose a critical discussion of the *lessons learned* in the design and application of the presented "best practices":

- Separation of concerns is a mainstream design driver, but is often used in isolation, i.e. 'separation of concerns hence reusable entities'. We learned that *composition is as important as separation*. This is *the* difference between the 4C's of Radestock et al. [50], and the 5C's as used in this paper, which explicitly focuses on (structural) *Composition*.
- We have (mis)led ourselves during more than a decade in believing that *"components"* are the fundamental building blocks for reusability of functionalities in various architectural compositions. Now, the more complex but very structured and motivated *Composition Pattern* of Fig.1 has become the first-class citizen in our system design. Components are still necessary building blocks, but they should not be the *fundamental* building blocks anymore. This is a very important difference, since a component that is *designed* to be part of the Composition Pattern will be different from a component that is designed without that context, since the explicit separation of the `Coordinator`, `Composer`, `Scheduler`, `Configurator`, `Monitor`, `Functional Entity`, and Communication aspects improve the different qualities (the "ilities" such as adaptability, reusability, etc.) of the building blocks. The first four entity types "manage" the last two, keeping the component flexible during usage, hence improving their adaptivity and adaptability. Moreover, this separation distinguishes application specifics (for example concrete controller gains, monitored conditions to switch controllers, or the succession of the control algorithms to use). Only by exception, one or more of the various parts of the Composition Pattern are left out in a concrete design.
- The *modelling* of software has become second nature to us, since thinking about which DSL(s) would be needed to let non-software (but domain) experts exploit our software frameworks, has been proven to be a better driver for more structured coding than any other design paradigm that is taught in modern computer science curricula.
- The emphasis on modelling is only becoming more and more important, the closer robotics moves towards "cognitive" robot systems, because the latter *have* to be able to reason about their own functionalities, structure and behavior. Such reasoning is only possible when formal, symbolic models of those aspects are available, so the DSLs are expected to be disruptive in that area too.
- The Composition Pattern introduces a significant number of "design forces", which take a bit more time to grasp fully than the more simple 5Cs. The advantage however, is that this more elaborate structure results invariably in much smaller configuration files or software libraries, because developers find it a lot easier to define the scope of each particular software development effort.

This paper focuses on structure, an important complementary research topic, outside the scope of this paper, are formal verification and validation tools, which check consistency of the different models used in an application.

We introduced the Composition Pattern as an architectural proto-pattern. The full assessment of amongst others the different qualities of the pattern, following the format used by Gamma et al. [52], is subject of ongoing work.

None of the above-mentioned lessons learned, and neither the 5C's nor the Composition Pattern, are derived from unshakable "first principles", hence they can, and should, be subject of continuous critical reflections. The higher than usual degree of structure in the presented material should make such refutation a lot easier; but it is this same "easiness" with which human developers can grasp this structure that has led to the maturation of the concepts, and the clarification of the "design forces", to a level that has stood firmly against dozens of new software project developments, as well as refactorings of existing frameworks.

REFERENCES

[1] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *IJRR*, vol. 26, no. 5, pp. 433–455, 2007. 1, 2.2, 2.4, 8

[2] D. Vanthienen, T. De Laet, R. Smits, and H. Bruyninckx, "itasc software," http://www.orocos.org/itasc, 2011, last visited November 2013. 1

[3] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The use of reuse for designing and manufacturing robots," Robot Standards project, Tech. Rep., 2009, http://www.robot-standards.eu//Documents_RoSta_wiki/whitepaper_reuse.pdf. 1.1, 8

[4] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The Brics Component Model: A model-based development paradigm for complex robotics software systems," in *28th ACM Symposium On Applied Computing*, 2013, pp. 1758–1764. 1.1, 3, 7

[5] D. Kortenkamp and R. G. Simmons, "Robotic systems architectures and programming." in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Springer, 2008, pp. 187–206. 2.1

[6] N. J. Nilsson, "Hierarchical robot planning and execution system," Stanford Research Institute, Tech. Rep., 1973. 2.1

