Rea: Workflows for Cyber-Physical Systems

Dávid Juhász, László Domoszlai, and Barnabás Králik

Department of Programming Languages and Compilers
Faculty of Informatics, Eötvös Loránd University
juhda@caesar.elte.hu, dlacko@pnyf.inf.elte.hu, kralikba@elte.hu

Abstract. Cyber-Physical Systems (CPSs) are distributed systems composed of computational and physical processes, often containing human actors. In a CPS setting, the computational processes collect information about their physical environment via sensors and react upon them using actuators in order to realize a change in the physical world.
In the approach presented in this paper, a CPS application is described as a hierarchical workflow of loosely-coupled tasks whose execution can be constrained with various conditions. We have designed a framework (P*éα*) of a minimal set of combinators implementing features relevant to CPS programming. The details are revealed through an illustrative example defined in our fully functional implementation embedded into an extended version of the Erlang distributed functional programming language.

Keywords: cyber-physical system, task-oriented programming, workflow, domain-specific language

1 Introduction

Cyber-Physical Systems (CPSs) are around us. The information systems that influence our lives so much are getting integrated, and increasingly interact with activities and processes of the real world. There are many application domains where Cyber-Physical Systems have appeared. However, from the programmers' perspective, Cyber-Physical Systems also constitute a well-defined domain. Programming such systems requires a certain set of techniques, and many CPS applications share a certain set of requirements. Therefore, we aim to discover methodologies for developing CPS applications and provide support for CPS programming.

The approach we have taken is based on a recent programming paradigm, task-oriented programming (TOP), in which computations are defined as workflows of simpler computational steps usually called primitive tasks. We designed a hierarchical workflow language (P*éα*), whose features are presented in this paper. The language provides a minimal set of combinators that are relevant to CPS programming.

P*éα* is a domain-specific workflow language the first incarnation of which is an embedding into an extended version of the distributed functional programming language Erlang. We choose Erlang due to its built-in capabilities

of seamless distribution. Moreover, TOP and functional programming makes it able to orchestrate a complex application from loosely coupled building blocks. This seems to be a crucial characteristic for easing the testing and verification of such complex systems.

To demonstrate the capabilities of our framework, a small scale example has been worked out the implementation of which poses all the challenges with which developers of large scale CPS applications have to cope. Besides implementing a workflow application for the example, we have tailored a special piece of hardware to run the application and give a real-world demonstration. Different features of P$\acute{\varepsilon}\alpha$ are revealed through the step-by-step construction of a control application for that example.

We have two contributions presented in this paper:

- We designed P$\acute{\varepsilon}\alpha$, a distributed, hierarchical workflow system for building CPS applications from loosely coupled tasks whose execution can be constrained with various conditions.
- We implemented a fully functional P$\acute{\varepsilon}\alpha$ framework as an embedding into an extended version of Erlang.

The rest of the paper is structured as follows. The problem domain at hand, Cyber-Physical Systems, is summarized in Sect. 2. Section 3 describes the principles of P$\acute{\varepsilon}\alpha$ followed by a review of an illustrative example in Sect. 4. The workflow application controlling the device described there is revealed using the actual syntax of the P$\acute{\varepsilon}\alpha$ implementation embedded into Erlang in Sect. 5. Related work is discussed in Sect. 6 and, finally, Sect. 7 concludes the paper.

2 Cyber-Physical Systems

Cyber-Physical Systems [14] are networks of computational and physical processes, often containing human actors. In a CPS setting, the computational processes collect information about their physical environment via sensors and react upon them using actuators in order to realize a change in the physical world. Some examples, to illustrate the vast diversity of the application domains, are as follows: automated production lines, automated transportation systems, infantry fighting vehicles, robotic surgery, smart home and smart city applications.

Nevertheless, those very different application domains have common attributes raising issues that are to be addressed. Computational devices involved in a CPS are typically embedded devices, i.e. limitations on power consumption and performance have to be considered. There is a network of computational devices working to reach a common goal, which needs an efficient goal-driven distribution of data and computation among those nodes. Physical environment is to be taken into account when defining the behaviour of the system, which also should be able to react in a physical way. To that end, handling sensors and actuators has to be an essential piece of the building blocks upon which a CPS application is built. A special part of the applications' physical environment is the segment of human beings. Making humans able to interact with a smart system in a

comfortable way is to be settled as well. Last but not least, most of the CPSs are critical systems on which even human lives might depend.

There are also other requirements with which CPSs' software must deal, e.g. real-time constraints, robustness, fault-tolerance, failure recovery, adaptivity, safety and security.

Our first step towards answering the aforementioned challenges is P*έα* presented in this paper. Its current implementation addresses the basic questions of CPS development. More sophisticated software features – e.g. failure recovery, adaptivity or a limited scope of timing constraints – could be easily implemented using current features. Other issues – such as precise worst-case execution time and resource consumption estimation – need more research to be done.

3 P*έα* – A language for Cyber-Physical Workflows

In P*έα*, the behaviour of a CPS application is described as a workflow of loosely coupled tasks whose execution can be constrained with various conditions, where tasks are composed by combinators that are applicable to CPS programming. Principles and considerations behind the design of P*έα* are published in [12].

The basics of task-oriented programming are summarized in Sect. 3.1. The details of P*έα* compared to the general principles of TOP are exposed in Sect. 3.2, and a short description on the DSL implementation in Erlang is provided in Sect. 3.3. A more detailed elaboration of P*έα*'s features is provided through an example in Sect. 5.

3.1 Task-Oriented Programming

Task-Oriented Programming (TOP) [18] is a novel programming paradigm for the development of distributed multi-user applications which extends pure functional programming with a notion of tasks and operations for composing programs from tasks. Its four main concepts are as follows:

- *Tasks:* Tasks are abstract descriptions of interactive persistent units of work that have a typed value. Other tasks can observe the *current* value of a task. The observed current value can be of three kinds: (1) the task has no observable value; (2) the current value of the task is *unstable*, it may change in the future; (3) the current value of the task is *stable*, it is the final value of the task.
- *Many-to-many Communication with Shared Data:* When multiple tasks are executed simultaneously, they may need to share data among each other. In TOP, typed abstract interfaces, the so-called *Shared Data Sources* are provided to read, write and update shared data atomically. When one task modifies shared data, the other tasks can observe this change.
- *Generic Interaction:* A TOP framework generates user interfaces *generically* for any type of data used by tasks. This means that the framework can be asked to manage single interactions such as entering, updating or displaying some data, and it takes care of all the related job automatically, e.g.

generating a user interface, client-server communication, state management, etc.
- *Task Composition:* TOP defines a small carefully designed set of core combinator functions from which complex patterns can be constructed. These are: (1) *dynamic* sequential composition, where dynamic means that the subsequent task can be dependent of the current value of some initial task; (2) parallel composition: the simultaneously executed tasks have read only access to the current values of their siblings to be able to monitor each other.

3.2 Tasks and Combinators in P*éα*

A P*éα* *task* corresponds to that of TOP, dynamic sequential and parallel compositions are supported as well. Instead of shared data sources, P*éα* provides the pipe construct for tasks to observe the current value of another task. Note that the functionality of a shared data source can be easily simulated by using pipes. P*éα* has some primitives for interacting with the user via a form-based user interface. The implementation of this might seem rudimentary - but keep in mind that P*éα* is supposed to be used (mostly) in headless embedded systems.

