STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVIII**, Number 2, 2013

WORKFLOW DESCRIPTION IN CYBER-PHYSICAL SYSTEMS

TAMÁS KOZSIK, ANDRÁS LŐRINCZ, DÁVID JUHÁSZ, LÁSZLÓ DOMOSZLAI, DÁNIEL HORPÁCSI, MELINDA TÓTH, AND ZOLTÁN HORVÁTH

Abstract. Cyber-physical systems (CPS) are networks of computational and physical processes, often containing human actors. In a CPS-setting, the computational processes collect information on their physical environment via sensors, and react upon via actuators in order to reach a desired state of the physical world.

In the approach presented in this paper a CPS application is implemented as a hierarchical workflow of mostly independent tasks, which are executed in a distributed environment, and satisfy timing constraints. In certain cases such workflows can be defined from natural language descriptions with the use of ontologies. The structure of a workflow, as well as the constraints put on the constituting tasks, are expressed in a domain-specific programming language.

1. Introduction

There is a growing need for complex controllable distributed systems. Some examples, selected randomly to illustrate the vast diversity of the application domains, are as follows: automated production lines, public transportation with driverless cars, infantry fighting vehicles, robotic surgery and internet-based multi-player augmented reality games. Cyber-physical systems (CPS) are networks of computational and physical processes, often containing

Received by the editors: June 1, 2013.

2010 *Mathematics Subject Classification.* 68N15, 68M14.

1998 *CR Categories and Descriptors.* D3.2 [**Programming Languages**]: Language Classifications – *Applicative (functional) languages, Concurrent, distributed, and parallel languages*.

Key words and phrases. cyber-physical system, task-oriented programming, workflow, timing constraint, domain specific language.

This paper has been presented at the International Conference KEPT2013: Knowledge Engineering Principles and Techniques, organized by Babeş-Bolyai University, Cluj-Napoca, July 5-7 2013.

human actors. In a CPS-setting, the computational processes collect information on their physical environment via sensors, and react upon via actuators in order to reach a desired state of the physical world.

There is a related emerging challenge for system and software development, which is concerned with the multi-faceted (physical, computational, psychological, cognitive), multi-layered (mobile, WLAN, backbone), multi-level (micro-to-macro), distributed computation and communication, from RF MEMS to cloud computing.

We suggest a novel route to address the challenge that we call *CPS Programming*, and which is concerned with the development of a domain specific language (DSL) for cyber-physical systems. The DSL is capable to describe distributed workflows, the timing constraints of the tasks involved, and fault-tolerance mechanisms.

In this paper, we propose a way for defining the computational parts of complex cyber-physical systems as workflows composed from hierarchical, independent building blocks. We have worked out a workflow system, where there are combinators specifically relevant to CPS Programming: time constraints, distribution, fault-tolerance and error-correction can be easily defined by means of them. We also describe the embedding of a domain-specific workflow language in Erlang.

The rest of the paper is structured as follows. Section 2 exposes the concepts for defining cyber-physical systems with workflows. Our workflow-system and its relevance to CPS Programming are revealed in Section 3. Finally, Section 4 concludes the paper.

2. The main concepts

The methodology proposed here splits up an application into modules, each implementing an (orchestrated set of) action(s) that can be executed, often in a reactive manner, in parallel with other actions. The main requirement is that interaction between modules should be minimized. There are two underlying reasons behind this constraint. Firstly, if interaction is complex, then time requirements of testing, as well as bandwidth requirements for real time error correction and reconfiguration will become unbearable. Secondly, unexpected instabilities might emerge because of the complexity and the accumulation of (small) errors in a controlled non-linear system.

We note that interaction with an ongoing process is slow, and the modification of a stable trajectory could be hard and may take time. Consider, for example, that in a given moment, say at time t, some deviation α from planned behavior is detected. If we were to choose a correction module that can correct α, we would ignore that the execution of the correction module begins after a certain delay, the execution itself takes time, and the ongoing process may have side-effects on our correction module.

Hence, in order to meet our constraints, the progress of the application should be monitored, and whenever a failure, or some deviation from planned behavior is detected, a new correcting (sub)goal must be defined. In other words, we notice the deviation, predict its future dynamics, and make a plan to minimize the costs in the future on the top of the ongoing process. We have made the following assumptions: (i) we have a model, (ii) we can carry out model-based prediction, (iii) we can perform long-term cost optimization on the top of any ongoing process by means of modules, and (iv) these modules have minimal (or at least tolerable) side-effects on the ongoing process. This way we may (eventually) cope with the unavoidable delays apparent in a distributed and/or concurrent system.

