iTaSC 2.0 Manual

—

version 0.1

Dominick Vanthienen
Mechanical Engineering, K.U.Leuven, Belgium

November, 2011
Terminology

- **iTaSC**: The paradigm for task specification
- **Virtual Kinematic Chain (VKC)**: Spatial relation model with $\geq 6$ indpt. auxiliary coordinates $\chi_f$ + estimation + uncertainty $\chi_u$.
- **Interaction Model (IM)**: Similar as VKC, but returns $H$ instead of $J_f^{-1}$.
- **Constraint/Controller**: Output equation (=combination of $\chi_f$ and $q$) and controller.
- **Robot**: Kinematic model of a robotic manipulator: $q$ and uncertainty $\chi_u$.
- **Object**: Kinematic model of an object and uncertainty $\chi_u$.
- **SceneGraph**: Component that collect the spatial configuration of the elements in the Scene.
- **Scene**: Combination of SceneGraph, Robots and Objects: All elements and their pose in space.
- **Solver**: Algorithm to solve the desired joint output
- **Supervisor**: Implements a skill
- **Skill**: A specific combination of the configuration and coordination of Tasks
- **Task**: Combination of Constraint/Controller with VKC or IM (and/or Trajectory Generator)
- **Composite Task**: Combination/composition of tasks
Concept

How to make a new iTaSC application

▶ empty Scene

▶

▶

▶

▶

▶
How to make a new iTaSC application

- empty Scene
- add Robots and Objects
How to make a new iTaSC application

- empty Scene
- add Robots and Objects
- define Object Frames
How to make a new iTaSC application

- empty Scene
- add Robots and Objects
- define Object Frames
- add SubTasks
  - Virtual Kinematic Chains
How to make a new iTaSC application

- empty Scene
- add Robots and Objects
- define Object Frames
- add SubTasks
  - Virtual Kinematic Chains
  - Constraint/Controller
How to make a new iTaSC application

- empty Scene
- add Robots and Objects
- define Object Frames
- add SubTasks
  - Virtual Kinematic Chains
  - Constraint/Controller
- add a Solver
local RobotLocation = rtt.Variable("KDL.Frame")
local TableLocation = rtt.Variable("KDL.Frame")

return rfsm.simple_state{
    entry=function()

-- define the LOCATION of the base of the robots and objects
RobotLocation:fromtab( {M={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={X=0.0,Y=0.0,Z=0.0}} )
TableLocation:fromtab( {M={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={X=1.0,Y=1.0,Z=0.0}} )

-- add ROBOTS to the scene
-- addRobot("<robot component name>",<kdl.Frame with location of the base in the scene>)
addRobot("Robot", RobotLocation)
addRobot("Table", TableLocation)

-- add OBJECT FRAMES to the robots
-- addObjectFrame("<objectFrameName you choose for this objectFrame>",
-- "<segment of robot where it is attached to>","<robot component name>")
addObjectFrame("endEffector_robot","ee","Robot")
addObjectFrame("top_table","top","Table")

-- add TASKS
-- addVirtualKinematicChains or InteractionModels
-- addVirtualKinematicChain("<name of the VKC component>",{"<1st objectFrameName as chosen above>",
-- "<2nd objectFrameName as chosen above>")
addVirtualKinematicChain("VKC_TraceLine","endEffector_robot","top_table")

-- add Constraints/Controllers
-- add ConstraintController("<name of the CC component>",{"<1st objectFrameName as chosen above>",
-- "<2nd objectFrameName>","<name of the VKC to constrain>",{<prioritynumber>}
addConstraintController("CC_TraceLine","endEffector_robot","top_table","VKC_TraceLine",1)

-- add SOLVERS
addSolver("Solver")
end,
}
OROCOS components: Application level

- Application supervisor
- iTaSC
- Robot interface
- Sensor
- Filter
OROCOS components: iTaSC level

Tasks
Constraint/Controllers
Output equation
\( y = f(q, \chi_f) \)
Control equation
\( \dot{y}_d = f(\dot{y}_d, y_d, y) \)

Virtual Kin. Chains/Interaction Models
Prediction: \( \tilde{\chi}_f \)
Correction: \( \hat{\chi}_f \)

Measurements
\( h(q, \chi_f) = z \)

Supervisor
Configuration
Coordination

Scene
SceneGraph
Generation of A matrix

Solver
\( \dot{q}_d = A_W^{\lambda,\#} (\dot{y}_d + B \hat{\chi}_u) \)

Robots/Objects
Robot model

Communication interfaces
OROCOS components: iTaSC level

Tasks
Constraint/Controllers
Output equation
\[ f(q, \chi_f) = y \]
Control
\[ \dot{y}_d = \dot{y} = \dot{y}_d + K_p (y_d - y) \]

Virtual Kin. Chains/Interaction Models
Prediction: step 1
\[ \chi_f = -t^{-1} \left( t_1 q + t_2 \chi_u \right) \]
\[ y = \chi_f = \int \chi_f \]
Prediction: step 2: loop closing iteration
Correction