[7] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. Kemp, "ROS Commander (ROSCo): Behavior creation for home robots," pp. 467–474. 2.1

[8] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using uml/p statecharts," pp. 461–466. 2.1, 2.2

[9] R. P. Bonasso, D. Kortenkamp, D. P. Miller, and M. Slack, "Experiences with an architecture for intelligent, reactive agents," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, pp. 237–256, 1995. 2.1

[10] A. Angerer, A. Hoffmann, A. Schierl, M. Vistein, and W. Reif, "Robotics api: Object-oriented software development for industrial robots," *J. Software Engin Robotics*, vol. 4, no. 1, pp. 1–22, 2013. 2.1

[11] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and T.-H. Nguyen, "Toward a more dependable software architecture for autonomous robots," *IEEE Rob. Autom. Mag.*, vol. 16, 2009. 2.1

[12] T. Abdellatif, S. Bensalem, J. Combaz, L. de Silva, and F. Ingrand, "Rigorous design of robot software: A formal component-based approach," *Rob. Auton. Systems*, vol. 60, no. 12, pp. 1563–1578, 2012. 2.1

[13] M. Beetz, L. Mösenlechner, and M. Tenorth, "CRAM—A cognitive robot abstract machine for everyday manipulation in human environments," in *Int. Conf. Advanced Robotics*, 2010, pp. 1012–1017. 2.1

[14] P. Doherty, F. Heintz, and J. Kvarnström, "High-level mission specification and planning for collaborative unmanned aircraft systems using delegation," *Unmanned Systems*, vol. 1, no. 1, pp. 75–119, 2013. 2.1

[15] M. T. Mason, "Compliance and force control for computer controlled manipulators," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-11, no. 6, pp. 418–432, 1981. 2.2

[16] H. Bruyninckx and J. De Schutter, "Specification of force-controlled actions in the "Task Frame Formalism": A survey," *IEEE Trans. Rob. Automation*, vol. 12, no. 5, pp. 581–589, 1996. 2.2

[17] B. Siciliano and O. E. Khatib, *Springer Handbook of Robotics*. Springer-Verlag, Berlin, Heidelberg, 2008. 2.2

[18] C. Samson, M. Le Borgne, and B. Espiau, *Robot Control, the Task Function Approach*. Oxford, England: Clarendon Press, 1991. 2.2

[19] W. Decré, R. Smits, H. Bruyninckx, and J. De Schutter, "Extending iTaSC to support inequality constraints and non-instantaneous task specification," in *Int. Conf. Robotics and Automation*, Kobe, Japan, 2009, pp. 964–971. 2.2

[20] D. Vanthienen, T. De Laet, W. Decré, H. Bruyninckx, and J. De Schutter, "Force-sensorless and bimanual human-robot comanipulation," in *10th IFAC Symposium on Robot Control (SYROCO)*, vol. 10, Dubrovnik, Croatia, September, 5–7 2012. i

[21] B. Finkemeyer, T. Kröger, and F. M. Wahl, "Executing assembly tasks specified by manipulation primitive nets," *Advanced Robotics*, vol. 19, no. 5, pp. 591–611, 2005. 2.2

[22] T. Kröger and B. Finkemeyer, "Robot motion control during abrupt switchings between manipulation primitives," in *Workshop on Mobile Manipulation at the IEEE Int. Conf. Robotics and Automation*, Shanghai, China, May 2011. 2.2

[23] N. Mansard, O. Stasse, P. Evrard, and A. Kheddar, "A versatile generalized inverted kinematics implementation for collaborative working humanoid robots: The stack of tasks," in *Int. Conf. Advanced Robotics*, Munich,Germany, 2009. 2.2

[24] L. Sentis and O. Khatib, "Synthesis of whole-body behaviors through hierarchical conrol of behavioral primitives," *Int. J. Hum. Rob.*, vol. 2, no. 4, pp. 505–518, 2005. 2.2

