Generation of Flow-Preserving Orocos Implementations of Simulink/Scicos Models

M. Morelli[1] and M. Di Natale[1]

Models are used in control domains such as automotive and avionics for early validation of system properties, using simulation or formal verification (model-checking), and for the automatic generation of control software. Model-Based Design (MBD) tools such as Simulink [1] and Scicos [2], [3], based on a synchronous-reactive (SR) execution semantics, allow the modeling and simulation of hybrid systems, in which functionality is represented using an extended finite-state machine formalism. Early verification of the system functionality by simulation or model-checking is highly valuable in many modern robotics systems, which perform complex actions demanding a timely, predictable and certifiably safe behaviour in many applications (e.g., disaster recovery and space exploration).

MBD modeling tools represent the control functionality in abstract terms, that is, executing according to a set of events in logical time. When the functionality is implemented on a computing platform that executes the controls as a set of software tasks and network messages, the *synchronous assumption* (conformance with respect to the model execution semantics) must be preserved despite computation and communication delays [4].

In most cases, designers are interested in a looser property, called flow preservation, consisting of the guarantee that the signal data in the implementation are the same as in the model. However, checking whether a platform allows for a flow-preserving implementation (or possibly synthesizing one), or evaluating the consequences of additional delays in case an implementation without delays is not feasible, requires a detailed model of the control implementations and the execution hardware and software [5].

We propose an approach in which a functional model of the controls is matched to a purposely constructed model of the execution platform through an intermediate mapping model, that represents the software tasks and messages (local or on the network) that realize the functions on the given platform.

For the model of the execution platform, there are several possible options. Our objective is to be able to represent the common execution hardware in use in robotics systems and, on top of it, the Orocos-RTT framework [6], which encapsulates communication and RTOS services and acts as a middleware.

[1]TECIP Institute, Scuola Superiore S. Anna, Pisa, Italy
matteo.morelli, marco.dinatale at sssup.it

We defined our model according to a custom metamodel in the Ecore/Eclipse framework, although our future work includes the definition of a profile for robotics applications on top of the standard UML/SysML MARTE profile [7].

Our metamodel not only provides the concepts to define the platform, the mapping and evaluate the computation and communication delays, but can also be used for the automatic generation of a robotics controls implementation on top of Orocos.

This is done using the standard (and open source) Model-to-Code transformation tool Acceleo.

Our approach consists of the following steps. A functional model of the controls is developed in Simulink or Scicos and code is generated for each subsystem (with the restriction that subsystems must be single-rate). Next, an abstract view of the functional model is generated and imported in the Ecore framework. Using Ecore, a model of the execution platform is created, using our custom metamodel, and an implementation of the subsystems as a set of tasks, executing on the platform nodes and exchanging messages is defined.

The task model may be constrained in such a way that only flow-preserving implementation of the functional model are allowed. Next, the Acceleo tool processes the mapping model and generates the Orocos description of the tasks and the inter-task communication according to the specification. The Orocos tasks execute the C/C++ functions, generated from Simulink or Scicos and implementing the control subsystems.

In the development of our flow, we highlighted several limitation of the Orocos framework and we outline the desirable traits of a modeling language and a middleware support for producing a *correct* flow-preserving implementation of a model as a multi-task application running on a possibly distributed system.

A simple use case is discussed to illustrate the applicability of the proposed approach: the reconstruction of the joint-position profile of a Kuka LWR arm, given a desired end-effector pose $\mathbf{T}_d$ (constant).

This task consists in solving the inverse kinematics problem at the first-order differential level. We compute the joint-velocity profile according to the Jacobian transpose algorithm as $\dot{\mathbf{q}} = \mathbf{J}^T(\mathbf{q})\mathbf{K}\mathbf{e}$, and adopt the Forward Euler numerical integration scheme with an integration time of $\Delta t = 1$ms. We also use a post-processing block (currently a simple undersampling block) that adapts the output rate of the computed set-points to the dynamics of the robot arm.

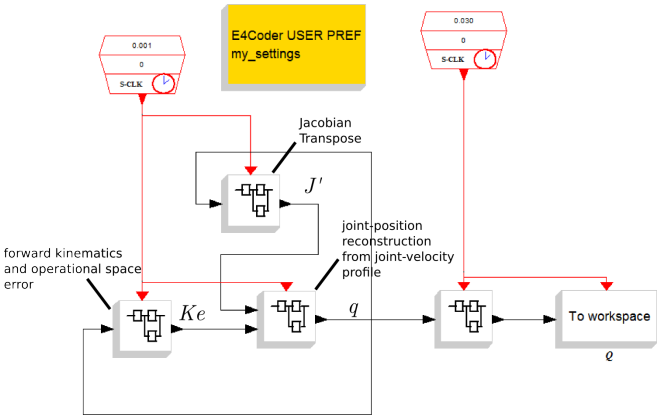

Fig. 1: Scicos model of the Jacobian transpose inverse kinematics algorithm.