2.1. **An illustrative example from nature.** Evolution teaches us for the relevance of the cost of interaction [1]. The brain has 10^{11} neurons, but only 10^{14} connections (instead of 10^{22}). Furthermore, the evolved system is built from robust (and complex) modules, but with minimized side-effects. Consider, for example, the mammalian control system [2], which has the following properties.

(1) Control space is divided according to high level tasks (eating, grasping, chewing, defense, manipulation in central space, climbing etc.), only a few may be concurrent at a time, but many (low complexity) combinations are executable in parallel.
(2) Each high level task is divided into sub-tasks; e.g., grasping is divided according to the discretization of the allo-centric 3D space within reach. It thus avoids combinatorial explosion of muscle space and is robust with respect to the huge dimension of body configuration space.

We conclude that module structure should be goal (task) driven and that the minimization of side-effects seems mandatory at least for the mammalian decision making and executive systems.

2.2. **Task and Test Driven Development.** A specific feature of CPS Programming is the software development methodology. The methodology must respect certain constraints, such as the use of a large number of units, stochastic behavior, delays in the execution of modules, system components originating from different sources, and humans-in-the-loop. Moreover, due to dependability requirements inherent in the CPS domain, the methodology should support both testing and verification. The main problem to solve here is the avoidance of combinatorial explosion both in the number of variables, and in the number of test cases.

The number of basic variables is typically large, and the full space scales with the number of variables in the exponent. Even evolution does not have the time to test all structures against one another in such a huge space. As opposed to evolution, our design serves certain tasks, so we are to test only those structures that have the promise of solving the task.

Note that tasks should be defined by decomposing goals, so they express a top-down approach. Testing, on the other hand, concerns the cases determined by the existing variables, so it is a bottom-up process. Unless we can limit the number of variables stepwise, we cannot test our solutions. This leads us to the well-known concept of side-effect free concurrent modules: if we test the individual modules at one level, then they may become our variables (in the sense that we may decide whether to include them) at the next level.

2.3. **Implementation aspect.** The above mentioned methodology is best supported by a domain specific language which is suitable to describe tasks, as well as task hierarchies and related timing constraints. Task-oriented programming [3] (TOP) provides the right paradigm for this DSL. “In TOP, a task is a specified piece of work aiming to produce a result of known type. When executed, tasks produce (temporary) results that can be observed in a controlled way. As work progresses it can be continuously monitored and controlled by other tasks. Tasks can either be fully automated, or can be performed by humans with computer support.”

Complex workflows involving sensors, actuators, humans and communication in a distributed environment can be expressed as compositions of simpler tasks, using predefined and programmer-defined combinators. The DSL can facilitate the introduction of application-specific combinators in the form of higher-order functions. The description of timing constraints should be a central language feature in the DSL.

The technique of language embedding allows the use of a powerful host language in the embedded domain specific language. We are to start with Erlang; Erlang will be the host language. This choice is motivated by certain Erlang features: (i) Erlang is well-suited to programming distributed and concurrent systems and even more importantly, (ii) Erlang’s concept of supervisor processes enables the easy implementation of fault-tolerance mechanisms.

We describe our framework, which is a good basis for bottom-up definition of workflows, in Section 3. The aim of a task-oriented programming DSL is to provide a syntax that resembles to natural language description of workflows, and focuses on the high level structures relevant to domain experts. Since its expressive power can lead to the definition of rather complex systems, and thus testing and verification might suffer from this complexity, the use of the Task and Test Driven Development methodology is fostered. The top-down methodology can be improved by connecting with dialogue systems, and making the generation of workflows from application domain specific ontologies and natural language commands feasible.

3. The CPS Workflow System

In this section, a brief informal introduction to our framework is provided as follows. Section 3.1 classifies the entities of our system as special kinds of tasks. Section 3.2 presents a simple example that already utilizes the basic combinators. Finally, Section 3.3 gives a short description on embedding our DSL in Erlang.

3.1. **Everything is a task.** In a workflow system, tasks are first class citizens. This means that tasks can be arguments and results of other tasks, and, since we focus on a distributed execution environment, they can even be transmitted over the network. Tasks can be composed using "combinators", forming more complex tasks. In this approach, a complete workflow is a task as well.

Different kinds of tasks can be identified according to their function in a CPS workflow. The differentiation of tasks in our system can be seen in Figure 1.

A task consists of two parts: the description of its behavior, and the constraints on its execution. The behavior defines the computation the task performs when launched. The constraints part of a task specify spatial, timing and resource requirements, such as where to execute the task in a distributed environment, or what timeout triggers the cancellation of the task in the case of some failure.