Measurements
\[ h(q, \chi_f) = z \]

Supervisor
Configuration
Tasks definition (in .cpf)
Connect ports of the Scene
Generate tasks
Coordination
Setup (call config/start/stop-Hook)
Timing: call operations running iTaSC

SceneGraph
Generation of A matrix per priority

Robots/Objects
Robot model

Communication interfaces
OROCOS components: run.ops

1. import packages
2. create components
3. give components a frequency (setActivity), 0 for iTaSC level (timer triggered)
4. make components that will communicate with each other peers
5. load the FSM’s in the Supervisor components
6. configure the Supervisors
7. load properties of the components
8. connect ports of the application level components
9. configure and start application level components
10. start the timer
Skills

- **Skill:** A specific combination of the configuration and coordination of Tasks
- Implemented using Finite State Machines (FSMs)
- **LUA** scripting language
- **Events** cause the FSMs to transition from one state to another
- Each **FSM** should be **OROCOS (RTT) independent**
- Each FSM is loaded in a **Supervisor component**, which contains the OROCOS (RTT) specific parts of the FSM
- 3 FSMs (levels) for an iTaSC application: **Application**, **iTaSC**, **Task** ⇒ your application is always in 3 states: one for each of level
- Each state shown on the next slide consists of two separate states in practice: eg. Configure ⇒ Configuring and Configured
Skills: Finite State Machines

Figure: Black arrow = state transition, blue arrow = how event triggers state transition.

- iTaSC coordinator and task FSM are running sequentially
- GREEN = variation points → you may/should adapt this
Skills: The 3 levels of FSMs

1) Application FSM
   ▶ Coordinates the components outside iTaSC and iTaSC as a whole
   ▶ Put in this FSM the statements to configure/start/stop...the components on the application level, in the corresponding state
Skills: The 3 levels of FSMs

2) iTaSC FSM
   ▶ Coordinates the components **inside iTaSC, except the task components** (VKCs and CCs), they are threatened as a whole
   ▶ Put in this FSM the statements to configure/start/stop... the components on the iTaSC level, in the corresponding state

3) Task FSMs
   ▶ Each Task FSM coordinates the **task components** of a certain task (coresponding with 1 Virtual Kinematic Chain, if one exists)
   ▶ Put in this FSM the statements to configure/start/stop... the components of a certain task, in the corresponding state
Skills: the Running state

- The running states (except application level) consists of 2 sequential parts: an algorithm coordination part and a subFSM part

- **Algorithm coordination:** This part takes care of the right order of execution of the algorithm. The iTaSC algorithm coordination (running_itasc_coordination.lua) takes the lead and requests the algorithm coordination of the tasks at a certain moment (all tasks in parallel), by sending a triggerTasks event.

- **SubFSM:** The iTaSC subFSM (composite_task_fsm.lua) contains the actual behaviour you want from the application. It will trigger the subFSMs of the Tasks involved in a certain Composite Task (=Combination of Tasks, eg. the laser tracing on the barrel and the laser tracing on the table in the first example)
Skills: the Running state: subFSMs: example

**composite_task fsm.lua**

- **TraceFigureOnTable state**
  - send event: “e_traceCircleOnTable”

- **TraceFigureOnTableAndBarrel**
  - send event: “e_traceCircleOnTable”
  - send event: “e_traceSquareOnBarrel”

**running_trace_figure_on_table_fsm.lua**

- **TraceCircleOnTable**
  - startCircleGenerator
  - activateTask (in RTT: W=1)

- **HoldStill**
  - stopCircleGenerator
  - activateTask

**running_trace_line_on_barrel_fsm.lua**

- **TraceSquareOnBarrel**
  - startSquareGenerator
  - activateTask

- **HoldStill**
  - stopSquareGenerator
  - activateTask
Tasks

- Tasks are **general**, separate packages, that can be shared eg. over the internet
- A task package should contain:
  - `/scripts/` (FSM’s and supervisor component in lua)
    - task_supervisor.lua
    - task_fsm.lua
    - running_task_coordination.lua
    - running_task_fsm.lua
  - `/cpf/` (properties)
    - VKC_task.cpf
  - ...
  - `/src/` (components)
    - VKC_task.hpp+cpp
    - or IM_task.hpp+cpp
    - CC_task.hpp+cpp
- A task package contains no info of scene, particular robots etc.
Conclusions

Steps to an implementation

▶ Design your application (first on paper)
  ▶ Which Robots/Objects/Frames are involved
  ▶ Virtual Kinematic Chains
  ▶ Constraints/Controllers
  ▶ Solver

▶ Create/download task packages

▶ Change the FSM scripts on the three levels (application-iTaSC-task)
  ▶ Create run.ops and run.sh (to execute run.ops) file
  ▶ Test it!