[25] R. Smits, "Robot skills: design of a constraint-based methodology and software support," Ph.D. dissertation, Dept. Mech. Eng., Katholieke Univ. Leuven, Belgium, May 2010. 2.4

[26] H. Bruyninckx and P. Soetens, "Open RObot COntrol Software (OROCOS)," http://www.orocos.org/, 2001, last visited November 2013. 2.4, 4.3, 8

[27] P. Van de Poel, W. Witvrouw, H. Bruyninckx, and J. De Schutter, "An environment for developing and optimizing compliant robot motion tasks," in *ICAR*, 1993, pp. 713–718. 2.4

[28] W. Witvrouw, P. Van de Poel, and J. De Schutter, "COMRADE: Compliant motion research and development environment," in *3rd IFAC/IFIP workshop on Algorithms and Architectures for Real-Time Control*, 1995, pp. 81–87. 2.4

[29] D. Vanthienen, M. Klotzbuecher, T. De Laet, J. De Schutter, and H. Bruyninckx, "Rapid application development of constrained-based task modelling and execution using domain specific languages," in *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, Tokyo, Japan, 2013, pp. 1860–1866. 2.4, 6.1.1, 8

[30] S. Joyeux, "ROCK: the RObot Construction Kit," http://rock-robotics.org/, 2010. 3.1, 3.3

[31] M. Klotzbücher, G. Biggs, and H. Bruyninckx, "Pure coordination using the coordinator–configurator pattern," in *Proceedings of the 3rd International Workshop on Domain-Specific Languages and models for ROBotic systems*, 2012. 3.1, 3.2, 5.1, 6.1

[32] R. Ierusalimschy, W. Celes, and L. H. de Figueiredo, "Lua Programming Language," http://www.lua.org, 2012, last visited October 2013. 3.2, 4.3

[33] M. Klotzbücher, P. Soetens, and H. Bruyninckx, "OROCOS RTT-Lua: an Execution Environment for building Real-time Robotic Domain Specific Languages," in *Int. Workshop on DYn. languages for RObotic and Sensors*, 2010, pp. 284–289. [Online]. Available: https://www.sim.informatik.tu-darmstadt.de/simpar/ws/sites/DYROS2010/03-DYROS.pdf 3.2

[34] R. Smits, H. Bruyninckx, and E. Aertbeliën, "KDL: Kinematics and Dynamics Library," http://www.orocos.org/kdl, 2001, last visited August 2012. 4.1

[35] G. Schreiber, A. Stemmer, and R. Bischoff, "The Fast Research Interface for the KUKA Lightweight Robot," In IEEE Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications How to Modify and Enhance Commercial Controllers (ICRA 2010), Anchorage, USA, May 2010. 4.1, 4.3

[36] Willow Garage, "Willow Garage," http://www.willowgarage.com/, 2011, last visited November 2013. 4.1

[37] T. De Laet, S. Bellens, R. Smits, E. Aertbeliën, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies (Part 1): Semantics for standardization," *IEEE Rob. Autom. Mag.*, vol. 20, no. 1, pp. 84–93, 2013. 4.1, 7.3

[38] T. De Laet, S. Bellens, H. Bruyninckx, and J. De Schutter, "Geometric relations between rigid bodies (part 2): from semantics to software," *IEEE Rob. Autom. Mag.*, vol. 20, no. 2, pp. 91–102, 2013. 4.1, 7.3

[39] B. Houska, H. J. Ferreau, and M. Diehl, "ACADO—An open-source toolkit for automatic control and dynamic optimization," in *Proceedings of the 2009 Belgian-French-German Conference on Optimization*, Leuven, Belgium, 2009, p. 167. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/oca.939/pdf 4.1

[40] Object Management Group, "OMG," http://www.omg.org. 2

[41] National Institute of Advanced Industrial Science and Technology, Intelligent Systems Research Institute, "OpenRTM-AIst," http://www.openrtm.org, last visited November 2010. 4.3

[42] N. Ando, T. Suehiro, and T. Kotoku, "A software platform for component based RT-system development: OpenRTM-Aist," in *Conf. Simulation, Modeling, and Programming of Autonomous Robots*, Venice, Italia, 2008, pp. 87–98. 4.3

[43] A. Brooks, T. Kaupp, A. Makarenko, S. Williams, and A. Orebäck, "Orca: A component model and repository," in *Software Engineering for Experimental Robotics*, ser. Springer Tracts in Advanced Robotics, 2008, pp. 231–251. 4.3

[44] 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 *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, Grenoble, France, 1997, pp. 842–848. 4.3, 4.4

[45] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. B. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009. 4.3

[46] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Int. Conf. Robotics and Automation*, Seoul, Korea, 2001, pp. 2523–2528. 4.3

[47] P. Soetens, "A software framework for real-time and distributed robot and machine control," Ph.D. dissertation, Department of Mechanical Engineering, Katholieke Universiteit Leuven, Belgium, May 2006, http://www.mech.kuleuven.be/dept/resources/docs/soetens.pdf. 4.3, 5.2, 7.1

[48] K. YouBot, "Youbot ros packages," https://github.com/youbot/youbot-ros-pkg/. 4.3

[49] M. Klotzbücher and H. Bruyninckx, "Coordinating robotic tasks and systems with rFSM Statecharts," *J. Software Engin Robotics*, vol. 3, no. 1, pp. 28–56, 2012. 6.1, 8, 8

[50] M. Radestock and S. Eisenbach, "Coordination in evolving systems," in *Trends in Distributed Systems. CORBA and Beyond*. Springer-Verlag, 1996, pp. 162–176. 7, 8

[51] D. Bálek and F. Plášil, "Software connectors and their role in component deployment," in *Proceedings of the IFIP TC6 / WG6.1 Third*

International Working Conference on New Developments in Distributed Applications and Interoperable Systems (DAIS), 2001, pp. 69–84. 7.1

[52] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Reading, MA: Addison-Wesley, 1995. 8

Dominick Vanthienen received his B. Sc. and M. Sc. degrees in mechanical engineering (Burgerlijk Ingenieur) from the University of Leuven, Belgium in 2006 and 2008, respectively. After that he worked as a mechanical production process developer in industry. In 2009 he returned to University to pursue a PhD in the field of Robotics at the University of Leuven, Belgium. His research focuses on control of different robotic systems using constraint-based programming approaches such as iTaSC.

Markus Klotzbuecher In 2004, dr. Klotzbücher received a Dipl.-Inf (FH) degree from the University of Applied Sciences in Konstanz, Germany. After that he worked for several years as an independent consultant in the area of embedded and realtime systems, and also held seminars on embedded Linux system design and Linux device driver development. In 2013 he obtained a PhD in the field of Robotics at the University of Leuven, Belgium. His research focuses on domain specific languages to support constructing complex and reusable robot systems that can operate under hard real-time constraints.

Herman Bruyninckx Dr. Bruyninckx obtained the Masters degrees in Mathematics (Licentiate, 1984), Computer Science (Burgerlijk Ingenieur, 1987) and Mechatronics (1988), all from the University of Leuven, Belgium. In 1995 he obtained his Doctoral Degree in Engineering from the same universsity, with a thesis entitled "Kinematic Models for Robot Compliant Motion with Identification of Uncertainties." He is full-time Professor at the the University of Leuven, and held visiting research positions at the Grasp Lab of the University of Pennsylvania, Philadelphia (1996), the Robotics Lab of Stanford University (1999), and the Kungl Tekniska Hogskolan, Stockholm (2002). Since 2007, he was Coordinator of the European Robotics Research Network EURON (http://www.euron.org), and at the time of the merger of EURON into the euRobotics AISBL, he became that association' Vice-President Research. His current research interests are on-line Bayesian estimation of model uncertainties in sensor-based robot tasks, kinematics and dynamics of robots and humans, and the software engineering of large-scale robot control systems. In 2001, he started the Free Software ("open source") project Orocos (http://www.orocos.org), to support his research interests, and to facilitate their industrial exploitation.