Ρέα: Cyber-Physical Workflows

Barnabás Králik, Dávid Juhász
ELTE PhD School of Informatics
kralikba@elte.hu, juhda@caesar.elte.hu

Abstract. Ρέα (Rea or Rhea) is a workflow language for CPS programming [1]. The design of Ρέα is based on the paradigm of task-oriented programming (TOP). This paradigm has formerly been used to implement classical workflow-based systems using the rich tools and concepts provided by mature lazy functional programming languages. For an in-depth discussion of the basic ideas, see [2]. Ρέα strengthens this paradigm by making the notion of unstable values first class citizens of the framework. These represent the ever-changing values of signals coming from the physical environment.

TOP. Task-oriented programming is a novel programming paradigm for the development of distributed multi-user applications. It builds a purely functional fortress on top of the notion of workflows. Its basic notion is the task. Complex workflows – themselves also being tasks – can be built from them using side effect free combinators. The indivisible subtasks of a workflow are called primitive tasks. Application developers are expected to design most of these primitive tasks and implement them in a platform-specific way. In the case of heavily human-dependent workflows the main vehicle of data flow is the passing of the results of the individual tasks – that is, their stable values. In the case of cyber-physical systems, many tasks have conceptually time-continuously changing observable unstable values.

Tasks. Tasks are abstract descriptions of interactive persistent units of work that have typed values which can change throughout the execution of the task. However, most tasks are expected to produce a final value. Normally, this is a stable value; a result of the encapsulated computation. Sometimes – as a rule of thumb, when some external device fails – the task may ultimately end with an exceptional value. In a cyber-physical setting, the non-computational parts of the system are also modelled as primitive tasks.

On the level of implementation, tasks are actively communicating, distributed processes that adhere to a common protocol. Tasks are computationally pure; the only interference that may happen between them is through external (physical) phenomena. Observe that this kind of impurity does not introduce any additional complexity into the system as sensory data is usually acquired in a completely non-deterministic fashion.

Combinators. While tasks are indivisible, combinators are the composition of subtasks that define the control graph. The combinators themselves also have an associated communicating process which actually provides the same interface as any other tasks. The three basic combinators are: (1) the sequential combinator which combines tasks along the axis of stable values; (2) the controller which combines tasks along the axis of unstable values; (3) and the parallel combinator which combines tasks along the axis of time.

The *sequential combinator* processes stable values; from the viewpoint of the sequential combinator, tasks are transformations on stable values. The sequential combination of two tasks is a task that "applies" the first task to the input value, and then the second task to the

result of the task. The intermediate result can be observed as the sole new unstable value generated by the combinator. Also, unstable values of the subtasks are propagated through.

The *controller* processes and reacts upon unstable values. It combines two tasks of significantly different natures: (1) the task to be controlled, which can be a task of any kind; (2) and a plan which is actually reacts upon the unstable values of the controlled task. The plan is instantiated for each unstable value and yields a tagged value. Tagging a value *unstable*, the controller itself emits the given value as an unstable value. An *undefined* tag makes the controller remain silent and emit no new values. To prematurely end a task, the plan tags the value *stable* which can then be considered the final value of the complete controller construct.

The *parallel combinator* executes an n-tuple of tasks in parallel. It accepts an n-tuple of input values which are then passed one-by-one to the tasks as input values. The value of the combinator will then be the n-tuple of the subtasks' values.

Specificators. Specificators provide a declarative mechanism for imposing constraints on the execution of tasks. For the outside world, they can be considered tasks as well: they have an associated communicating process which provides the same interface as any other task.

An important kind of specificator is the *resource specificator*. Each Péa node has an associated set of resources identified by unique names. Each of the resources provides a set of primitive tasks executable on the given node. Resource specificators constrain the execution environment of a task by requiring the executing node to have some statically defined set of resources. Should it be required, implicit and transparent code mobility will be involved at run time. The resource specificators have one parameter: a set of resource names. Its main variants may (1) execute the task on exactly one matching node; (2) on the matching nodes that are present when the specificator starts; or (3) on the matching nodes that may be present any time during the specificator's lifetime.

Pipes. So far, the data flow graph of a Péa program shadows its control flow graph. However, in some cases, the programming problem to be solved requires communication between distant nodes of this control flow graph. In order to keep the control flow tidy, but still allow these structurally distant nodes to share information, we introduce the notion of pipes. These can be thought of as a shared reactive variable with at most one writer in its lifetime. To assign this writer, we can label a task with a given pipe's name, which we call tapping the task with the pipe. Tapping introduces a new edge from the tapped task toward the pipe. In other parts of the workflow, we can introduce outlets connected to this pipe. From a high-level point of view, these outlets are tasks that replicate the state of the tapped task.

References

[1] D. Juhász, L. Domoszlai, and B. Králik, "Rea: Workflows for cyber-physical systems," in Domain specific languages (DSL2013, selected papers), 2014, accepted for publication.

[2] R. Plasmeijer, B. Lijnse, S. Michels, P. Achten, and P. Koopman, "Task-oriented programming in a pure functional language," in Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming, PPDP '12. New York, NY, USA: ACM, 2012, pp. 195–206.