We can distinguish two kinds of tasks. Primitive tasks are the simplest building blocks of workflows: interaction with a sensor or an actuator is a primitive task in a cyber-physical system. Moreover, any computation that is considered atomic according to the problem domain will be defined as a primitive task.

Combinators are tasks building new tasks from existing ones. They can be classified into two groups: a constructor combines the computational parts of its arguments, and a specificator establishes the constraints of a task. More details on combinators will be exposed through examples later on.

3.2. **Basic combinators.** As an illustration, let us consider a simple example, which contains already some of the main concepts of real-world cyber-physical systems, albeit in a small scale. We emphasize four issues regarding a CPS problem here: reading sensors, controlling actuators, operating under specified constraints, and using model-based predictions for correction modules.

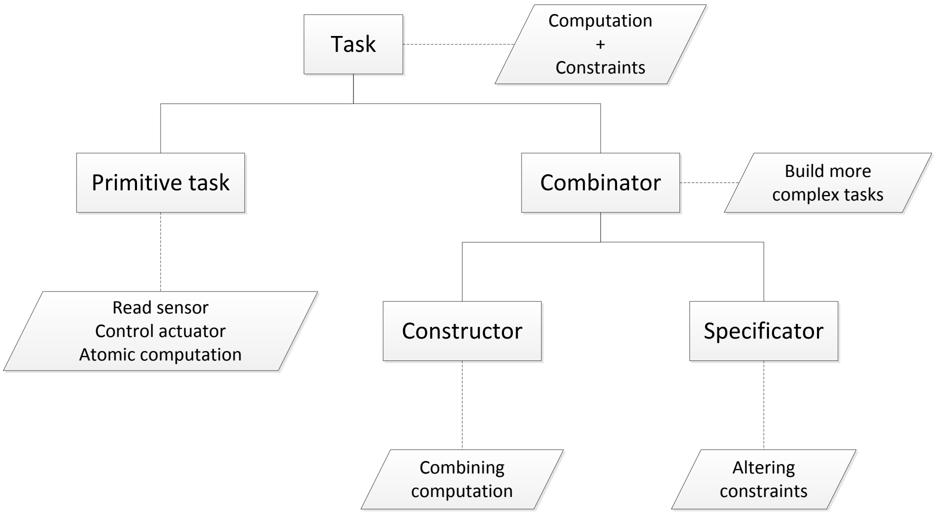

FIGURE 1. Everything is a task in a workflow system

In the example we want to bring water to boil in a kettle: we can switch on a coil to heat the water, we can check the water temperature regularly, and we can switch off the coil when the boiling point is reached. Constraints specify the location where the boiling water is needed, as well as the deadline when the water must reach 100^oC. Furthermore, we want to bring water to boil fault-tolerantly. For example, if the coil breaks down, we want to switch on another one in order to make sure that the water will eventually boil. We have a simple physical model: the temperature of the water is increasing continuously when the coil that heats the water is on. If we observe that this condition is not met, we conclude that either the coil or the thermometer is broken. By using more than one thermometers, and by cumulating sensory data, we can detect breakage of the thermometer. By using more than one coils, electricity to the broken coil can be cut off, and another coil can be switched on. A workflow implementing this functionality is depicted in Figure 2, and the source code of this workflow is presented in listing *kettle.wf*.

This workflow must be run a node connected to a kettle. Having the coil switched on, we start checking the temperature with three thermometers in parallel. We repeatedly read the thermometers, and the stream of sensory data is channeled to the model. We compute the average of the measured temperatures, and check whether the boiling point of water is reached. 97^oC is used due to sensor accuracy. The conditional control structure is provided by