In P*éα*, a constant value can be turned into a task with the function `return`; while a complex computation defined by a host-language function can be transformed into a task by using the function `task_create`. Such tasks are called *primitive tasks*, as they are the smallest building blocks of workflows. More complex tasks, that are considered *workflows*, can be created by combining already defined ones using *combinators*. Besides stable and unstable values, a task in P*éα* can result in a special kind of final value, exception, which stops the execution of subsequent tasks.

The system provides predefined primitive tasks. Two general ones are the following: (1) `delay` blocks for a given amount of time, then results in a special stable value `timeout`; (2) `current_node` returns the name of the Erlang virtual machine it is executed in. GUI operations, pipes and message passing among simultaneously executed tasks have their predefined primitive tasks as well.

Primitive tasks are considered atomic operations, which typically do not have unstable values. The predefined task `show_form`, which handles the user interaction with a GUI form, is an exception to this as it raises unstable values in correspondence with state changes of GUI elements. Nevertheless, the function `task_create`, which is provided for workflow developers to create their own application-specific primitive tasks, supports only the creation of tasks without unstable values.

Combinators that can be used to compose tasks to build more complex ones in P*éα* are shown in Fig. 1. We introduce a graphical representation of the combinators in order to ease understanding of program logic for domain experts. There is a 1-to-1 mapping between elements of the graphical notation and elements of the language - thus, executable code can be generated from such diagrams in a straightforward manner.

The *sequence combinator* simply executes two tasks in a sequence, passing the result of the first task to the second one. It is the only combinator which

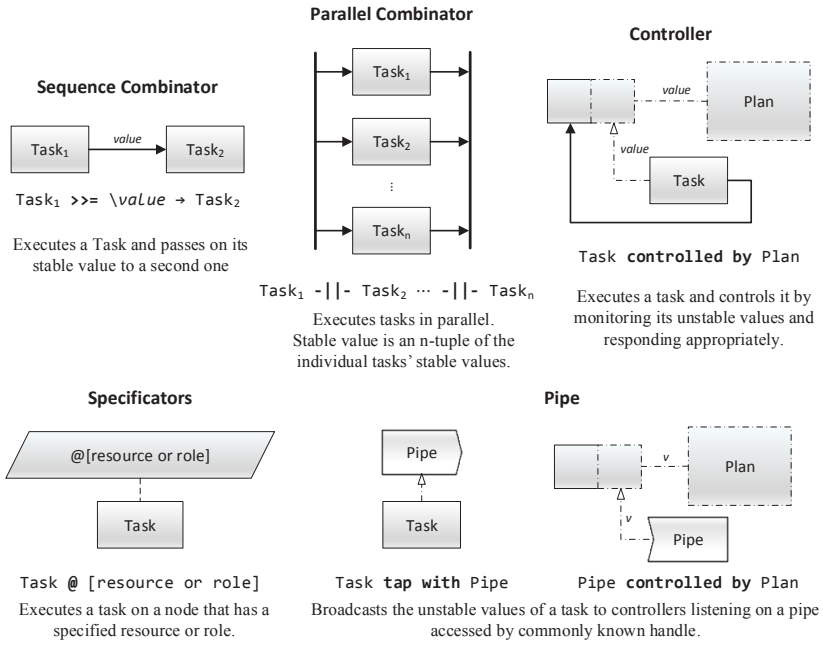

Fig. 1. Combinators in P$\varepsilon\alpha$

raises new unstable values: the inner result of the sequence becomes an unstable value when produced by the first task. All the other combinators propagate or process the unstable values raised by instances of the sequence combinator. While stable values are passed on according to the control flow, unstable values are propagated backward in the control flow graph.

The *parallel combinator* is to use when concurrency is required in a workflow. The combinator operates in the fork-join model: all the subtasks are executed in parallel and the whole construct would finish when all parallel tasks are finished. The unstable and stable values of the parallel combinator is a list consisting of those of the subtasks. A task executed by the parallel combinator is able to observe the actual state of its siblings through a pipe created by the combinator. Moreover, parallel tasks are able to send messages to each other according to the roles associated to them.

A *controller* executes a task and processes its unstable values. Each unstable value raised inside of the observed task triggers the execution of the so-called plan, which is a task with a special result. The stable value of a plan can be of two kinds: (1) an `unstable` result indicates that the observed task can continue and the value resulted by the plan is propagated as unstable value to other tasks observing the controller; (2) a `stable` result means that the observed task is to be stopped and the value resulted by the plan is the result of the controller. If the observed task completes without the controller stopping it, the result of the

controller is that of the observed task. Hierarchies of tasks can be defined by means of nested controllers. Note that the unstable values of a plan are ignored by the executing controller. Though a plan could be constructed with controllers inside, such a practice would lead to workflows of bad design.

P*éα* provides *specificators* to constrain the execution of tasks according to a number of conditions. Currently, specificators defining constraining conditions on the location and time of the execution of a task are built in to P*éα*. The usage of the *resource specificator* is shown in Fig. 1. When executing a task annotated with a resource specificator, the runtime environment ensures that the annotated task is going to be executed on a node which provides the specified resource(s). Note that we can say *role* when talking about a number of resources connected to each other and provided always together, e.g. a node running on a kettle provides the resources *heating elements* and *thermometers*, and is referred to as a node of the role `kettle`.

A *pipe* can be used to observe the state of a task which is in an unconnected part of the control flow graph – namely in a parallel branch of execution. Note that unstable values are propagated backward in the control flow, thus sibling tasks cannot observe each other's state without pipes. A pipe can be used in two ways according to its two endpoints: (1) a task can be tapped with a pipe, in which case the unstable values of the tapped task are propagated through the pipe, and the behaviour of the tapped task does not change locally; (2) the receiving end of a pipe can be observed by a controller, so making it possible for the remote state to be taken into account. Data flow paths unrelated to the control flow can be introduced in a workflow this way.

More details about the combinators and their usage are discussed in Sect. 5.

3.3 Implementation as an Embedding into Erlang

A fully functional implementation of the system is developed as a domain-specific language embedded into an extended version of the Erlang [1] functional programming language. Erlang has been chosen as a host language for the first implementation of P*éα* since it is a widely used programming language for implementing highly scalable, distributed, reliable, and fault-tolerant software systems [6]. Even though having appropriate properties for implementing the designed features, Erlang is certainly not suitable as a host language for embedding DSLs into and does not support code mobility.

To mend the above mentioned flaws, P*éα* workflows are implemented in an extended version of the Erlang language. Programs in this language are translated back to simple Erlang in one single step, then compiled and run as usual Erlang applications. The required transformations are implemented twice with two different software transformation tools separately to compare their capabilities. One of the tools is a fork of RefactorErl [4], which is a static analysis and refactoring toolkit for Erlang. The other one is a standalone tool developed using the Spoofax language workbench [9], which is a general-purpose toolkit for implementing (domain specific) languages.

The combinators of P$\acute{\varepsilon}\alpha$ are implemented as regular Erlang functions, but are provided for workflow developers as operators of the language. Custom operators cannot be defined nor overridden in pure Erlang. However, the extended language gives us a natural way to let arbitrary function names be used in prefix or infix form. Even precedences can be assigned to them. Appropriately exploiting these features, even mixfix operators can be expressed.

Using resource specificators requires the framework to transmit tasks between separate P$\acute{\varepsilon}\alpha$ nodes. Sending tasks between Erlang virtual machines is not trivial. More often than not, a task consists of primitive tasks defined by means of anonymous Erlang functions. Erlang does not support sending such functions out of the virtual machine in which it is defined. To overcome this limitation, we extended Erlang with so-called portable functions. These are supported by a transformation which turns anonymous functions into complex data terms representing the computations along with their dependencies attached.

Further details on the language extension are revealed in [8].

4 Illustrative Example

Having described our problem domain, Cyber-Physical Systems, and the basics of our framework, an illustrative example, the implementation of which poses all the challenges with which developers of large scale CPS applications have to cope, is discussed. The problem and the hardware abstraction layer of the solution are revealed in the current section, while the control logic implemented as a P$\acute{\varepsilon}\alpha$ workflow is presented in Sect. 5.

4.1 Bringing Water to Boil

The example is about an interactive, computer-controlled, networked, safe and fault tolerant kettle. Features include letting the user define when she would like to have hot or boiling water; how hot exactly should the water be; automatic fault detection and correction by using hot spares of hardware components; cutting off heating when water reached the desired temperature and maintaining that level of temperature until water is consumed or unit is turned off.

The kettle is a wireless device connecting to a network. Other devices connected to the same network can monitor and control the kettle. This scenario is shown in Fig. 2. Currently, only one device is allowed to be in control of the kettle at a given time; nevertheless any number of devices could monitor its status.

In that small scale Cyber-Physical System, the control application may be executed on separate devices in a distributed manner – among which there could be embedded systems as well. The kettle itself is an embedded device in such a system. The application measures specific properties of the physical world via thermometers and reacts upon changes via actuators – namely by turning coils on and off.

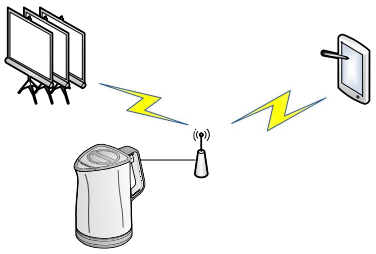

Fig. 2. Wireless access to the kettle

Some details about the hardware and the software environment running on it are exposed in Sect. 4.2; its P*έα* interface – as a set of application-specific primitive tasks - is described in Sect. 4.3. Main parts of the control application are elaborated in Sect. 5.

4.2 The Hardware

Our tailor-made piece of hardware, "The Budapest Kettle", consists of multiple coils and thermometers which are directly controlled by a custom circuit. The controller application runs on a tiny, WiFi-capable router built into the kettle, which is thus able to take part in a distributed network of P*έα* nodes.

The heater tank contains 3 off-the-shelf electric kettles built into one common enclosure along with 3 thermometer chips. The coils and the temperature measurement units are directly connected to a self-designed interface board with relays, data converters and an 8-bit microcontroller. The main computing unit of the kettle is a travel router with a MIPS24Kc processor. The serial communication link provided by the interface board is connected to the router via a Serial over USB adapter.

The firmware of the interface board has also been developed in our lab. The router runs a copy of OpenWRT, an embedded Linux distribution for routers, on which P*έα* is executed by a stripped down version of Erlang R16B01 release.

More details about the hardware and the software code are provided on our website: `http://cps.elte.hu/kettle`.

4.3 Low-Level Control

Without exposing the actual protocol implemented by the interface board, a short summary on the features is given.

- The board can read the current values from thermometers and return a list of raw measurements – voltage levels –, that need to be converted to degrees Celsius according to the specification of the used thermometers.
- The board can be instructed to change the states of relays connected to it, so switching individual coils on and off.

– Last but not least, the board is able to provide information about the project during which the research and development have been carried out.

The interface board has a serial communication port connected to the embedded router via USB using a Serial to USB converter, which yields a serial device on the Linux system. Connecting to the serial device directly from Erlang is not so easy to do because the Erlang virtual machine does not allow blocking functions. Therefore, natively implemented functions (NIFs) have to be used to realize such functionality. There is a publicly available Erlang module, `srly`, providing direct access to serial devices. However, it uses platform-specific features in NIFs that make it incompatible with the architecture of our embedded router's CPU. Thus, communication is performed through TCP, which needs a Serial-TCP bridge running on the router.

For safety reasons, the board automatically cuts off the power supply of the coils when there is no communication for a considerable length of time.

The controller module, `kettle_controller`, is implemented as an instance of Erlang OTP's `gen_tcp` behaviour, which connects to the Serial-TCP bridge on *localhost* and runs as a service in the Erlang virtual machine. A P*éα* node running on a kettle has the role `kettle`, which indicates that the service implemented by the module `kettle_controller` is provided for the workflows executed on the node.

P*éα* applications are able to interact with the kettle via the interface functions exported from the module `kettle_controller`. The two relevant interface functions of `kettle_controller` are wrapped into primitive tasks, that are utilised by the kettle controller application.

– `read_temperature` receives measurements from thermometer chips, converts them to actual temperature values and returns a list of them.
– `set_relays` is a unary function whose argument is a list of 3 logical values indicating for each coil if it has to be provided with power supply.

5 The Control Application

In this section, we define the main parts of an application controlling the kettle described in Sect. 4. The application is revealed step-by-step from a trivial solution to a complex one which has the features listed previously. The different capabilities of our system are also explained along the way.

Only snippets of the full program code are presented in the paper. Fully functional workflow applications are available for download on our web page.

5.1 Simply Bring Water to Boil

The very first version of the control application does not provide any sophisticated features, it only brings the water to boil. All the coils are switched on at the beginning, then switched off when the water temperature has reached its boiling point.

```
kettle_plan() ->
 fun!(V) ->
  case V of
   #task_value{value = {ok, Ts}} ->
    effective_temperature(Ts) >>= fun!(T) ->
    Error_Threshold = 3,
    if
     T > 100 - Error_Threshold ->
      kettle_controller:set_relays([off,off,off]) >>|
      stable({done, T});
     true ->
      unstable(T)
    end
  end end
 end.

temperature_reading() ->
 iterate(fun!() ->
  kettle_controller:read_temperature() >>= fun!(Ts) ->
  continue(Ts) end end).

control_workflow() ->
 kettle_controller:set_relays([on,on,on]) >>|
 temperature_reading() controlled by kettle_plan().

main() ->
 execute(control_workflow() @! [kettle]).
```

Code 1.1. Simply bring water to boil

The Trivial Solution. The entry point of the workflow listed in Code 1.1 is the function `main`, which executes the workflow defined in `control_workflow` on a Pέα node having the role `kettle`. On such a node, the `kettle_controller` module is available, thus kettle-specific operations of the control application can be executed. The remote execution combinator `@!`, which is a resource specificator, ensures that exactly one kettle will be selected to execute the workflow.

The control workflow consists of two tasks combined with a sequential combinator which discards the result of its first component, i.e. it is not used by the rest of the computation. First, all three coils are switched on. After that, the actual temperature of the water is monitored. The latter is implemented by a controller which executes `temperature_reading` controlled by `kettle_plan`.

As the temperature must be measured continuously in order for the application to be able to cut the power when the water starts to boil, measuring is implemented as an iterative task. The iterative combinator `iterate` is provided with a nullary function which defines a workflow reading the actual temperature and returning it wrapped into a `continue` value. The resulted value indicates

```
temperature_reading() ->
 iterate(fun!() ->
  kettle_controller:read_temperature() >>= fun!(Ts) ->
  delay(timer:seconds(1)) >>|
  continue(Ts) end end).
```

Code 1.2. Delaying subsequent sensor readings

that the workflow belonging to `iterate` is to be executed again. In that workflow, the original variant of the sequential combinator is used, which makes it possible to use the result of the first task later on. Note that temperature values returned by `read_temperature` are propagated towards the plan of the controller as unstable values.

