2013 IEEE/RSJ International Conference on
Intelligent Robots and Systems (IROS)
November 3-7, 2013. Tokyo, Japan

Rapid application development of constrained-based task modelling and execution using Domain Specific Languages.

Dominick Vanthienen, Markus Klotzbücher, Joris De Schutter, Tinne De Laet, Herman Bruyninckx

Abstract— **Current state-of-the-art robot program development needs expert programmers. Moreover, most robot programs developed today are robot hardware and software specific, and therefore little reusable without modifications. This paper realizes easier robot (re-)programming, by software framework independent models that can be executed using different hard- and software platforms. First, the paper focuses on the formalization of the tasks to be fulfilled by a robot, more specifically constraint-based programming tasks using a Domain Specific Language (DSL). Second, it gives a reference implementation in Lua [1]. The presented DSL makes it easy to develop applications, yet is powerful to execute. It enables automatic model verification and code generation for different hard- and software platforms, diminishing code debugging efforts. Experimental validation shows the ease of creating an application and adapting it, the reduction of the amount of hand-written code, and the debugging aid offered through meaningful errors returned by model verification.**

I. Introduction

You have an important demonstration to give on your robot, and as Murphy predicted, the robot breaks right before your demonstration. If you only could quickly change to the other robot in the lab, which unfortunately has another kinematic structure. Of course you'll have to adapt your tasks to the new kinematic structure, with another number of degrees-of-freedom, adapt your control gains, redefine tasks, reconnect and configure all parts of the code… Or don't you? If the task concept and software were be separated from your platform description, your problem would be easy to solve. The example outlines the motivation for this work: simpler robot (re-)programming, by software framework independent models that can be executed using different hard- and software platforms.

Different languages have been developed to model and separate concerns involved in a robotic application. Simmons et al. [2] introduced a Task Description Language for robot control, generating a high-level task tree. Nordmann et al. [3] introduced a Domain Specific Language (DSL) for rich motor skill architectures and automated code-generation from the model. Ingés-Romero et al. [4] on the other hand focused on a DSL to express run-time variability, using an optimization problem to bind variability at run-time. These approaches focus primarily on the 'higher-level' task descriptions and scheduling, but have rather generic domain models for robot control tasks. This paper however focuses on the formalisation of the tasks to be fulfilled by a robot, more specifically constraint-based programming tasks. Furthermore, it gives a reference implementation in Lua [1].

All authors are with the Department of Mechanical Engineering, KU Leuven, Belgium. All authors gratefully acknowledge the financial support by the Flemish FWO project G040410N, KU Leuven's Concerted Research Action GOA/2010/011, and European FP7 projects RoboHow (FP7-ICT-288533), BRICS (2008-ICT-231940), and Rosetta (2008-ICT-230902). Tinne De Laet is a PostDoctoral Fellow of the Research Foundation - Flanders (FWO) in Belgium. Corresponding author: `dominick.vanthienen@mech.kuleuven.be`

Constraint-based programming imposes constraints on the modeled relative motions between robots and objects. The paper introduces a DSL that formalizes and structures constraint-based programming applications in robotics, in a way that is simple to use, yet powerful to execute. It further separates concerns, enabling a platform- and application-independent model, and enables automatic model verification and code generation. However, the proposed DSL does not describe all sub-domains of an application, but permits the integration of more specific DSLs such as rFSM [5] for finite-state machines. Hence it forms a DSL between 'higher'-level domains, such as symbolic reasoning or planning and 'lower'-level domains such as control.

DSLs have great potential within robotics to aid robot programming by formalizing domains and enabling automatic model verification and code generation. The Geometric Relation Semantics [6]–[8] project is an example of such a DSL in robotics that shows the assistance of modelling in robot programming. It focuses on the formalization of a small domain and delivers tooling for easy use and integration.

This work uses the instantaneous Task Specification and estimation using Constraints (iTaSC) framework [9], a generalization of constraint-based programming that uses particular sets of auxiliary coordinates to express task constraints and model geometric uncertainty. iTaSC describes a robot application as an optimization problem consisting of a set of constraints and one or multiple objective functions. A software implementation of this framework [10], [11] is available under an open-source license. The framework can handle any kind of robot that can be represented as a kinematic tree.