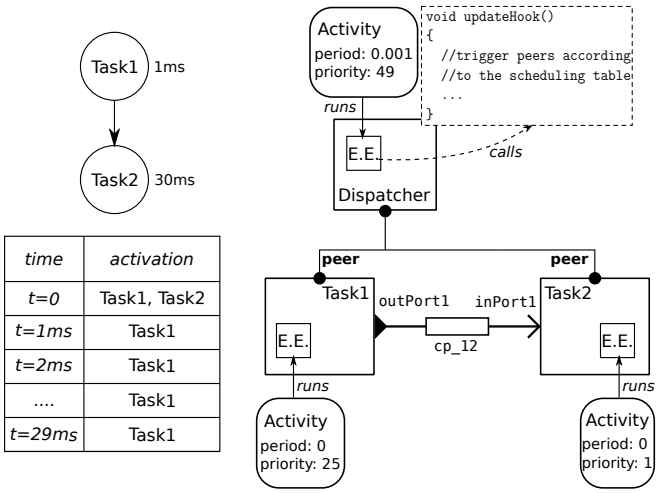

Fig. 3: Deployment of subsystems and threads in a single core.

The inverse kinematics scheme is modeled in Scicos and validated by simulation (Figure 1). Scicos blocks implementing robot-specific functionalities (e.g., forward kinematics, Jacobian computation) are provided by the RTSS library [8].

The designer partitions the multi-rate control model in a set of superblocks. For each superblock, the execution is controlled by a single discrete-time clock. Once the simulation results are satisfactory, the behavioral code is automatically generated for each superblock by E4Coder [9] (Figure 2), and the Scicos model is translated into the XML format that is used to import functional model descriptions in the Ecore framework.

Next, the designer defines an Ecore model of the execution platform, with the available hardware and software resources. In our example, the execution platform is modeled as a single-CPU computation node running Orocos-RTT on top of Real-Time Linux [10]. Finally, the mapping model is defined, describing the implementation of functional superblocks inside two software tasks. Superblocks executing at 1ms are mapped into `Task1` and the remaining superblock into `Task2` (Figure 2). `Task1` executes before `Task2` to preserve the order of execution.

From the models above, Acceleo scripts generate two `RTT::TaskContext` objects executed by *non*-periodic activities. The `RTT::TaskContext Task1` is given a priority of 25, whereas `Task2` is given a priority of 1. To guarantee the execution order among tasks, one additional `RTT::TaskContext` instance is generated as a dispatcher task, which runs *periodically* at the base period (1ms), and with highest priority (49). Then, `Task1` and `Task2` are added as peer components to the dispatcher, which is also provided with a scheduling table specifying which tasks must be activated, and in which order, in the hyperperiod (Figures 2 and 3).

The platform and mapping models are used also to generate the code implementing the flow-preserving communication between the functional subsystems mapped into the RTT components. Intra-task communications are resolved simply accessing global data variables, whereas the communication between `Task1` and `Task2` is realized by lock-free, thread-safe write and read accesses to the ports of the components.

References

[1] The MathWorks, Inc. Simulink: Simulation and Model-Based Design. [Online]. Available: http://www.mathworks.com/products/simulink/

[2] INRIA. Scicos: Block diagram modeler and simulator. [Online]. Available: http://www.scicos.org/

[3] S. L. Campbell, J.-P. Chancelier, and R. Nikoukhah, *Modeling and Simulation in Scilab/Scicos with ScicosLab 4.4*, 2nd ed. Springer Publishing Company, Incorporated, 2009.

[4] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli, "Synthesis of multi-task implementations of simulink models with minimum delays," vol. 6 No 4, 2010.

[5] A. Sindico, M. Di Natale, and A. Sangiovanni-Vincentelli, "An industrial application of a system engineering process integrating model-driven architecture and model based design," Sept. 30th - Oct. 5th 2012.