The plan in this case is very straightforward. It receives the actual temperature reading as a task value and lets the application run until the temperature has reached the boiling point. The hardware interface returns a list of readings from all of the temperature sensors. This list is then transformed into one value by the task `effective_temperature`. That task can compute the average of the values or perform more sophisticated computations with the list, e.g. discarding extreme values or sudden changes from a given sensor. In general, such design decisions are made by domain experts.

Having an effective temperature computed, there is one simple question to answer: is the water boiling yet? If the temperature is within the range of the boiling point – considering an error threshold –, the coils are switched off and a `stable` value is returned. Otherwise, the water is deemed to require more heating. Note that returning a `stable` value indicates that the controlled task is to be ended, and the workflow would continue its execution with subsequent tasks.

In our example, as there are no subsequent tasks after the controller in the workflow, the kettle control application is terminated.

Slowing Down the Sampling Rate. In the previous version of the workflow, temperature is read continuously, which yields an overflow of sensory data. Instead, the sampling can be made coarser by delaying subsequent sensor readings. Again, it is up to domain experts to decide the right sampling rate. In the case of the kettle, a 1 Hz sampling rate is sufficient to safely detect water temperature.

The extended version of task `temperature_reading` is listed in Code 1.2. In this version, an extra task is put between the sensor reading and `continue`. One of the predefined primitive tasks in P*éα* is `delay`, which blocks for the given amount of time, then returns a special value, `timeout`.

Note that higher level timing combinators, e.g. setting a timeout for tasks and rerun tasks periodically, can be easily defined by the means of `delay` and basic P*éα* combinators.

```
form("kettle monitor",
 [label(l1, "Temperature:"), label(temp, ""),
  label(l2, "Coils:"), label(heat, ""),
  label(comment, "")], []).
```

Code 1.3. Declarative description of the monitor form

Note that sensory data is propagated to the associated plan as soon as sequential combinators raise unstable values. The set delay only postpone later readings.

5.2 Monitoring Status

The next step is to somehow visualise the status of the kettle. This is achieved in a simple way, by using predefined P*éα* GUI form components.

Creating a Monitor Form. First of all, a form is to be defined. P*éα* supports the declarative description of forms, as shown in Code 1.3. The `form` function needs three arguments: the title of the form, a list of the form components and a list of actions corresponding to buttons in the form — the rest is handled by the framework.

In our example, the actual temperature and status of coils are committed to the form with some comments, e.g. indicating that the water has reached the desired temperature, or that something went wrong.

Having a form description generated by `form`, a form instance must be created on a capable P*éα* node and then shown on that particular node. The instantiation of forms is performed by the task `create_form`, which creates a form on the local node and returns a descriptor which can be used anywhere inside the P*éα* network to reach the instance. A form instance can be shown, updated and stopped.

Let us consider the beginning of the life-cycle of a form as presented in Code 1.4. A form is created on a node with role `iot_monitor`, then displayed by `show_form`, which is run with `control_workflow` in parallel. That is because `show_form` is blocking until the form is stopped, closed or one of its actions is selected by clicking on a button. Nevertheless, the form descriptor is now passed to `control_workflow` as it is needed to update the components of the form.

A task updating temperature value in a form can be seen in Code 1.5. The temperature value is converted into text, with which an update request is generated using `form_update`. The update then can be realized by `update_form` fed with a form descriptor and an update request. A form can be updated before and while it is shown, but not afterwards.

Implementing other form updates are left as an exercise to the reader.

For this version of the workflow to be ready, `kettle_plan` also needs some modification to keep the information on the monitor form up-to-date: a few

```
kettle_workflow() ->
 create_form(monitor_form()) @! [iot_monitor] >>= fun!(Form) ->
 par([
  show_form(Form),
  control_workflow(Form) @! [kettle]
 ]) end.
```

Code 1.4. The new entry point of the control workflow

```
temperature_writer(Form) ->
 fun(T) -> task_create(fun!()->
  Text = io_lib:format("~p", [T]),
  Update = form_update([label_update(temp, Text)]),
  update_form(Form, Update)
 end) end.
```

Code 1.5. Writing temperature to a form

update requests for the values on the form have to be inserted in certain points; and the form is also to be stopped just before returning the final `stable` value from the plan. This latter can be done by using the primitive task `stop_form` fed with the form descriptor.

Note that the remote execution combinator `@`, which is an other kind of resource specificator beside `@!`, executes an instance of a task on each such node that provides the required resource(s). Thus, the task of Code 1.6 results in a list of form descriptors by creating a monitor form on each `iot_monitor` node connected to the P*éα* network. Maintaining a number of forms would require only a slight extension of the workflow to issue update requests for all of the forms instead of just one of them.

The Form Is Important. The definition of a monitor form does not contain any actions (see Code 1.3), so `show_form` will never return because a button was pressed. Nevertheless, the user is able to close the form by clicking the 'x' at the top of the tab (see Fig. 3). Although one form cannot be forced to remain open against the user's will, the user can be forced to deal with the form by recreating it over and over again. As the status of the kettle is considered to be important, the user will be forced to keep an eye on it when executing further revisions of the workflow application.

```
create_form(monitor_form()) @ [iot_monitor].
```

Code 1.6. A task creating a number of monitor forms

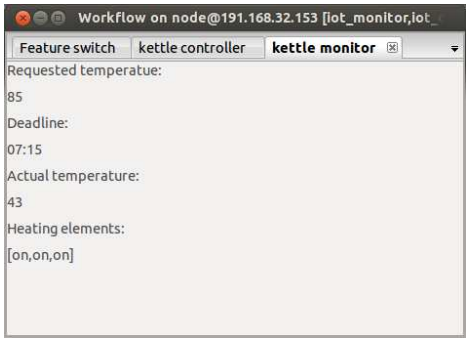

Fig. 3. The final shape of the kettle monitor form

Once a form somehow ends (closed, stopped or a button has been clicked), its life-cycle is over and there is no way to do anything with it. A new form has to be instantiated according to the same description and the new instance is to be shown next time. This scheme can be implemented using the iterative combinator, as can be seen in the second function in Code 1.7.

In this case, `control_workflow` is not executed in a common parallel construct with a `show_form`, but with the iterative task taking care of recreating the monitor form if needed. Moreover, that task has the role `monitor`, which makes other tasks executing in the same parallel environment able to send it messages by using the predefined primitive task `send_role_msg`. Utilising message passing facilities of P*éα* is necessary because the dynamically changing form descriptor is required for updating the status information.

Inside the iterative task, the descriptor of the form actually displayed is available, thus the `form_updater` task is able to directly issue updates on that form. The implementation of `form_updater` consists simply in iteratively receiving messages by means of the predefined task `receive_msg`, and executing the proper form updaters that were already used in the previous version of the application. On the other side, form updaters are replaced with tasks sending appropriate messages to the role `monitor` from `control_workflow`.

The plan controlling the parallel task inside the iterative combinator seems a bit complicated because parallel combinators are propagating a list of task values inside an unstable value. Thus, pattern matching against a list of an appropriate length is necessary. The result of a displayed form is of the record `form_value`, which contains an action and the final values of the form components. The action can be one of the user-defined actions belonging to the form or one of two special values: `closed` and `stopped`. The former indicates that the form has been closed by the user, whilst the latter indicates that the form has been stopped by the application itself. In the case of `monitor_form_plan`, the iterative task is to be rerun if the user closed the form, i.e. it returns a `continue` value if the resulted action is `closed`.

```
monitor_form_plan() ->
 fun!(#task_value{value = [#task_value{value = FV}, _]}) ->
  case get_action(FV) of
   closed ->
    stable(continue(FV));
   stopped ->
    stable(FV);
   _ ->
    unstable(FV)
  end;
  (V) -> V
 end.

controlled_monitor_form() ->
 iterate(fun!() ->
  create_form(monitor_form()) @! [iot_monitor] >>= fun!(Form) ->
  par([
   show_form(Form),
   form_updater(Form)
  ]) controlled by monitor_form_plan()
  end
 end).

kettle_workflow() ->
 par([
  {monitor, controlled_monitor_form()},
  control_workflow() @! [kettle]
 ]).
```

Code 1.7. Recreating the monitor form iteratively

Note that form updates are causing unstable values of the record `form_value` raised with action `updated`, which is why the last branch is present in the conditional expression. Also note that different instances of the monitor form might appear on separate `iot_monitor` nodes as the node is selected before each instantiation. One could force the form to be recreated on the same node by two means: (1) moving the remote execution operator outside of the iterative combinator or (2) selecting an `iot_controller` node outside of the iterative construct and execute `create_form` on that particular node every time by using the remote execution combinator. The difference between the two alternatives is whether the whole iterative task or only the form instantiations are to be executed on the selected node.

5.3 Detecting Failures

The application is now able to report its status – at least the actual temperature of the water and whether each one of the heating elements is on or off. But the application is unable to detect any kind of failures; for example, the application is going to wait indefinitely for the water to boil if only broken heating elements are switched on. In order to detect erroneous situations, the workflow must be aware of its *history*, that is, whether some events and states occurred recently. Such functionality can be implemented in P$\acute{\varepsilon}\alpha$ by using a controller with an *accumulator*. To detect broken coils, only two parts of the workflow need to be changed, whose new version can be found in Code 1.8.

In the function `control_workflow`, the controller construct is extended with an accumulator using the construct (`with accumulator`) with an initial value of 0.

Now the plan has two arguments, the first one is the actual value of the accumulator and the second is the unstable value propagated from the controlled task. In the case of returning an `unstable` result, two values have to be defined: a new accumulator value and a value which is to be propagated as an unstable value upwards in the controller hierarchy.

As can be seen in the code listing, the plan sends messages in order to inform the form updater running under the role `monitor` about the actual temperature.

The definition of the task `shutdown` is not listed in Code 1.8 as its implementation is really simple: it turns all the coils off and instructs the form updater to write its actual argument into the comment label of the monitor form. Note that the form updater here also takes care of stopping the form after setting the comment.

The logic implemented by the plan is able to detect when the water temperature is dropping, thus determining if coils are broken. The accumulator value is always set to the maximum of the previous value and the actual reading in order to prevent the water from cooling down slowly unnoticed, which could otherwise happen when the temperature difference of subsequent measurements is under the value `Error_threshold`.

```
kettle_plan() ->
 fun!(Old, V) ->
  case V of
   #task_value{value = {ok, Ts}} ->
    effective_temperature(Ts) >>= fun!(T) ->
    send_role_msg(monitor, {temp, T}) >>| task_create(fun!() ->
    Error_Threshold = 3,
    if
     T > 100 - Error_Threshold ->
      shutdown("Water boiled.") >>|
      stable({done, T});
     Old - T > Error_Threshold ->
      shutdown("Heating element broken!")>>|
      stable({error, T});
     true ->
      unstable(max(Old, T), T)
    end
  end) end end
 end.

control_workflow() ->
 change_heating_element_status([on,on,on]) >>|
 temperature_reading()
  controlled by kettle_plan() with accumulator 0.
```

Code 1.8. Control plan mainaining an accumulator

```
control_form(T, D) ->
 form("kettle controller",
  [label(l1, "Temperature:"), textfield(t, T),
   label(l2, "Deadline:"), textfield(d, D)],
  [action(ok, "Ok"), action(cancel, "Cancel")]).
```

Code 1.9. Declarative description of the control form

5.4 The Human in the Loop

The application up to now is only slightly interactive, as it can show its status to the user, but the user cannot influence the behaviour of the kettle. The next version of the workflow allows the user to parameterise via a form.

The new form has two input fields: temperature and deadline. In the example, only the value from the former is used, however the deadline could also be similarly utilised with further modification of the controller plan.

Creating a Control Form. The definition of the control form is listed in Code 1.9. The form has two input fields, whose initial values are given as arguments of the function, and two actions to indicate whether the actual input values have to be submitted to the control plan or reverted to the most recent ones. The form is to be recreated every time it ends until the workflow is finished, which is easily done with an iterative task.

First, let us consider the changes needed in the already familiar parts of the application. On one hand, the monitor form should reflect the current user values registered by the application. This is easily achieved by putting two new labels on the form and extending the form updater with a new kind of message. On the other hand, the control logic must go through deeper changes.

The start of the workflow can be seen in Code 1.10. The initial user request (`T` for temperature and `D` for deadline) is set programatically in the first line, but the control form could be executed to obtain a real input instead. First of all, notice that the plan monitoring the temperature also must monitor input values to be aware of the user's current wishes. The task measuring the temperature could be moved inside a parallel construct along with a task managing the control form, i.e. inside `control_workflow`; but that would cause the whole user control logic to run on the kettle, while the control form must be present on some other `iot_controller` node. In order to avoid that pitfall, the task responsible for the control form is combined with the same parallel combinator than the monitor form and the control logic.

Now the values coming from the user must be passed to the control plan, for which the pipe construct is to be used. A pipe can be created with the predefined primitive task `pipe` and must be destroyed with `destroy_pipe`. Between the execution of those two tasks, the created pipe can be used for propagating values between tasks that are residing in parallel parts of the same workflow. Task values

```
{T, D} = {97, ""},
pipe() >>= fun!(P) ->
par([
 {monitor, controlled_monitor_form()},
 {control, controlled_control_form(T,D,P) @! [iot_controller]},
 control_workflow(T, D, P) @! [kettle]
]) >>|
destroy_pipe(P) end.
```

Code 1.10. The task starting up the application

normally are propagated according to the control flow: stable values forward and unstable values backward. Pipes can be used to open one-way tunnels between parallel control flow paths to propagate task values.

On the receiving side, a pipe behaves very similarly to usual tasks as it raises unstable values. However, it would never end on its own. The usage of values propagated via a pipe is shown in Code 1.11. The pipe is used as any normal task: it gets executed in parallel with temperature reading. The plan `pipe_filter` is only turning form values with an action other than '`ok`' into `novalue`, so making the implementation of `kettle_plan` simpler.

Instead of the simple parallel construct used previously, a special form of it is utilised in `control_workflow`. Controlling a parallel construct may be cumbersome when determining which component of the parallel construct triggered the raise of a new list of values. In the case of the example at hand, different things have to be done according to whether the pipe or sensor reading gave a new unstable value. For such situations, P$\acute{\varepsilon}\alpha$ provides a parallel combinator with a tightly coupled monitor whose plan is fed with a triplet instead of a list consisting of recent task values. The triplet provided for the plan consists of (1) an index, (2) the kind of the third component of the triplet and (3) a value coming from the task corresponding to the index. Remember that a task's value can be of three kinds: `unstable`, `stable` and `exception`.

Now, the accumulator belonging to `kettle_plan` is a three-tuple: the recent temperature extended with the currently requested temperature and deadline. The implementation of the plan is straightforwardly derived from its previous version. Some clarification is required only when processing a form value supplied by the pipe. The action and new user input is taken out from the form value in the first line of the corresponding branch. Then user input must be converted from strings to numbers. Note that the action is ensured to be '`ok`' at that point by `pipe_filter`.

Now let us see the iterative task handling the control form in Code 1.12. The implementation is very similar to that of the monitor form, the only main difference is the usage of a pipe. On the side where values are issued, a pipe can be connected to a task using the `tap with` operator. A task tapped with a pipe behaves exactly as a normal one; tapping is completely transparent.

```
kettle_plan() ->
 fun!({RT, RD, Old}, V) ->
  case V of
   #task_value{value = {1, unstable, {ok, Ts}}} ->
    effective_temperature(Ts) >>= fun!(T) ->
    send_role_msg(monitor, {temp, T}) >>| task_create(fun!() ->
    Error_Threshold = 3,
    if
     T > RT - Error_Threshold ->
      shutdown("Water temperature at the desired level.") >>|
      return(stable({done, T}));
     Old - T > Error_Threshold ->
      shutdown("Heating element broken!") >>|
      return(stable({error, T}));
     true ->
      return(unstable({RT, RD, max(Old, T)}, T))
    end end) end;
   #task_value{value = {2, unstable, #form_value{} = FV}} ->
    {ok, T, D} = get_action_and_values(FV),
    {T2, D2} = {convert_temperature(T), convert_time(D)},
    send_role_msg(monitor, {request, T, D})>>|
    return(unstable({T2, D2, Old}, Old));
   _ ->
    V
  end
 end.

control_workflow(T, D, Pipe) ->
 change_heating_element_status([on,on,on]) >>|
 controlled_par([
  temperature_reading(),
  Pipe controlled by pipe_filter()
 ], kettle_plan(), {T, D, 0}).
```

Code 1.11. Control logic of the kettle

```
controlled_control_form(T, D, Pipe)  ->
 iterate(fun!({T, D}) ->
  create_form(control_form(T, D)) >>= fun!(Form) ->
  par([
   show_form(Form) tap with Pipe,
   receiver()
  ]) controlled by control_form_plan(T, D)
 end end) with accumulator {T, D}.
```

Code 1.12. Iterative task for handling user input

```
to_integer(S) ->
 case string:to_integer(S) of
  {error, R} -> throw({to_integer, R, S});
  {T, _} -> T
 end.

convert_input(FV) -> task_create(fun!() ->
 {A, T, D} = get_action_and_values(FV),
 case A of
  ok ->
   {ok, {convert_temperature(T), convert_time(D)}};
  _ ->
   {A, {}}
 end end).

get_and_convert_input(Form) ->
 show_form(Form) >>= fun!(FV) ->
 convert_input(FV)
 end.
```

Code 1.13. Converting user input

The task running in parallel with the form waits for one message indicating that the form should be stopped. When that message is received, the plan controlling the parallel construct returns a non-continue stable value. Also note that the accumulator here belongs to the iterative task in order to access the most recent user input when recreating the control form. An iterative task maintaining an accumulator must be built from a unary function whose argument is the actual value of the accumulator. The initial value can be defined with the operator `with accumulator`, just like in the case of a controller.

Dealing with Erroneous Input. Converting text typed by the user inside the control plan is not good practice. There are many theoretical reasons for that. The most practical one is that the user could not be informed about an erroneous input because of the pipe being unidirectional. In the following revision of the workflow, the conversion of user inputs is moved from the control plan to a task executed immediately after the input is over.

The workflow needs to be modified at three points. Task `show_form` is to be replaced with `get_and_convert_input` of Code 1.13 in the iterative task related to the control form. Then `pipe_filter` and `kettle_plan` is to be adjusted to the value format resulting from `convert_input`. Finally, the plan controlling the control form is to be modified to handle exceptions.

The input has to be converted only when the action is '`ok`', and can be ignored otherwise. The implementation of the two converter functions are not interesting, but note that they utilise the function `to_integer`. If the given string cannot be

```
fun!(V) ->
 case V of
  % other branches
  #task_value{value = [#task_value{stability = exception},_]}->
   stable(continue({T, D, "Wrong format!"}));
  _ ->
   V
 end
end.
```

Code 1.14. Pattern matching against an exception

converted to an integer, an exception is thrown. Exceptions are wrapped into a task value of kind exception automatically as long as they are thrown inside a task. The plan controlling the form then can catch the exception in Code 1.14. In this case, the form is extended with a new label for giving feedback to the user about errors, and the accumulator tuple also has a third slot for storing the text of that label.

It Could Be Shut Down. So far, the kettle can be shut down only by the application itself, however the user must be able to stop it at any time. Therefore, make the control form is extended with a new action, '`shutdown`'.

From now on, the plan controlling the form is to return a non-continue stable value when the form returns with the action '`shutdown`', thus keeping the form from being recreated again. On the receiving side of the pipe, some simple modifications are also in order. The plan `pipe_filter` must keep values triggered by the action '`shutdown`' also intact, besides the ones triggered by the '`ok`' button. Then, the control plan has to check whether the value coming from the pipe is a new user input or a shutdown request. In the latter case, task `shutdown` is to be executed to end the workflow.

Once again, these modifications are left as an exercise to the reader.

5.5 Keeping the Water Warm

The last feature of the kettle allows the water to stay warm until it is consumed; this latter fact being indicated by clicking the '`shutdown`' button.

Up to now, the workflow used to end when the temperature reached the desired level. Now, it will follow a different plan which maintains the temperature and falls back to the heating plan if the user changes their mind and sets a higher temperature. Only the control logic requires modifications, other parts of the workflow remain the same.

Changing Plans on Demand. We need to find a way to change plans on demand. A straightforward-looking approach would call for creating a complex

```
kettle_status_reading(Pipe, ControlFun, InitAcc) ->
 controlled_par([
  temperature_reading(),
  Pipe controlled by pipe_filter()
 ], ControlFun, InitAcc).

heating(Pipe) ->
 recent_piped_value(Pipe) >>= fun!({A, I}) ->
 case A of
  shutdown ->
   return({shutdown, nil});
  _ ->
   {T, D} = I,
   change_heating_element_status([on,on,on]) >>|
   kettle_status_reading(Pipe, heating_plan(), {T, D, 0})
 end end.

control_workflow(Pipe) ->
 iterate(fun!() ->
  heating(Pipe) >>= fun!(Res) ->
  case Res of
   {done, T} ->
    keeping_warm(T, Pipe);
   _ ->
    return(Res)
  end
 end end).
```

Code 1.15. Changing of plans

plan which starts with deciding which scenario is active actually: heating the water or maintaining its temperature. The accumulator is to be extended with a flag indicating the actual scenario in this case. The plan itself would consist of a `case` expression with a number of patterns, which comes obviously with some thinking.

However, a more verbose approach is presented here to discuss other issues. The relevant parts of the revised implementation are shown in Code 1.15.

The task running on the kettle is defined by `control_workflow` as an iterative task. It starts with `heating` after which `keeping_warm`, a task maintaining the user-defined temperature, would be executed if the heating was successful. That second part can result in a `continue` value if the water needs more heating due to the user having changed the desired temperature to a higher value. Otherwise, a non-continue value would be returned eventually.

The two tasks, `heating` and `keeping_warm`, are very similar, thus only `heating` is revealed in the provided code snippet. The only difference is in the second branch of the `case` expression: in the task `keeping_warm`, there is no need

for turning the coils on, and, of course, a different plan with a proper initial accumulator value is to be passed as actual argument to `kettle_status_reading`.

There is an issue, however, that must be addressed in this approach due to starting separate controllers, one in `heating` and another one in `keeping_warm`. After one controller is ended and before the next one is started, there is a tiny period of time in which no controller is monitoring the pipe and values propagated by it could be missed. The predefined task `recent_piped_value`, which results in the value that has been most recently gone through the pipe, is to be used in order to mitigate the impact of such unfortunate circumstances.

Note that, in this case, the task `convert_input` requires the most recent user input with each '`cancel`' action, thus being able to set the initial accumulators of control plans.

The plan for `heating` is exactly the same as in the previous versions of the workflow. The other plan is a bit more complicated. Its detailed implementation is not presented here, but we provide a short discussion on it. Its implementation consists of two main cases:

- If new user input is received, there are three different scenarios: (1) if the requested temperature is below of the current one, the controller needs to update its accumulator; (2) if the requested temperature is set to a higher value, the iterative task is to be restarted to heat up the water; (3) in case of a '`shutdown`' action, the plan returns a stable value which ends the whole application.
- There are also three different scenarios when the execution of the plan is triggered by a new temperature measurement: (1) if temperature is higher than necessary, coils have to be turned off; (2) if temperature is below of the desired level with a given threshold, the coils have to be switched on; (3) if the coils are already on, a hardware failure can be detected just like earlier, in which case the plan results in a stable value ending the whole workflow.

Stop Wasting Energy. The final revision of the application is able to shut down the kettle if there was no user interaction for a long time while maintaining water temperature. This needs only the modification of `control_workflow`, in which timing constraints can be defined in an elegant way by using P*éα* combinators. The new version of that task is presented in Code 1.16.

Note that the combinators used in this revision to define timing constraints are predefined in P*éα*, but could be implemented in a couple of lines with the help of the primitive tasks and combinators mentioned in the paper.

The combinator `or after` lets `keeping_warm` run for 5 minutes. If the task does not end within the given time, the combinator stops it and executes the other task provided after the `do` operator, i.e. the sequentially combined task would be executed after 5 minutes. A warning message is sent to monitor form, then `keeping_warm` is executed again. That second running of `keeping_warm` is to be timed out by the combinator `timeout after` after another 5 minutes. That timeout will eventually end the workflow as it results in a value which is not wrapped into a `continue`.

```
control_workflow(Pipe) ->
 iterate(fun!() ->
  heating(Pipe) >>= fun!(Res) ->
  case Res of
   {done, _} ->
    keeping_warm(Pipe) or after timer:minutes(5) do
    send_role_msg(monitor, {cmt, "Hot water is ready!"}) >>|
    keeping_warm(Pipe) timeout after timer:minutes(5);
   _ ->
    return(Res)
  end
 end end).
```

Code 1.16. Keeping the water warm for a limited time

It is also noteworthy that such timeouts could be implemented inside the plan which controls `kettle_status_reading` in the task `keeping_warm`. In that case, however, the high level control structure would be less clear. Elapsed time would have to be computed and meticulously kept track of within the plan by passing it around explicitly.

6 Related Work

The design of P*έα* is based on the principles of task-oriented programming, more specifically on the iTask System [18]. As the domain on which our research is focused is different from that of iTasks, P*έα* implements a modified set of TOP principles to fit the requirements of Cyber-Physical Systems. For example, we do not really need to generate user interfaces automatically. However, letting the system operate in a distributed manner is first principle for us.

Workflows are used for modelling and organizing business processes [11] for a very long time, because such graphical tools are easy for managers and other non-programmer persons to understand. Besides programming CPSs with P*έα*, our aim is to provide a tool for non-programmer domain-experts to create their very own applications easily. Therefore, ideas worked out for widely used workflow languages are also to be considered as possible extensions for the front-end of our system.

Reactive programming has recently gained popularity in developing event-driven and interactive applications [2]. P*έα* has basic primitives that support reactive programming, and notions of functional reactive programming (FRP) can easily be expressed.

P*έα*, in fact, makes it easy to express concepts of FRP. If we drop the notion of stable values outside plans, we would get a system highly similar to FRP. The matching operation of function composition would then be the controller; the matching notion of signals would be that of streams of unstable values.

There are many different approaches and tools to model, design and program Cyber-Physical Systems. The current and planned features of P*éα* are implemented by some of them to some extent. However, none of them integrates a sufficient set of features to build and orchestrate a comprehensive application connected to different cyber-physical domains. Some of the existing tools are dedicated to particular vertical domains; others need not only engineers but experienced software craftsmen to utilise their capabilities. In fact; both of these are true for most contemporary toolkits. The emerging need for a comprehensive tool which integrates the orthogonal design concerns of Cyber-Physical Systems is well known [20]. One way to ease the actual situation is defining design methodology in terms of existing tools and techniques. A thing which is more or less done already, for example [5] contains the very high-level principles of such an integrated methodology. Nevertheless, that combined design apparatus would be better used to identify the different aspects of the CPS design process and then implement one consistent tool supporting those separate concerns.

The main industrial parties also have their standardized methodologies for systems design, which nowadays can be considered as CPS design standards. For example, one of the standards of automotive industry is EAST-ADL [3] describing the different concerns have to included in any design documents. Nevertheless, implementation issues of the designed systems have a separate standard, AUTOSAR [21].

The Orc Programming Language [10] is advertised as a tool for CPS applications which heavily involve human interaction. The language is based on a concurrency calculus of the same name, extended with a functional core. The language can be used to distribute computation among many nodes dynamically, which truly makes it suitable for CPS programming. However, it supports features that make the language impure and hard to reason about. Compared to P*éα*, Orc does not support reactive programming; user interfaces can be defined only with external tools involved; and the implicit parallelism inherited from the base calculus, which is present throughout the language, makes it hard to understand its semantics without significant background knowledge.

One way to create cyber-physical applications quickly based on existing services is writing glue code for those services. A tool that makes gluing web services together possible is described in [19]. The paper proposes two ways to define the glue application: writing Python code, which needs a considerable body of knowledge on programming in Python; and defining the glue logic in a 2D tabular workspace, which is also not so straightforward according to the experience of the authors of this paper. The system is called event-driven, yet it needs manual synchronisation of each event in the glue application. Moreover, the control logic itself is to be developed as a web service.

There is a workflow engine, ERWF [7], with high-level goals similar to ours. The research is aiming at creating a framework for describing the computational part of user-centric Cyber-Physical Systems as workflows and execute them on embedded systems in a real-time manner. Despite the similar vision, the implementations could not be more different. ERWF is implemented using the Real

Time Application Interface of Linux, which makes the system able to execute tasks taking time constraints into account – a feature that lacks from P$\acute{\varepsilon}\alpha$ when this paper is being written. It is noteworthy that the decision about task execution is based on probabilistic approximation of worst-case execution time. Moreover, definition of ERWF tasks involves low-level features like global variables for data sharing and wait/notify primitives for synchronisation, which makes programming the system really hard and error-prone. The low-level workflows of ERWF are not comparable with those of P$\acute{\varepsilon}\alpha$.

Besides the somewhat general tools, which require solid programming knowledge to harness them, there are domain-specific ones as well. Those tools are easy to use for domain-experts, but the area where they can be applied is very limited.

Regiment [15] is a DSL to program sensor networks on the global level rather than individual sensor nodes. The approach in which the global network program is automatically translated into node-local programs is called macroprogramming. Regiment is a good tool for macroprogramming sensor networks, however it is not flexible enough for solving general CPS problems. Dynamic behaviour of applications, dynamically changing network configuration and rapid queries, which are present in P$\acute{\varepsilon}\alpha$ as the propagation of unstable values, cannot be implemented with Regiment.

There is a tool described in [13] which supports designing energy efficient buildings in a model-driven way. An UML-like modelling DSL is provided to define all the facilities that affect the energy consumption of a building. The tool can be used to make the building more energy efficient through the model-driven design process. Having the construction done, the sensors deployed throughout the premises is to be controlled by a distributed application generated by the tool and collecting data about the energy consumption. The data collected is then used to create different kinds of reports. The application has nothing to do with controlling the building, yet it could be a good basis for such a reactive system.

Not only buildings, but every mission-critical systems have to be monitored to check their behaviour. Copilot [16,17] is a real-time monitoring tool with the ability to oversee temporal properties of running systems in a non-intrusive way. The recent version of Copilot is able to trigger callbacks in the system in certain situations as well. That kind of monitoring tool comes in handy for auditing legacy systems. However, monitor and control functionality is better included into new applications by design, which can be implemented simply in P$\acute{\varepsilon}\alpha$.

7 Conclusion

A workflow system, P$\acute{\varepsilon}\alpha$, specifically designed for programming Cyber-Physical Systems is presented in this paper. P$\acute{\varepsilon}\alpha$ is based on the principles of task-oriented programming, nevertheless it restricts some of its features while extending its capabilities with new ones to suit the system to the needs of CPS programming. A working P$\acute{\varepsilon}\alpha$ framework is implemented in an extended version of the Erlang distributed functional programming language.

The features of the current implementation are revealed through a small scale illustrative example, the implementation of which poses all the challenges with which developers of large scale CPS applications have to cope.

The first version of our workflow system, as it is published in this paper, addresses the basic issues of CPS programming and provides a good basis for continuing research and adding more sophisticated features to the system in order to solve further open questions of the field of CPS programming.

A referee asked if there would be a second "incarnation" of P$\acute{\varepsilon}\alpha$. Having the first version of this paper submitted, we started to work on implementing P$\acute{\varepsilon}\alpha$ in Scala using the Akka library. Scala has a rich static type system, which can be leveraged to ease the development of P$\acute{\varepsilon}\alpha$ workflows by preventing many issues that occurs as runtime errors in the Erlang implementation, and raising static type errors instead. Moreover, Akka implements the actor model of Erlang, which makes it easy for us to port the Erlang implementation into Scala. Although the embedding into Scala is not completed at the time when this paper is finalised, we believe that the changes that are forced by the strong type system of Scala would result in a new incarnation of P$\acute{\varepsilon}\alpha$ that can be used in a more concise way than the original version presented in this paper.

Acknowledgement.

The authors would like to thank Dr Christian Rinderknecht for his comments and discussion. They are also grateful to the anonymous referees for their suggestions.

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. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (2007)
2. Bainomugisha, E., Carreton, A.L., Cutsem, T.v., Mostinckx, S., Meuter, W.d.: A survey on reactive programming. ACM Comput. Surv. 45(4), 52:1–52:34 (Aug 2013), http://doi.acm.org/10.1145/2501654.2501666
3. Blom, H., Lönn, H., Hagl, F., Papadopoulos, Y., Reiser, M.O., Sjöstedt, C.J., Chen, D.J., Kolagari, R.T.: EAST-ADL - An Architecture Description Language for Automotive Software-Intensive Systems. Tech. rep., The EAST-ADL 2 Consortium (2012)
4. Bozó, I., Horpácsi, D., Horváth, Z., Kitlei, R., Kőszegi, J., M., T., Tóth, M.: Refactorerl - source code analysis and refactoring in erlang. In: Proceedings of the 12th Symposium on Programming Languages and Software Tools, ISBN 978-9949-23-178-2. pp. 138–148. Tallin, Estonia (October 2011)

5. Broman, D., Lee, E.A., Tripakis, S., Törngren, M.: Viewpoints, formalisms, languages, and tools for cyber-physical systems. In: Proceedings of the 6th International Workshop on Multi-Paradigm Modeling. pp. 49–54. MPM '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2508443.2508452
6. Cesarini, F., Thompson, S.: ERLANG Programming, chap. Introduction, pp. 1–3. O'Reilly Media, Inc., 1st edn. (2009)
7. Chen, W.C., Shih, C.S.: Erwf: Embedded real-time workflow engine for user-centric cyber-physical systems. Parallel and Distributed Systems, International Conference on 0, 713–720 (2011)
8. Horpácsi, D.: Extending erlang by utilising refactorerl. In: Proceedings of the Twelfth ACM SIGPLAN Workshop on Erlang. pp. 63–72. Erlang '13, ACM, New York, NY, USA (2013), http://doi.acm.org/10.1145/2505305.2505314
9. Kats, L.C.L., Visser, E.: The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In: Rinard, M. (ed.) Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno, NV, USA. pp. 444–463 (2010)
10. Kitchin, D., Quark, A., Cook, W.R., Misra, J.: The Orc programming language. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) Proceedings of FMOODS/FORTE 2009. Lecture Notes in Computer Science, vol. 5522, pp. 1–25. Springer (2009)
11. Ko, R.K., Lee, S.S., Lee, E.W.: Business process management (bpm) standards: A survey. Business Process Management journal 15(5) (2009)
12. Kozsik, T., Lőrincz, A., Juhász, D., Domoszlai, L., Horpácsi, D., Tóth, M., Horváth, Z.: Workflow description in cyber-physical systems. STUD UNIV BABES-BOLYAI SER INFO LVIII(2), 20–30 (2013), http://www.cs.ubbcluj.ro/~studia-i/2013-2/052-Horvath.pdf
13. Kurpick, T., Pinkernell, C., Look, M., Rumpe, B.: Modeling cyber-physical systems: Model-driven specification of energy efficient buildings. In: Proceedings of the Modelling of the Physical World Workshop. pp. 2:1–2:6. MOTPW '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2491617.2491619
14. Lee, E.A.: Cyber physical systems: Design challenges. Tech. Rep. UCB/EECS-2008-8, EECS Department, University of California, Berkeley (Jan 2008), http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-8.html
15. Newton, R., Morrisett, G., Welsh, M.: The regiment macroprogramming system. In: Proceedings of the 6th International Conference on Information Processing in Sensor Networks. pp. 489–498. IPSN '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1236360.1236422
16. Pike, L., Goodloe, A., Morisset, R., Niller, S.: Copilot: A hard real-time runtime monitor. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G.J., Rosu, G., Sokolsky, O., Tillmann, N. (eds.) RV. Lecture Notes in Computer Science, vol. 6418, pp. 345–359. Springer (2010), http://dblp.uni-trier.de/db/conf/rv/rv2010.html#PikeGMN10
17. Pike, L., Wegmann, N., Niller, S., Goodloe, A.: Copilot: Monitoring embedded systems. Innov. Syst. Softw. Eng. 9(4), 235–255 (Dec 2013), http://dx.doi.org/10.1007/s11334-013-0223-x
18. Plasmeijer, R., Lijnse, B., Michels, S., Achten, P., Koopman, P.: Task-oriented programming in a pure functional language. In: Proceedings of the 14th Symposium on Principles and Practice of Declarative Programming. pp. 195–206. PPDP '12, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2370776.2370801

19. Srbljić, S., Škvorc, D., Popović, M.: Programming languages for end-user personalization of cyber-physical systems. AUTOMATIKA: časopis za automatiku, mjerenje, elektroniku, računarstvo i komunikacije 53(3), 294 – 310 (2012)
20. Sztipanovits, J.: Composition of cyber-physical systems. In: ECBS. pp. 3–6. IEEE Computer Society (2007), http://dblp.uni-trier.de/db/conf/ecbs/ecbs2007.html#Sztipanovits07
21. Voget, S.: Autosar and the automotive tool chain. In: Proceedings of the Conference on Design, Automation and Test in Europe. pp. 259–262. DATE '10, European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2010), http://dl.acm.org/citation.cfm?id=1870926.1870988