This paper follows the meta-model approach of Model Driven Engineering (MDE) [12], introducing the concept of Domain Specific Languages (DSL) to constraint-based programming, as such extending the work by Klotzbücher et al. [13]. MDE proposes a systematic approach to model a domain, using four levels of abstraction. This paper follows the meaning given to the levels in [13]:

M3	: Highest level of abstraction, model of the constraints that a valid iTaSC specification DSL should conform to.
M2	: The level of the application-independent iTaSC specification DSL, as a parameterized template.

M1 : The level of application-specific iTaSC specification DSL.

M0 : The level of concrete implementations using software libraries and frameworks.

uMF [14], a declarative and light-weight metamodelling framework forms the M3 level, enabling the modelling and validation of structural constraints on the presented DSL. As for uMF, this paper presents a Lua [1] based internal DSL. Lua is a light-weight scripting language, already integrated in several robotic software frameworks and DSLs, such as Orocos [15], ROS [16], and rFSM [5].

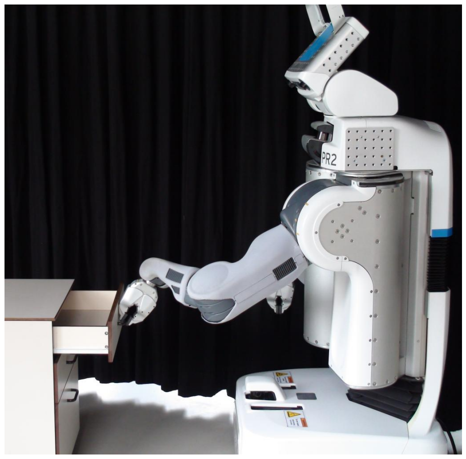

Fig. 1: Setup of the drawer opening example.

The paper first introduces the running example in section II, and then introduces the meta-model of the iTaSC specification DSL in section III. Next it elaborates on a model of an iTaSC specification in section IV. Further it explains the transition from M1 to the executable code on M0 in section V. Section VI discusses and evaluates the proposed DSL, and finally section VII summarizes the innovations and future work.

II. RUNNING EXAMPLE

All concepts introduced in subsequent sections will be explained using the following example. The example consists of a drawer opening application with a PR2 robot as shown in figure 1. The robot has to (i) reach for the handle with its right gripper, (ii) grasp the handle, and (iii) open the drawer, (iv) while keeping close to a preferable joint configuration, and (v) staying away from joint limits. A video and the full model of the example can be found online at [17]. Listings 1 - 3 show the model for the drawer opening part of the example, which will be explained in detail in the following sections.

III. APPLICATION-INDEPENDENT META-MODEL FOR CONSTRAINT-BASED PROGRAMMING (M2)

The M2 model describes a *template* for a constraint-based programming robotics application. The iTaSC framework eases the domain analysis to identify concepts and structures of the domain of the language, since it has a systematic design workflow and software taking into account the separation of concerns [9]–[11].

The design workflow consists of six steps, and is briefly recapitulated here: (i) identify the *robots and objects* involved in the application and their location in the scene, (ii) define the *object frames* on the robots and objects at locations where a task will take effect, (iii) parametrize the space between pairs of object frames, as a *virtual kinematic chain (VKC)* with the *feature coordinates* $\chi_{\boldsymbol{f}}$ as joint coordinates, (iv) choose the *outputs* $\boldsymbol{y} = f(\boldsymbol{q}, \boldsymbol{\chi_f})$ to be constrained, (v) impose *constraints* on the relative motion between two object frames by selecting the type of constraints (equality or inequality) and the control law that enforces them, (vi) select a constraint-optimization problem solver that calculates the desired robot joint inputs.

The software framework on the other hand, reflects this systematic way of describing tasks. The implementation of the functionality follows *the separation of concerns principle of the 5C's* [18]–[20] separating the **c**ommunication, **c**omputation, **c**oordination, **c**onfiguration, and **c**omposition functionality. It builds upon the Orocos software component framework [15] and rFSM statecharts [5], [20], [21]. Furthermore, it integrates with ROS.

Building on the iTaSC theory and software concepts, we developed the iTaSC DSL, integrating well established DSLs such as rFSM.