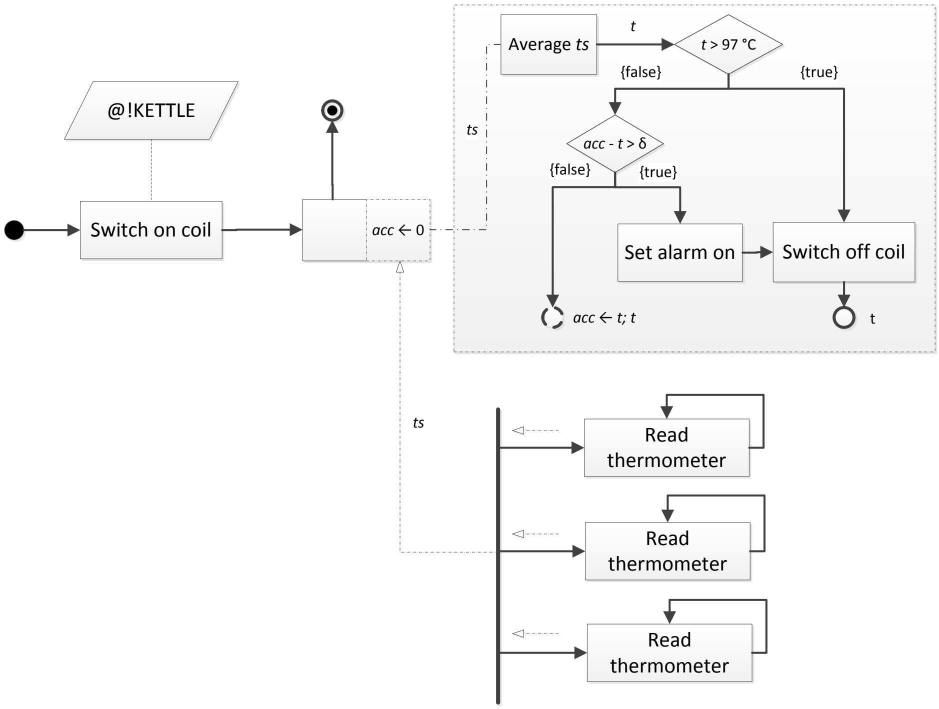

FIGURE 2. Bringing water to boil

the host language our DSL is embedded into. If the boiling point is reached, we switch off the coil, and stop the parallel tasks. If the water is not hot enough, we can compare the temperature against our physical model. If the temperature has not been raised since the last reading of the thermometers, we trigger an alarm and end the workflow. The state maintained by the model is "acc" (which stands for accumulator); it contains the previously read temperature data. Its initial value is 0, and the current temperature is saved into it at the end of each activation when the workflow is not stopped.

Now consider the building blocks of this simple workflow example. The primitive tasks here are the following: read a thermometer, switch on/off a coil and set the alarm. In the graphical representation two special symbols represent the starting and terminating points of the workflow.

What kind of combinators can we observe in this example? First of all, two tasks can be combined sequentially, which means that the second one will be launched when the first one ends. In some cases the result of the first task

```
-module(kettle).

-export(main/0).

main() ->
  Workflow =
    kettle_control:set_coil(on) >>|
    par([rec(kettle_control:read_thermometer(i) >>= fun(t) ->
         continue(t) end)
        | i <-[1,2,3]])
    controlled by fun(acc, ts) ->
      average(ts) >>= fun(t) ->
      if
        t > 97 ->
          kettle_control:set_coil(off) >>|
          return(t);
        acc - t > error_treshold ->
          kettle_control:set_alarm(on) >>|
          kettle_control:set_coil(off) >>|
          return({error, t})
        true ->
          continue({t, t})
      end
    end
  end
  with accumulator 0 @! [kettle],

  execute(Workflow).
```

. kettle.wf

is needed by the second. For instance, after computing the average of the values supplied by the three thermometers, we pass the result to the decision making task. (This is expressed by the `>>=` constructor in the DSL, which binds the value of the first task to a fresh variable – described as an "explicit `fun`-expression" in Erlang.) On the other hand, after switching the coil on, we can start reading the thermometers without passing any values from the first task to the second one. (This is expressed with the `>>|` constructor in the DSL.)

To describe parallel control flow, one can use the `par` constructor, which launches a number of tasks simultaneously. The compound task ends only if all of its components have ended, and its result is a list of the results of the components. Reading sensory data from the three thermometers is implemented with the parallel combinator in our example.

Workflows are often described in the style of reactive programming: the state of a subsystem can be monitored, and the workflow can react on changes of this state. According to our approach, we have a model which can plan an ideal trajectory in the problem space, and decide about error correction when deviation from that trajectory is detected.

Constraints can be associated to tasks, such as the spatial constraint `@!KETTLE` in the example. The specificator `@` can be used to impose requirements on the location where a task must be executed. In our example, we specify that the complete workflow must be launched on a node that is annotated with the "KETTLE" property. Specifying requirements on available resources as well as on user identities and roles is also possible in this way.

An interesting aspect of the `@` combinator is that it ensures implicit code transfer among nodes of a distributed system. Adaptivity of components in a CPS application and autonomy of tasks are fostered by allowing their code to be transfered by the workflow runtime in a transparent way. As an additional consequence, there is no need to deploy all the components of an application on all the nodes of the executing distributed execution environment, a minimal workflow runtime suffices: the code of the tasks can be transfered by the runtime to the appropriate node when needed, and hence the distributed execution environment can dynamically (re-)configure itself.