[6] H. Bruyninckx. Open RObot COntrol Software. [Online]. Available: http://www.orocos.org/

[7] OMG: Object Management Group. Modeling analysis of real time embedded systems (marte) profile. [Online]. Available: http://www.omg.org/spec/MARTE

[8] M. Morelli and A. Bicchi. RTSS – the Robotics Toolbox for Scilab/Scicos. [Online]. Available: http://rtss.sourceforge.net

[9] Evidence Srl. E4Coder: The toolset for simulation and code generation for embedded devices. [Online]. Available: http://www.e4coder.com/

[10] Real-Time Linux. [Online]. Available: https://rt.wiki.kernel.org/index.php/Main_Page

```
/*
 * Superblock that reconstruct the joint-position profile from the joint-velocity profile:
 * Computes: q(k) = 0.001 * ( J'(q(k-1))*K*e(k-1) ) + q(k-1)
 */
void Superblock_32_step(void)
{
 /******** Output Updates ********/
 delayTD_block_32_82_out[0] = delayTD_32_82_dstate[0];
 ...
 delayTD_block_32_82_out[5] = delayTD_32_82_dstate[5];
 /* Read Input: J' */
 CG_read(Superblock_32, 1, cg_read_block_32_9_out);
 /* Read Input: K*e */
 CG_read(Superblock_32, 2, cg_read_block_32_11_out);
 /* J'*K*e */
 math_dgemm_OutputUpdate( cg_read_block_32_9_out, cg_read_block_32_11_out, cg_dgemm_block_32_14_out);
 /* Gain Block: 0.001 * ( J'*K*e ) */
 gain_block_32_83_out[0] = 0.001 * cg_custom_block_32_14_out[0];
 ...
 gain_block_32_83_out[5] = 0.001 * cg_custom_block_32_14_out[5];
 /* Summation Block: Gain Block + q */
 sum_block_32_85_out[0] = gain_block_32_83_out[0] + delayTD_block_32_82_out[0];
 ...
 sum_block_32_85_out[5] = gain_block_32_83_out[5] + delayTD_block_32_82_out[5];
 /* Write Output: q(k) = 0.001 * ( J'(q(k-1))*K*e(k-1) ) + q(k-1) */
 CG_write(Superblock_32, 1, delayTD_block_32_82_out);

 /******** State Updates ********/
 delayTD_32_82_dstate[0] = sum_block_32_85_out[0];
 ...
 delayTD_32_82_dstate[5] = sum_block_32_85_out[5];
}
```

```
/* Task1: {32, 65, 31} */
class Task1 : public TaskContext
{
  ...
  public: /* methods */
  Task1(std::string const& name)
    : TaskContext(name)
  {
    ...
  }
  bool configureHook()
  {
    Superblock_32_init();
    Superblock_65_init();
    Superblock_31_init();
    return true;
  }
  void updateHook()
  {
    Superblock_32_step(); // q(k)
    Superblock_65_step(); // K*e(k)
    Superblock_31_step(); // J'(q(k))
  }
  void cleanupHook()
  {
    Superblock_32_end();
    Superblock_65_end();
    Superblock_31_end();
  }
};
```

```
...
Task1* pTask1;
Task2* pTask2;
...
int ORO_main(int argc, char** argv)
{
  ...
  // Create the tasks:
  Task1 task1("Task1");
  Task2 task2("Task2");
  Dispatcher dispatcher("Dispatcher");

  // Point the TaskContexts
  // (These pointers are used by the code that implements the
  // flow-preserving communication to access the components'
  // ports, when connected functional subsystems are mapped onto
  // different RTT components)
  pTask1 = &task1;
  pTask2 = &task2;

  // Create the activities which run the tasks' engines:
  task1.setActivity( new Activity(ORO_SCHED_RT, 25,  0) );
  task2.setActivity( new Activity(ORO_SCHED_RT, 1, 0) );
  dispatcher.setActivity( new Activity(ORO_SCHED_RT, 49, 0.001) );

  // Assign peers
  dispatcher.addPeer(&task1);
  dispatcher.addPeer(&task2);

  // Create data-flow connections
  ConnPolicy cp_12 = ConnPolicy::data(ConnPolicy::LOCK_FREE, true);
  task1.outPort1.connectTo(&(task2.inPort1), cp_12);
  ...
}
```

Fig. 2: Behavioral code of the superblock that reconstructs the joint-position profile from the joint-velocity profile, automatically generated by the E4Coder tool (top). Step functions of the superblocks mapped into `Task1`, serialized according to the mapping order, consistent with the partial order of execution imposed by the model semantics (bottom-left). Creation of components and activities in `ORO_main()` (bottom-right).