The design incorporates three *levels* that group coordinated entities. The three levels are, from high to low level: `Application`, `iTaSC`, and `Task`. Each level has a similar structure, with a coordinator `FSM`, and the following three attributes: (i) The `Name` identifies the entity within the model, (ii) the `uri` (Uniform Resource Identifier) uniquely identifies the model, (iii) and the `dsl_version` identifies the M2 model version. The `FSM` of a level coordinates its behavior by communicating events with the level's entities and the `FSM` of the lower level. It is a pure event processor, independent of the other four *concerns*. The `FSM` incorporates the abovementioned rFSM DSL.

Figure 2 gives an overview of the domain entities of the iTaSC DSL and their relations, as will be presented in the following paragraphs. [1]

A. Application level

The `Application` forms the highest level in the entity tree and consists of the following entities:

- The `setpoint_generators` entity holds the models of the different `SetpointGenerator`s, which deliver desired values to the controllers in the application, for example a trajectory in task space to open the drawer. A `SetpointGenerator` can be very different in nature; as simple as a fixed value, complexer trajectory generators, or planners. A `SetpointGenerator` contains a reference to the `Task` that uses the setpoints.

[1]Names of groups of class entities use snake case, names of entity classes use camel case.

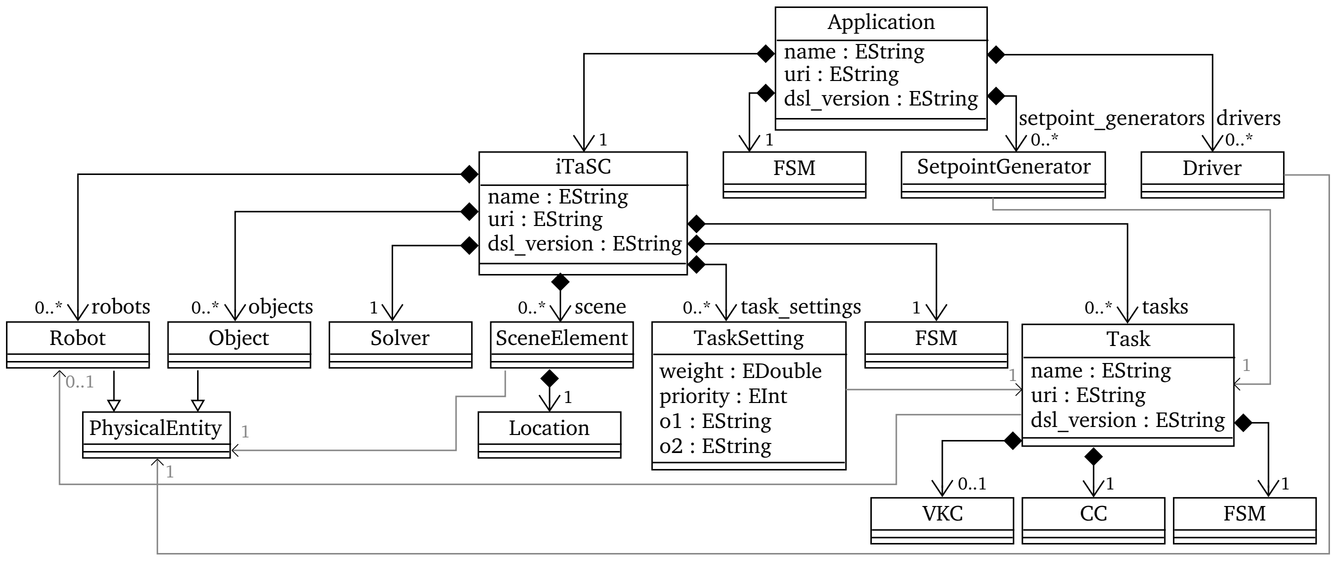

Fig. 2: Overview of the model structure of the DSL with a UML 2.0 class diagram, using the Ecore dialect [22]. Attributes such as `Config`, `package`, and `type` are left out for readability.

- The `drivers` entity holds the models of the external platform- and hardware interfaces, for example the interface to the PR2 controller manager of ROS. A `Driver` contains a reference to the related `PhysicalEntity`.
- The `FSM` coordinates the behavior of the full application for example the configuring and starting of the different setpoint generators, drivers, etc.
- The `iTaSC` entity contains the model of the actual constraint-based task specification, for example all the constraints needed to reach, grasp, and open the drawer. Next section III-B explains `iTaSC` in detail.

B. iTaSC or composition level

The `iTaSC` level composes the different tasks in a *composite task* to be fulfilled by the robots. This composite task consists of a set of constraints resulting in an over- and/or underconstrained [9] optimization problem. The `iTaSC` level consists of following entities:

- The `robots` entity contains the robots involved in the application, eg. the PR2 in the drawer opening example. Each `Robot` integrates the actual model, containing the kinematic and dynamic structure and a reference to the `Driver` to be used. The model integrates software to represent kinematic or dynamic structures such as Collada [23] or URDF [24].
- The `objects` entity contains the objects involved in the application, for example the drawer to be opened. An `Object` has the same structure as a `Robot`, but doesn't have controllable degrees-of-freedom (DOF).
- The `scene` contains `SceneElements` that position a `Robot` or an `Object` at a `Location` in the scene. This `Location` can be fixed or an external input, such as the location of the drawer detected by a computer vision algorithm.
- The `tasks` entity contains the different `Tasks` to be executed by the robot, for example keeping joints away from joint limits. A `Task` will be explained in the next section.
- The `task_settings` entity contains a `TaskSetting` for each `Task`. The `TaskSetting` (i) assigns a `weight` and a `priority` to the `Task`, two measures to deal with over- and/or underconstrainedness of the composite task [25], [26], and (ii) defines the `o1` and `o2` frames between which a `Task` is defined, these frames refer to a frame on a `Robot` or `Object` kinematic model.
  In the running example the task to stay close to a preferable joint configuration has a lower priority than the reaching motion. As such, it defines the *composition* of the composite task. On the other hand, the object frames for the opening task are the `r_gripper_tool_frame` of the PR2 and the `handle_frame` of the drawer.
- The `FSM` of this level coordinates the composite task behavior by enabling and disabling tasks, changing weights and priorities, etc. For example disabling the reaching task and activating the grasping task once an event is received that signals that the handle is reached.
- The `Solver` entity contains the algorithm that solves the optimization problem for a certain objective function, taking the constraints of the composite task into account. This results in the desired inputs for the robot, typically desired joint velocities, accelerations or torques. In the running example a prioritized damped-least squares solver [25], [27], [28] is used, solving for joint velocities. The objective is to minimize the error in task space and the joint velocities.

C. Task level

A Task exists of a set of constraints on the task space, and has following entities:

- The VKC entity models the task space as a *Virtual Kinematic Chain*, with *feature coordinates* as joint coordinates. Since the handle of the drawer is a cylinder and its irrelevant from which side the robot approaches the handle, the example considers a cylindrical task space: TransZ, RotZ, TransX, RotX, RotY, RotZ. Where Trans means translation and Rot rotation, along the direction or around an axis of the moved coordinate frame.
- The CC entity models the *Constraint-Controller* that imposes a desired value on an output, enforced by a controller. The output is a function of the controllable robot joints and feature coordinates. In the open drawer example, we use a simple proportional controller on the position error and velocity feedforward on each feature coordinate ($y = \chi_f$).
- The task needs a reference to the Robots in case their joints are constrained directly, for example for the joint limit avoidance task in the example.
- The FSM coordinates the behavior of one task, for example enabling or disabling a single constraint of a task.

D. Decoupling

Each entity can have a Config entity containing its configuration, a type attribute specifying the specific type of an entity, and the package attribute pointing to the ROS package where to find the implementation of this type. Some entities use references to other parts of the model, enabling their decoupling. For example, a Robot or Object is independent of its Location or Driver. Similarly a Task model is independent from the weight or priority that is assigned to the Task, the object frames o1 and o2 in between which the Task is assigned, or the origin of the setpoint. The references are made using the Name attribute, assigned to the entities that are referred to.

Note the separation of the Configuration in Config, the Coordination in FSM, the Computation in the different entities, and the Composition by for example separating task_settings from the tasks themselves. Communication is not mentioned here, since it will depend on the software platform that is used on the M0 level.

IV. An iTaSC model (M1)

The M1 level model is an instance of the M2 model, filled in with the application-specific information. Due to the limited space, we restrict the example code to the model for the composite task of the drawer opening example, as listed in following sections. The full model can be found on [17].

The M2 model and uMF [14] tools enable to formally verify the conformity of the M1 model to the M2 model. This verification comprises syntax verification, the existence of referred entities and DSLs, and compatibility between entities. The automatic verification returns meaningful errors to the user.

A. Application level

Listing 1: Application level

```
return Application {
dsl_version = '0.1',
name = 'simple_open_drawer',
uri = 'be.kuleuven.mech.rob.app.drawer_simpr2',
fsm = FSM{fsm = "file://app_supervisor.lua"},
setpoint_generators = {
  SetpointGenerator{
    name="open_drawer_trajectory_generator",
    type="trajectory_generators::↩
        nAxesGeneratorPos"
    package="naxes_joint_generator",
    config={"file://open_drawer_traj_gen.cpf"},
    task="pull_drawer_handle"},
  SetpointGenerator{
    name="desired_joint_config_generator",
    type="trajectory_generators::↩
        SimpleGenerator6D"
    package="simple_generator_6d",
    config={"file://desired_jnt_config.cpf"},
    task="keep_joint_config"}},
drivers = { Driver{
  name="pr2_driver",
  file="file://pr2driver.lua",
  robot="pr2"}},
itasc = my_composite_task}
```

The FSM points to the rFSM [20] model of the coordination. The FSM of all levels share the same underlying structure, as shown in figure 3. Each of the states can be a state machine on its own. The configuration of a setpoint_generator is contained in an xml file, with a configuration property file (cpf) extension. All referred packages and types can be found in [17].

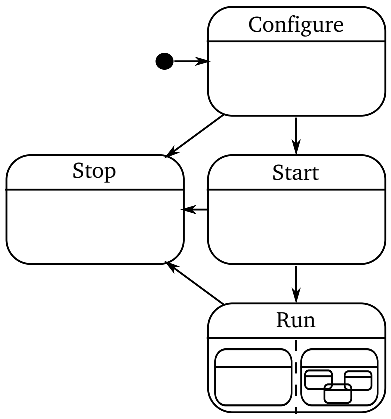

Fig. 3: Basic infrastructure of a FSM of a level. Each state is possibly a (combination) of state machines, as shown for the Run state.

B. iTaSC or composition level

Listing 2: iTaSC level

```
my_composite_task = iTaSC {
dsl_version = '0.1',
name = 'simple_open_drawer_task',
uri = 'be.kuleuven.mech.rob.itasc.drawer_pr2',
fsm = FSM{fsm = "file://itasc_supervisor.lua"},
robots = { Robot{
  name = "pr2",
  package = "itasc_pr2",
  type = "iTaSC::pr2Robot",
  config = {"file://pr2robot.cpf"}}},
objects = { Object{
  name = "upper_drawer",
  package = "fixed_object",
  type = "iTaSC::FixedObject"}},
scene = {
  SceneElement {
    robot = "pr2",
    location = Frame {
      M = Rotation{X_x=1,Y_x=0,Z_x=0,X_y=0,Y_y↩
          =1,Z_y=0,X_z=0,Y_z=0,Z_z=1},
      p = Vector{X=0.0,Y=0.0,Z=0.0}}},
  SceneElement {
    object = "upper_drawer",
    location = Frame {
      M = Rotation{X_x=1,Y_x=0,Z_x=0,X_y=0,Y_y=1,↩
          Z_y=0,X_z=0,Y_z=0,Z_z=1},
      p = Vector{X=2.8,Y=0.0,Z=0.5}}}},
task_settings = {
  TaskSetting{
    task = "pull_drawer_handle",
    weight = 1.0,
    priority = 1,
    o1 = "upper_drawer.handle",
    o2 = "pr2.r_gripper_tool_frame"},
  TaskSetting{
    task = "keep_joint_config",
    weight = 1.0,
    priority = 2,
    robot = "pr2"}},
solver = Solver{
  name="Solver",
  package="wdls_prior_vel_solver",
  type="iTaSC::WDLSPriorVelSolver"},
tasks = my_tasks}
```

The `iTaSC` level model introduces the PR2 robot and the drawer to open. The drawer could be part of a cupboard specified in a Collada model, but is omitted here for readability. The `Location` is expressed in the uMF frame specification, consisting of a rotation matrix and position vector. The locations are expressed with respect to the implicit world frame.

`o1` and `o2` of the pull_drawer_handle task refer to the handle of the upper_drawer `Object` and the gripper of the PR2 `Robot` respectively. The task to pull open the drawer doesn't need a reference to a `Robot`, since there are no constraints in joint space, while the task to keep a preferred joint configuration has a `Robot` reference, but no `o1` or `o2` since all constraints are in joint space.

`Tasks` with a lower `priority` number have priority over `Tasks` with a higher `priority` number. In the running example, the task to pull the drawer handle has priority over the task to keep a certain joint configuration. The weights will have no effect in this reduced example, since there are no conflicting constraints within each priority level.

The `FSM` of the iTaSC level is rather limited for the running example, since all tasks are running in parallel during this single opening action. The full drawer opening application needs more complicated, multi-state coordination at run-time [17].

C. Task level

Listing 3: Task level

```
my_tasks = tasks{
Task{
  name = "pull_drawer_handle",
  dsl_version = '0.1',
  uri = 'be.kuleuven.mech.rob.task.pull_handle',
  vkc = VKC{
    type= "iTaSC::VKC_sixDof",
    package="sixDof_pff",
    config={
      "file://VKC_sixDof.cpf" ,
      {chain={"TransZ","RotZ","TransX","RotX","↩
        RotY","RotZ"}}}},
  cc = CC{type="iTaSC::CC_sixDof_pff",
    package="sixDof_pff",
    config={"file://CC_sixDof_pff.cpf"}},
  fsm = FSM{
    fsm = "file://sixDof_pff_supervisor.lua"}},
Task{
  name = "keep_joint_config",
  dsl_version = '0.1',
  uri = 'be.kuleuven.mech.rob.task.keep_jnt_cfg'
  cc = CC{
    type = "iTaSC::CC_PDFFjoints",
    package = "joint_motion",
    config = {"file://CC_PDFFjoints.cpf"} }
  fsm = FSM{
    fsm = "file://jnt_config_supervisor.lua"}}}
```

The task to keep a preferred joint configuration has no `VKC`, since all constraints are in joint space. The configurations are contained in a xml file with .cpf extension. The types of the `CC` and `VKC` can be found in [11]; the sixDof_pff refers to a simple proportional controller with feed-forward for six DOF output $\boldsymbol{y}$, in this case the feature coordinates of the virtual kinematic chain VKC_sixDof, while the CC_PDFFjoints refers to a similar, more general PD controller with feed-forward for an n-DOF output $\boldsymbol{y}$.

V. Code generation: from M1 to M0

The M1 model *specifies* a robot application, that has to be transformed into an *implementation* that conforms to this M1 model. We provide software support that transforms the M1 model to a run-time configuration and instantiation using the existing iTaSC software implementation [11]. This iTaSC software is developed using the Orocos component framework.

VI. Experiments and evaluation

Table I compares the required lines of code for two more elaborate examples from previous work: lissajous-tracing with a KUKA youBot [29] and human-robot comanipulation with a PR2 robot [10]. The table shows the lines of code to be hand coded to generate the application. Next to this code, both implementations share the iTaSC Orocos components needed for the execution. The total lines to be hand coded have reduced by a factor of 2.5 when using the DSL. This reduction is possible by the automatic derivation of framework specific code from the model. Moreover the model provides a better readable overview of the application and introduces names in a more consistent 'hierarchical' manner: lower levels introduce names, referenced to by higher levels. In order to allow this referencing, each level has to expose the names of the entities it contains to the higher levels.

	laser tracing	comanipulation
model	97	155
original code	237	416

TABLE I: Comparison of code efficiency by lines of code of a laser tracing and comanipulation example.

The warnings and errors that the execution of the M1 model verification returns include:

- Syntax errors, such as the misspelling of an attribute or entity, or the assignment of a wrong type. For example when erroneously using `ame = 'pull_drawer_handle'` in stead of `name = 'pull_drawer_handle'` when assigning a name to the first task:

```
err@ app.itasc.tasks[1].ame:
  illegal field 'ame' in sealed dict
  (value: pull_drawer_handle)
err@ app.itasc.tasks[1]:
  non-optional field 'name' missing
```

  or when assigning a number to the `name`:

```
err@ app.itasc.tasks[2].name:
  not a string but a number
```

- The non-existence of referred entities and DSLs, for example robots not listed in `robots` or not found configuration files:

```
err@ app: failed to resolve SceneElement.↩
    robot:
  PR3
err@ app.itasc.tasks[1].fsm.fsm:
  non-existing configuration file pr2robot.↩
      cpf
```

- Incompatibility between entities, for example when assigning an always singular Virtual Kinematic Chain with two Z rotational joints as first two joints:

```
err@ app.itasc.tasks[1].vkc.config[2].chain:
  identical consecutive chain segments (1-2)
```

- The use of the same name for multiple entities, for example when giving the drawer and the robot the name 'pr2':

```
err@ : duplicate use of name pr2
```

- The use of an outdated version of the meta-model:

```
warn@ app.itasc.dsl_version:
Current iTaSC meta-model version number 0.1,
does not match required version number 0.2
```

One of the major advantages of the developed DSL is its ease to create and adapt applications. As a proof, the following paragraphs summarize some possible changes of the running example. Video fragments of some of the changes can be found at [17].

To change the robot that is used, as in the case given in the Introduction, one has to change: (i) the `Robot` (listing 2, line 6) to for example the KUKA YouBot, (ii) the robot `Driver` (listing 1, line 20), (iii) the `o2` of tasks that use the robot, in this case of the pull_drawer_handle task (listing 2, line 32), (iv) the configuration of joint space tasks, such as the keep_joint_config task (listing 3, line 24), (v) and possibly the `SceneElement` of the robot in the scene (listing 2, line 17). A total of 13 minor modifications are needed to change the robot platform used, far less than the more than 100 lines without the model. The same reasoning holds for changing an `Object`. In case one wants to open another drawer with the same model, the change can be as little as one line (listing 2, line 22).

Another common change to an application is the relation between the tasks by changing the `weight` and `priority` of one or multiple tasks. These settings are grouped by the `task_settings` entity. It is common that these settings are changed at run-time, for instance by the `FSM` at iTaSC level.

In case the drawer doesn't slide but swivels open like a door, one can adapt the model of the task space easily, by changing the configuration of the `VKC` (listing 3, line 11). Cylindrical coordinates, ease the task specification of the running example to a constraint on a single DOF, namely the angle around its pivot.

In case the drawer has a rim that is easier to grip, one can easily change the `o1` frame to this rim (listing 2, line 32).

Another common alteration is the change of control used for a task, which is easily done by replacing the `CC` of the task, for example listing 3 lines 12-14 to an impedance controller.

In case one wants to change the coordination of the composite task, for example deactivating the keep_joint_configuration task once the handle is grasped or change how and when the transition from grasping the handle to opening the drawer occurs, one only has to change the `FSM` of the iTaSC level (listing 2, line 5).

Further one can easily change which of the two arms of the robot should be used, by simply changing one word, namely the object frame on the robot; for the running example replace `r_gripper_tool_frame` by `l_gripper_tool_frame` (listing 2, line 32).

VII. Conclusions and Future Work

This paper structures and formally models constraint-based programming tasks using a Domain Specific Language (DSL). The presented DSL makes application development easy, yet is powerful to execute. Furthermore, the DSL enables automatic model verification and code generation for different hard- and software platforms, diminishing code debugging efforts. Moreover, it is shown that the needed code and hence development time of constraint-based programming applications can be significantly reduced. Next to reduced code size, the rapid development originates from (i) the DSL as a scripting language, without need for compilation, (ii) the separation of concerns, leading to a structured set of small configuration files that are easily adapted, (iii) and the DSL as a template, guiding the programming effort.

The model separates concerns following the 5 C's principle [18]–[20] and groups reusable functionality, allowing non-experts to develop applications by *composing* tasks and assigning them to robots and objects in the scene. As such, it can be viewed as a first step towards the robot programming language of the future. Moreover, the structured approach and DSL allows to integrate iTaSC in graphical programming tools, such as ABB's RobotStudio [30].

The proposed model opens up the possibility of tool support for design time model checking, using for example Xtext [31]. Further it allows the creation of a repository or store with models and/or implementations of entities, such as tasks i.e. a 'task store'.

Future work will focus on the integration of dedicated DSLs for all entities. Furthermore, the execution of the model on other constraint-based programming frameworks such as Stack of Tasks [32] will be investigated. Additionally, the presented formal modelling of constraint-based programming paves the way for robots to generate their own behavior, and reason on their behavior on a symbolic level.

The software implementation of the DSL will be made available under an open-source license.

References

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

[2] R. Simmons and D. Apfelbaum, "A task description language for robot control," in *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, vol. 3, oct 1998, pp. 1931 –1937 vol.3.

[3] A. Nordmann and S. Wrede, "A domain-specific language for rich motor skill architectures," in *3rd International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*, Tsukuba, Japan, 2012 2012.

[4] J. F. Inglés-Romero, A. Lotz, C. Vicente-Chicote, and C. Schlegel, "Dealing with run-time variability in service robotics: towards a dsl for non-functional properties," in *3rd International Workshop on Domain-Specific Languages and Models for Robotic Systems (DSLRob)*, Tsukuba, Japan, 2012 2012.

[5] M. Klotzbuecher, 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

[6] 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.

[7] 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.

[8] T. De Laet and S. Bellens, "Geometric semantics software," http://www.orocos.org/wiki/geometric-relations-semantics-wiki, 2012, last visited September 2012.

[9] 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," *Int. J. Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007.

[10] 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.

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

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

[13] M. Klotzbuecher, R. Smits, H. Bruyninckx, and J. De Schutter, "Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages," in *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, San Francisco, California, 2011, pp. 4684–4689.

[14] M. Klotzbuecher and H. Bruyninckx, "A lightweight, composable metamodelling language for specification and validation of internal domain specific languages," in *Proceedings of the 8th International Workshop on Model-based Methodologies for Pervasive and Embedded Software*, September 2012.

[15] H. Bruyninckx and P. Soetens, "Open RObot COntrol Software (OROCOS)," http://www.orocos.org/, 2001, last visited March 2013.

[16] Willow Garage, "Robot Operating System (ROS)," http://www.ros.org, 2008, last visited 2012.

[17] D. Vanthienen, M. Klotzbuecher, T. De Laet, J. De Schutter, and H. Bruyninckx, "itasc dsl," http://people.mech.kuleuven.be/~dvanthienen/IROS2013, 2013, last visited July 2013.

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

[19] 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.

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

[21] M. Klotzbuecher, 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*, November 2012.

[22] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[23] M. Barnes and E. L. Finch, "COLLADA—Digital Asset Schema Release 1.5.0," http://www.collada.org, 2008, last visited March 2010.

[24] Willow Garage, "Universal Robot Description Format (URDF)," http://www.ros.org/urdf/, 2009.

[25] B. Siciliano and J.-J. E. Slotine, "A general framework for managing multiple tasks in highly redundant robotic systems," in *Int. Conf. Advanced Robotics*, Pisa, Italy, 1991, pp. 1211–1216.

[26] Y. Nakamura, *Advanced robotics: redundancy and optimization*. Reading, MA: Addison-Wesley, 1991.

[27] H. Hanafusa, T. Yoshikawa, and Y. Nakamura, "Analysis and control of articulated robot arms with redundancy," in *Proc. IFAC World Cong.*, Kyoto, Japan, 1981, pp. XIV:78–83.

[28] P. Baerlocher and R. Boulic, "Task-priority formulations for the kinematic control of highly redundant articulated structures," in *Proc. IEEE/RSJ Int. Conf. Int. Robots and Systems*, Vancouver, British Columbia, Canada, 1998, pp. 323–329.

[29] D. Vanthienen, "itasc tutorials," http://orocos.org/wiki/orocos/itasc-wiki/itasc-tutorials, 2013, last visited July 2013.

[30] ABB, "ABB Robotics," http://www.abb.com/robotics/, 2011.

[31] "Xtext," http://www.eclipse.org/Xtext/, last visited July 2013.

[32] 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.