To make the workflow hierarchical, a special constructor, called a *controller* has been introduced. To understand its semantics, the concept of unstable values must be considered first. When a task ends, the value it results will never change: it is called a "stable value". In the case of sequences, when a task ends, its result is propagated forwards in the control flow as a stable value, but, in addition to this, it is also propagated backwards as an "unstable value". If we have a complex task, it may produce multiple unstable values before it reaches an end, and provides a stable value: its final result. As Figure 3 illustrates, the repeated read of a sensor (or any repeated execution of some task) will also yield a stream of unstable values.

Controllers provide a way to work with unstable values. The construct can be written as "`controlled by`" in the source code, and it binds a name to the unstable values observed. A controller has two tasks associated with it: a "controlled task" (of which the unstable values are observed by the controller), and a "plan". The plan is a task as well, with a very special meaning in the workflow. When a controller is launched, it launches the observed task. Every unstable value propagated from the observed task triggers the execution of the plan associated to the controller. Plans may maintain a state between different activations and can decide to stop the controlled task when a desired goal is reached, or in the case of faults/failures. Finally, when the controlled task ends, its stable value is propagated as the result of the controller.

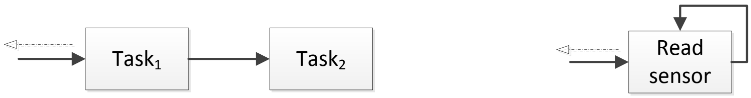

Figure 3. Raising unstable values

Note that error detection and correction can be implemented in the plan of a controller. Therefore, controllers are the major means for model-based prediction and decision making in workflows. This is reflected in the kettle example as well.

3.3. **Behind the curtain – How the DSL is implemented.** As mentioned already in Section 2.3, workflow systems are described by using a software framework, which is in fact a simple programming language embedded into Erlang. It was a design decision that we implement distributed workflow systems in Erlang, since it is one of the most favored programming languages for implementing highly scalable, distributed, reliable and fault-tolerant software systems. In addition, we prefer this language also because it gives us the potential of employing the RefactorErl tool for statically analyzing and transforming the source code.

We chose Erlang despite the fact that it is not extensible, and it is certainly not suitable for DSL embedding. However, we found that the program transformation capabilities of RefactorErl can be easily turned into program translation capabilities; thus, we can extend the Erlang programming language with some key elements required for effective language embedding. The framework and the workflows are implemented in an extended version of the Erlang language, which is translated back to simple Erlang in one single step. The resulting program is compiled and run as any other Erlang application.

Since Erlang does not allow sending functions among different nodes of the network, computations cannot be handed over in an intuitive and effortless way. We developed support for so-called portable functions, realized as a compile-time transformation, which turns anonymous functions into complex data terms representing the computations along with their dependencies attached. On the other hand, the embedding of domain specific concepts into the language is mainly supported by the possibility of defining prefix and infix operators composed of natural language elements. Beside these two main components, we introduced some pieces of syntactic sugar that result in more natural workflow descriptions.

4. Conclusion

A programming methodology for cyber-physical systems has been proposed in this paper. The methodology emphasizes the introduction of modules with limited interaction, constraining the concurrent execution of interfering modules, and the need to avoid combinatorial explosion of test cases for validation.

A domain specific language for describing workflows can facilitate the development of CPS applications. The main concepts of such a language are tasks, combinators and constraints. In the CPS domain timing constraints are especially relevant.

We have presented a domain specific workflow language embedded into Erlang. This embedding provides the constructs of a powerful host language, as well as the superior distribution and fault tolerance capabilities of the Erlang programming model.

Acknowledgement

The research was carried out as part of the EITKIC_12-1-2012-0001 project, which is supported by the Hungarian Government, managed by the National Development Agency, financed by the Research and Technology Innovation Fund and was performed in cooperation with the EIT ICT Labs Budapest Associate Partner Group (www.ictlabs.elte.hu).

References

[1] J. Clune, J.B. Mouret, and H. Lipson. The evolutionary origins of modularity. *Proc. R. Soc. B*, 280(1755), 2013.

[2] M. S. A. Graziano. The organization of behavioral repertoire in motor cortex. *Annual Review of Neuroscience*, 29:105–134, 2006.

[3] B. Lijnse. *TOP to the rescue: Task-Oriented Programming for incident response*. PhD thesis, Radboud Universiteit Nijmegen, 2013. ISBN 978-90-820259-0-3, IPA Dissertation Series 2013-4.

Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary