P*a*CE: A Data-Flow Coordination Language for Asynchronous Network-Based Applications

Mauro Caporuscio, Marco Funaro, and Carlo Ghezzi

Politecnico di Milano, Deep-SE Group - Dipartimento di Elettronica e Informazione
Piazza L. da Vinci, 32 - 20133 Milano, Italy
{caporuscio,funaro,ghezzi}@elet.polimi.it

Abstract. Network-based applications usually rely on the explicit distribution of components, which interact by means of message passing. Assembling components into a workflow is challenging due to the asynchronism inherent to the underlying message-passing communication model. This paper presents the P*a*CE language, which aims at coordinating asynchronous network-based components by exploiting the data-flow execution model. Specifically, P*a*CE has been designed for dealing with components compliant with the P-REST architectural style for pervasive adaptive systems. Moreover P*a*CE provides reflective features enabling run-time adaptation and evolution of workflows.

1 Introduction

The advent of new resource-constrained mobile computing devices (e.g., smartphones, and tablets) equipped with wireless networking technologies (e.g., WiFi, Bluetooth and 3G), together with the exploitation of new computing paradigms (e.g., Service Oriented Computing, Cloud Computing, and Pervasive Computing), is boosting a fast move from developing applications as standalone systems, to developing applications as network-based systems. Specifically, *network-based systems* rely on the explicit distribution of components, which interact by means of (asynchronous) message passing. Indeed, *network-based systems* differ from *distributed systems* in the fact that the involved networked components are independent and autonomous, rather than considered as integral part of a conceptually monolithic system [28].

In this settings, network-based applications can be easily modeled and developed as a set of interacting actors [4]. An actor is a computational unit that reacts to external stimuli (e.g., messages) by executing one or more of the following actions when stimulated: (*i*) sending messages to other actors, (*ii*) creating new actors, and (*iii*) designating the behavior for the next stimulus. Since there is no causal sequentiality between these actions, they can be carried on in parallel. Indeed, the Actor model is characterized by inherent concurrency among actors, dynamic creation of actors, and interaction through explicit asynchronous message passing (with no restriction on message arrival order).

T. Gschwind et al. (Eds.): SC 2012, LNCS 7306, pp. 51–67, 2012.

Although Actors are proper abstractions for modeling single reactive components that simply react to external stimuli, asynchronism makes them difficult to deal with when modeling *component compositions*. A composition is an "active" actor, also referred to as *orchestrator*, which orchestrates a process by accessing other components and consuming their artifacts. Indeed, the orchestrator queries actors, and aggregates their responses, to achieve its goal. However, since interactions are asynchronous, the orchestrator does not block its execution while waiting for responses. Rather, the orchestrator continues executing, and responses are processed asynchronously, with no causal order. To cope with such issues, we raised the level of abstraction, and devised a new coordination language satisfying the following requirements:

1. Using a RPC-like syntax
2. Retaining the inherent asynchronism of the pervasive environment
3. Making distribution and code parallelization as seamless as possible
4. Integrating local functions to carry out operations which are not coordination-related (i.e., manipulating the local state)

This paper presents P*a*CE (Prime Coordination languagE), a data-flow language for coordinating asynchronous network-based components. Data-flow languages [21][16] structure applications as a directed graph of autonomous software components that exchange data by asynchronous message passing. In the data-flow paradigm the components do not "call" each other, rather they are activated by the run time system, and react according to the provided input (received message). Once the output is available, the run time system is in charge of moving data towards the proper destination. Data-flow applications are inherently parallel. Exploiting the data-flow paradigm introduces a set of advantages: 1) concurrency and parallelism are natural and components can be easily distributed across the network, 2) asynchronous message passing is natural for coordinating independent and autonomous components, and 3) applications are flexible and extensible since components can be hierarchically composed to create more complex functionalities.

Specifically, the P*a*CE language has been designed and developed for composing and coordinating components built according to the P-REST architectural style [9], where components are called resources (we will use this term from now on) and are first-class abstraction acting as "prosumer" [24] – i.e., fulfilling both roles of producer (reactive actor) and consumer (active actor). To support the P-REST style we implemented the PRIME (P-rest Run-tIME) middleware [10]. Since PRIME has been specifically designed to deal with pervasive environments, where applications must support adaptive and evolutionary situation-aware behaviors, achieving *Adaptation* and *Evolution* is primary requirement for the middleware. *Adaptation* refers to the ability to self-react to environmental changes to keep satisfying the requirements, whereas *evolution* refers to the ability of satisfying new or different requirements. PRIME satisfies such goals by providing support for: (*i*) *flexibility*, the middleware is able to deal with the run-time growth of the application in terms of involved resources, (*ii*) *genericity*, the

middleware accommodate heterogeneous and unforeseen functionalities into the running application, and (*iii*) *dynamism*, the middleware is able to discover new functionality at run time and rearrange the application accordingly.

Therefore, P*a*CE allows developers to specify the active behavior (composition logic) of a *composite resource* in terms of the set of operations defined by the PRIME programming model. Moreover, P*a*CE exploits PRIME features to achieve both adaptation and evolution of compositions in terms of *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring* [23].

The paper is organized as follows: Section 2 discusses related work, Section 3 overviews the PRIME middleware, and its programming model. Section 4.1, Section 4.2, and Section 5 present the P*a*CE syntax, semantics, and interpreter, respectively. Section 5.1 discusses dynamic adaptation features provided by P*a*CE, and Section 6 assesses the work done by presenting a case study. Finally, Section 7 concludes the paper and sketches our perspectives for future work.

2 Related Work

The growth in complexity and heterogeneity of software systems imposes the need of raising the level of abstraction to make the software development process as rigorous as possible. Gelernter and Carriero [13] advocated for the sharp separation between computation (i.e., the tasks that must be executed to achieve the final goal), and coordination (i.e., how the tasks must be arranged to achieve the final goal) in large systems. P*a*CE is a data-flow coordination language.

Data-flow languages emphasize *data*, and the transformations applied to it to produce desired outputs. The introduction of this perspective is mainly motivated by the inherent unsuitability of the Von Neumann's architecture to the massive parallelism due to its global program counter and its shared memory that rapidly become bottlenecks in parallel programs [6].

In the data-flow computational model, a program is represented by a directed graph built at compile time, where nodes represent instructions, and arcs represent the data dependencies between instructions. When all the arcs entering a node (the *firing set*) have data on them, the node becomes *fireable*. During the execution, the instruction (represented by the fireable node) is executed and its result is placed on (at least) one of the outgoing arcs. Then, the node suspends executing as long as it is again non-fireable. Figure 1 shows the translation of a simple program (on the left-hand side) into the equivalent data-flow graph (on the right-hand side).

Liskov and Shrira in [19] introduced a similar solution for asynchronous and type-safe RPC in the Argus programming language. The basic idea is similar (i.e., continue executing as long as it is possible), but, with respect to data-flow languages, the degree of parallelism attained is lower because of the benefits granted by functional features.

Therefore, exploiting a data-flow approach to compose and coordinate software components is very appealing and the benefit is twofold: (*i*) focusing on

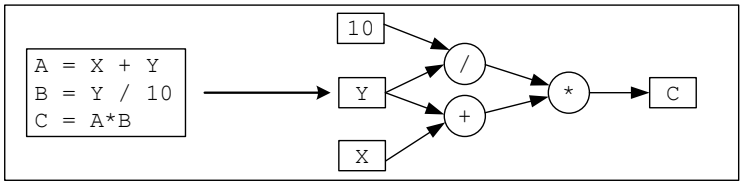


Fig. 1. A simple program and its translation into the equivalent data-flow graph

data allows for a more natural composition modeling approach since, if aided with a visual support, it can also be used and understood by non technical people; (*ii*) developers do not explicitly care about tasks concurrency. Rather, the execution model allows for automatic parallelization.

Pautasso and Alonso exploited the former benefit by proposing the JOpera visual composition language [26], and run-time support [25]. The JOpera language models both data-flow and control-flow dependencies among the tasks in the composition and the development environment is in charge of keeping the two perspectives consistent. The approach does not exploit the data-flow model to implicitly achieve parallelization, rather, the latter is achieved through imperative constructs inserted in the control-flow perspective. In [26] Pautasso and Alonso point out that the data-flow perspective is not enough to model every process because it ignores indirect dependencies (e.g., tasks communicating through databases or configuration files) or because there are dependencies that are not data-related like a compensation handler. P*a*CE addresses the first issue by adopting a purely functional approach, where no side-effects are allowed, and thus no indirect dependencies can be introduced. On the other hand, the compensation handler issue is out of our research scope.

Regarding the latter benefit – i.e., implicit parallelization – a complete general-purpose coordination language has not been proposed. Rather, researchers focused on raising the level of abstraction by proposing languages where nodes in the data-flow graph are functions written in different languages. For example Bernini and Mosconi [8] proposed a visual data-flow language called VIPERS where the node in the data-flow graph are Tcl fragments. Also textual approaches exist, like GLU [14] that embeds C fragments in the LUCID [29] data-flow language. These solutions exploit the implicit parallelism and delegate to other languages the whole computation. As a final remark, they are mostly focused on exploiting parallel computers for scientific computations and are not designed to fit distributed environments.

Another research area slightly related to this work, comprises agent-based workflow modeling and enactment. Indeed, an agent is an autonomous and intelligent program that make decisions on next actions to perform based on its current state. Hence, in agent-based workflow management, agents are provided with goals extracted from the overall workflow schema, thus each of them develops and its own work plan to achieve those goals. Agents-based workflow management systems coordinate agents by exploiting different approaches [15]:

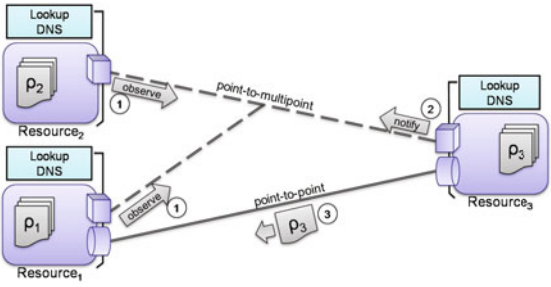


Fig. 2. P-REST architectural style

(*i*) *role-based*, where different agents fulfill different roles and perform a workflow autonomously, (*ii*) *activity-based*, where agents coordinate the execution of activities as defined within the workflow schema without the need for a central workflow enactment service, and (*iii*) *mobility-based*, where the workflow instance is migrated to different locations to perform specific tasks.

3 The P-REST Approach at a Glance

In this section we briefly introduce the Pervasive-REST (P-REST) [9] architectural style, and the PRIME middleware, which provides the run-time support needed for implementing P-RESTful applications – i.e., applications built following the P-REST style.

The P-REST architectural style (depicted in Figure 2) is defined as a refinement of the well known REST architectural style [12], to specifically deal with pervasive environments. P-REST promotes the use of *Resource* as first-class object that plays the role of "prosumer" [24], i.e., an entity that fulfills both roles of provider and consumer. To support coordination among resources, P-REST extends the traditional request/response mechanism through new primitives: (*i*) a *Lookup* service that enables the discovery of new resources at run time, (*ii*) a distributed *Domain Name System* (DNS) [22] service that maintains the mappings between resource URIs and their actual location in case of mobility, and (*iii*) a coordination model based on the Observer pattern [18] that allows a resource to express its interest in a given resource and to be notified whenever changes occur.

In P-REST, resources directly interact with each other by exchanging their representations. Referring to Figure 2, both Resource_1 and Resource_2 observe Resource_3 (messages ①). When a change occurs in Resource_3, it notifies (message ②) the observer resources. As the notification is received, Resource_1 issues a request for Resource_3 and obtains as a result the representation ρ_3 (message ③). Note that, while observe/notify interactions take place through the *point-to-multipoint* connector (represented as a cube), REST operations exploit *point-to-point* connector (represented as a cylinder). All the resources exploit both

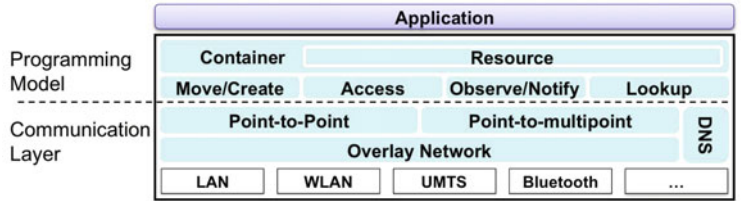


Fig. 3. PRIME layers

the `Lookup` operation to discover the needed resources, and the DNS service to translate URIs into physical addresses.

According to the *uniformity* principle [12] P-REST describes every software artifact as a Resource implementing a set of well-defined operations, namely `PUT`, `DELETE`, `POST`, `GET`, and `INSPECT`. Moreover, P-REST adopts semantic resource's descriptions to specify both functional and non-functional properties of a resource. Indeed, descriptions support the implementation of the lookup service by enabling run-time semantic-aware resource discovery.

P-REST enhances the REST *addressability* principle – i.e., a resources is identified by means of an URI – by distinguishing between *Concrete URI* (cURI) and *Abstract URI* (aURI). cURI identify concrete resource instances, whereas aURI identify groups of resources. Such groups are formed by imposing constraints on resource descriptions (e.g., all the resources implementing the same functionality). Therefore, cURI achieves point-to-point communication, and aURI achieves point-to-multipoint communication. Resources can be used as building-blocks for composing complex functionalities. A *Composition* is still a resource that can, in turn, be used as a building-block by another compositions. Resources involved in a composition are handled by means of a *Composition Logic*.

The PRIME(P-rest Run-tIME) [10] middleware provides the run-time support for the development of P-RESTful applications[1]. Referring to Figure 3, the PRIME architecture exploits a two-layer design where each layer deals with a specific issue:

Communication layer – To deal with the inherent instability of pervasive environments, PRIME arranges devices in an overlay network built on top of low-level wireless communication technologies (e.g., Bluetooth, Wi-Fi, Wi-Fi Direct, and UMTS). Such an overlay is then exploited to provide two basic communication facilities, namely *point-to-point* and *point-to-multipoint*. *Point-to-point* communication grants a given node direct access to a remote node, whereas *point-to-multipoint* communication allows a given node to interact with many different nodes at the same time. Furthermore, the PRIME communication layer provides facilities for managing both physical and logical mobility [27].

[1] PRIME is available at `http://code.google.com/p/prime-middleware/`, under the GNU/GPLv3 license.

Programming model – PRIME provides the programming abstractions to implement P-RESTful applications by leveraging the functional programming features of the Scala language [1] and the Actor Model [4]. In particular, PRIME defines two main abstractions and a set of operations to be performed on them. *Resource* represents the computation unit, whereas *Container* handles both the life-cycle and the provision of resources. The set of operations allowed on resources defines the message-based PRIME *interaction protocol* and includes: (*i*) *Access*, which gathers the set of messages to access and manipulate resources, (*ii*) *Observe/Notify*, which allows resources to declare interest in a given resource and to be notified whenever changes occur, (*iii*) *Create*, which provides the mechanism for creating a new resource at a given location, and *Move*, which provides the mechanism to relocate an existing resource to a new location, and (*iv*) *Lookup*, which allows for discovering new resources on the basis of a given semantics-aware description.

It is worth noticing that the PRIME programming model exploits the Actor Model, which in turn relies on the PRIME communication layer to provide message-passing interaction among actors.

4 P*a*CE – Prime Coordination languagE

This section presents both the syntax (§4.1) and semantics (§4.2) of P*a*CE, which have been specifically designed to offer the set of core features characterizing data-flow languages [16]: (1) single assignment of variables, (2) freedom from side-effects, (3) data dependencies equivalent to scheduling (statements are not executed in the order they are written, but as their input data become available), (4) an unusual notation for iterations (due to features 1 and 2), and (5) lack of history sensitivity in procedures (in a language without a deterministic control flow, histories cannot be univocally built by a developer).

Realizing such features strongly impacts on both syntax and semantics. In fact, (1) and (2) ask for a functional programming style, where multiple variable assignment is avoided, and functions are side-effects free (i.e., do not affect the environment, and their results depend only on input values). Moreover, features (1) and (2) are fundamental to induce scheduling from data dependencies (3). In fact, since scheduling is determined from data dependencies, it is important to guarantee that variables do not change between their definition and their use. Whenever variables are modified at run time, the data-flow graph (see Figure 1) would be invalidated. On the other hand, due to single-assignment, the order of statements is in general not relevant. However, single assignment conflicts with the imperative style in loops (4) because it forbids the increment of loop variables, thus loops are implemented through special constructs. Finally, data-flow languages inherit from the functional languages the lack of history sensitivity for procedures (5). In a language without a deterministic order of execution, histories cannot be univocally built by a programmer. Therefore, operations rely on the input parameters only and not on previous invocations. The functional operators along with the ordered queues are enough to guarantee a deterministic

```
ID          = [A-Za-z]([A-Za-z] | [0-9])*
INTEGER     = [1-9]([0-9])*
STRING      = "([A-Za-z] | [0-9])*"
URI         = ID | STRING
OP          = OPTWOPAR '(' URI ',' ID ')'
            | OPONENOPAR '(' URI ')'
OPTWOPAR    = 'post' | 'put'
OPONEPAR    = 'get' | 'delete' | 'inspect'
BLOCK       = '{' STATEMENT+ '}'
STATEMENT   = LOOP | ASSGNM | OBSERVE | CREATE | OUTPUT
            | INFLOOP | IF | WRITE | LOOKUP | SMFUN
OBSERVE     = 'observe' '(' URI ')' BLOCK
CREATE      = ID '=' 'create' '(' URI ',' ID ')'
            | ID '=' 'create' '(' URI ',' ID ',' URI ')'
LOOKUP      = ID '=' 'lookup(' ID ')'
ASSGNM      = ID '=' OP
            | ID '=' ID '(' (STRING)? (',' STRING)* ')'
            | ID '=' 'get(' 'stdin' ')'
SMFUN       ID '(' (STRING)? (',' STRING)* ')'
OUTPUT      = ID '(' (STRING)? (',' STRING)* ')'
            | 'put' '(' 'stdout' ',' ID ')'
            | 'put' '(' 'stdout' ',' STRING ')'
LOOP        = 'while' ID 'in' ID 'to' ID BLOCK
INFLOOP     = 'while' '(' 'true' ')' BLOCK
IF          = 'if' '(' BOOLEXP ')' BLOCK 'else' BLOCK
BOOLEXP     = BOOLEXP
            ( '&&' | '||' | '<' | '>'| '<=' | '>=' | '==' )
            BOOLEXP
            | ID | INTEGER | STRING |'!' BOOLEXP
            | '(' BOOLEXP ')'| 'true'| 'false'
```

Fig. 4. EBNF for P*a*CE

behavior for the model, that is, for a given set of inputs, a program always produces the same set of outputs [5][11][17].

4.1 Syntax

According to the above guidelines, P*a*CE's syntax is mainly inspired by common functional languages but for control structures, which instead are close to the imperative style. Therefore, every instruction is an assignment, operations are side-effects free (i.e., do not affect the environment, and their results depend only on input values), and multiple variable assignment is avoided. Moreover, since P*a*CE is tailored to the P-REST style, it directly embeds P-REST operations, which in turn are straightforwardly mapped to the PRIME programming model.

Referring to the generative EBNF for P*a*CE (see Figure 4), PRIME's access operations are derived from the `OP` non-terminal. `OPONEPAR` operations are invoked with one mandatory parameter – i.e., the list of `URI`s identifying the target

resources –, whereas `OPTWOPAR` operations get the `URI`s list, and an additional parameter containing a representation. Return values are lists of representations, and depend on the specific operation used: `GET` returns the representation of the target resource; `PUT`, `DELETE` and `POST` return a representation of the *status code* (e.g, "ERROR", "OK", and "NORESPONSE"), and `INSPECT` returns a representation of the resource description. Note that all PaCE operations are designed to work on lists. It is also possible to issue `GET` and `PUT` on two special `URI`s: `stdin` and `stdout`, respectively. The operation `a = GET(stdin)` reads a from the standard input, and assigns a value to the variable `a`. Conversely, the instruction `PUT(stdout, ID)` writes the value of `ID` on the `stdout`.

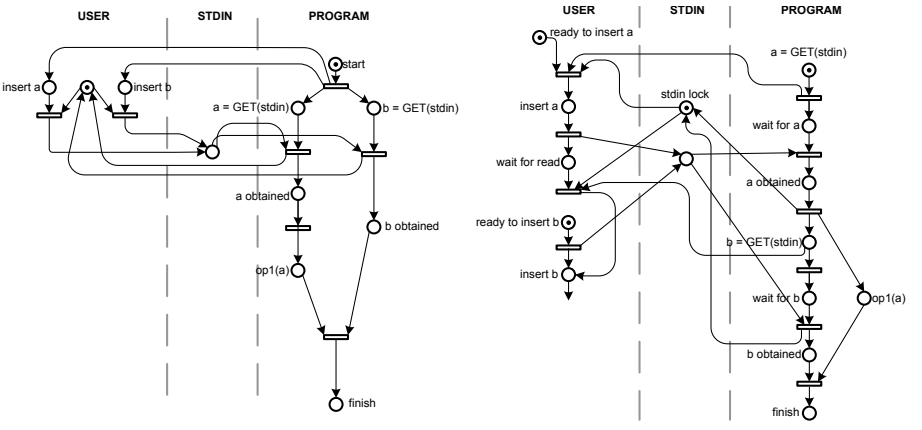
The EBNF can also generate invocation to two categories of external functions. The first one gathers the functions without a return type. These functions, thus, cannot directly affect the PaCE script but can only update the state of the composite resource the script is attached to. For this reason they are called *State-Manipulating Functions* (SMFUN from now on). The second one comprises the functions with a return value that are generated by the `ASSGNM` nonterminal. Notice that also these functions might have side effects on the composite resource state.

The `LOOKUP` operation is invoked with one parameter, which represents the identifier of the external function used to filter out the resources, and returns a list of URIs. The `CREATE` operation gets as input the container `URI` where the resource has to be created, and the representation used to initialize the new resource. A third parameter can be provided to impose a specific `URI` for the new resource. The return value is the `URI` of the new resource. The `OBSERVE` operation, which exploits the event-driven communication model provided by PRIME, is defined as control structure. `OBSERVE` defines a block of statements that is executed whenever an event, generated by observed resources (specified by the `URI` parameter), is received.

PaCE provides a set of simple control structures. The infinite loop (i.e., **while** `(true) {...}`), and conditional structure (i.e., **if** `(cond) {...}` **else** `{...}`) have the usual syntax. Whereas the `LOOP` control structure requires for special attention. As already mentioned, the *single assignment* feature prevents the implementation of standard loops where the loop variable is explicitly incremented at every iteration. To overcome such an issue, PaCE defines a control structure of the form: **while** `var` **in** $\texttt{b}_1$ **to** $\texttt{b}_2$ `{...}`, where `var` ranges from $\texttt{b}_1$ to $\texttt{b}_2$ by preventing its explicit assignment within the loop.

4.2 Semantics

Since PaCE is inspired by data-flow languages, it adheres to the data-flow execution model. However, while data-flow languages build explicitly the data-flow graph to drive the execution, in PaCE such graph is built implicitly. Indeed, PaCE does not completely depart from the sequential execution of instructions, but makes it asynchronous: statements are evaluated sequentially, but their execution is non-blocking. That is, given a two-statements sequence $< S_1, S_2 >$, S_2 can be executed independently of the S_1 termination, as long as the



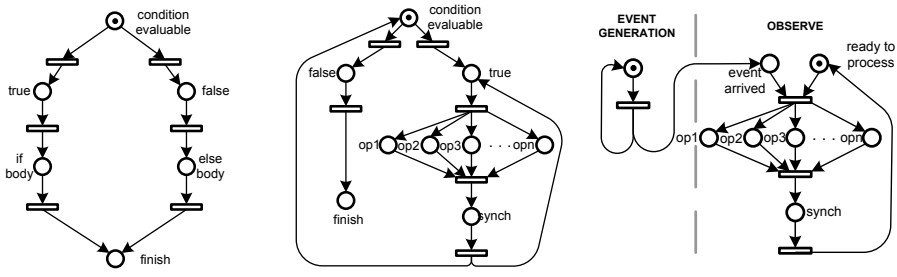
(a) Pure data-flow execution model (b) P*a*CE execution model

Fig. 5. Execution model

execution of S_2 does not need data produced by S_1. When a statement is invoked, it returns immediately by yielding a *future* variable – i.e., a special variable that will eventually contain the result of an asynchronous computation. Whenever the variable is accessed, execution is suspended if the value is not available, yet.

P*a*CE scripts are compliant with the data-flow execution model. However, their execution is not purely concurrent since parallel operations are executed as soon as the interpreter evaluates them, and needed data is available. Having non-pure parallelism, allows the P*a*CE interpreter to retain at run time the information about the instruction order. This is particularly important when dealing with I/O and control structures. In the data-flow execution model, sequentiality of instructions is lost when the data-flow graph is built, since only data dependencies are considered. For instance, let S_1 =<`a = get(stdin), c = op1(a) b = get(stdin)`> be a sequence with two independent `get`s. According to the data-flow execution model, they can be executed in any order. As a consequence, it is impossible to univocally determine which result, coming from a `get(stdin)`, must be stored in `a`, and which in `b` (see Figure 5a). To this end, P*a*CE semantics imposes the mutually-exclusive access to the `stdin` resource by exploiting the information about the instruction order. When the first `get(stdin)` is executed, it locks the standard input; then, when the second `get` tries to access `stdin`, it is suspended as long as the first `get` completes. In this case the two `get`s are ordered, and `stdin lock` ensures the mutual-exclusion policy for accessing to the standard input. It is worth noticing that such an issues does not concern the `put(stdout, ID)`, which can be executed whenever the `ID` variable is available.

As introduced in Section 4.1, P*a*CE scripts can also contain control structures: i.e., conditional (`if-else`), loops (`while`), and `observe` structures. Notice that for the sake of clarity, the Petri nets in Figure 6 do not take into account possible dependencies between instructions in the control structures's bodies and



(a) `if` (b) `while` (c) `observe`

Fig. 6. Semantics of the P*a*CE control structures

instructions outside the bodies. However, whenever any dependency holds, it is treated according to the data-flow execution model.

Conditional Structure – According to the execution model discussed above, the conditional structure is evaluated as soon as the *conditional expression* becomes evaluable. Hence, the proper branch is executed. Figure 6a shows the Petri net specifying the `if-else` semantics.

Loop Structure – As introduced in Section 4.1, the concept of loop does not fit the data-flow paradigm. However, having loops instead of the equivalent tail recursion, is fundamental for the adoption of the paradigm [3]. Therefore, the rationale of the loop syntax (see Figure 4) relies in the fact that P*a*CE is conceived as a coordination language. Indeed, any computation should be accomplished by either remote resources or external functions. According to the syntax given above, every loop iteration is forced to happen in isolation. That is, all the instructions in the loop body must complete before the next iteration can be executed. Moreover, all the variables, allocated within an iteration, are deallocated at the end the iteration, to guarantee the single-assignment property. Data dependences holding between instructions in the loop body and external instructions must be satisfied before the execution of the first iteration. Figure 6b shows the Petri net defining the `while` semantics. Note that, the isolation is guaranteed by `synch`, which forces the Petri net to wait until all the instructions in the body complete.

Observe Structure – The `observe` operation introduces the event-based programming paradigm: the `observe` is an infinite loop whose body is executed every time an event is produced in one of the observed resources. To avoid locking the execution in such infinite loops, every `observe` is executed separately to allow the interpretation of a script to continue beyond any `observe`. According to P-REST (see Section 3), the `observe` operation accepts as input the list of resources to be observed (specified by means of their `URI`s). Whenever an event is received from an observed resource, the body is executed according to the

data-flow execution model. Notice that also in this case the body is executed in isolation and events are queued and consumed sequentially. Figure 6c shows the Petri net specifying the `observe` semantics: `event generator` produces tokens (events), which `observe` consumes; `event arrived` models the incoming queue.

External Functions – P*a*CE allows for the definition of two types of external functions: *side-effect-free* and *state-manipulation* functions. *Side-effect-free* functions are used to manipulate P*a*CE variables (e.g., to translate the data from one encoding to another). *State-manipulation* (SMFUN) functions are used to manipulate the internal state of the composite resource. Due to the shared-nothing paradigm exploited by P*a*CE, SMFUN functions are critical, and two main problems arise. First, SMFUNs can conflict with the `POST` operation since it can modify the internal state. Second, concurrent SMFUNs can potentially modify the same data. To this extent, on one hand, P*a*CE provides mutual exclusion mechanisms to avoid the simultaneous access to the state of the composite resource. On the other hand, `POST` and SMFUNs should not be used at the same time to avoid unforeseen (and unpredictable) behaviors of the P*a*CE scripts. A POST changes the internal state of a resource, and SMFUNs can be used to read such internal state. Thus, the script behavior could be implicitly modified, and this fact would break the functional assumption of data-flow languages. Such interactions could lead to unforeseen (and unpredictable) behaviors of the P*a*CE scripts.

5 P*a*CE: Interpreter

This section presents the P*a*CE interpreter, and details how P*a*CE scripts are mapped to the underlying PRIME middleware.

The P*a*CE interpreter is developed in Scala [1], and exploits the Scala parser combinator library [20]. However, since in P*a*CE all the variables are stored as *future variables*, the interpretation algorithm makes use of an auxiliary symbol table containing all the bindings between variable names and future values. According to P*a*CE semantics (§4.2), every operation immediately returns a future variable that will be eventually filled with the result. Whenever the operation to be executed requires input parameters, whose values are not yet available, the interpreter suspends the execution of the analyzed instruction that will be resumed whenever the missing values will become available. Clearly, suspending the execution of an instruction does not suspend the execution of the whole script. Rather, the execution flow proceeds according to the data-flow paradigm, and allows for the implicit construction of the data-flow graph.

The interpreter also implement a basic error-handling mechanism. Requests are automatically reissued if a configurable timeout expires up to three times before terminating the script. This simple mechanism accounts for network problems only. Conversely, errors at the application level (i.e., errors generated by the queried resources) must be taken care of explicitly in the script because they are directly notified in the response.

PaCE implements P-REST operations by exploiting the PRIME middleware, which is in charge of dispatching requests and responses (see Section 3). Since PRIME operations are completely asynchronous, the PaCE interpreter implements an *indexing system* that (*i*) binds received responses to issued requests and, (*ii*) assigns the values embedded in the responses to the proper future variable in the symbol table.

The PaCE interpreter also offers an abstraction for developing and using external functions. In order to be used, external functions must be made available to the interpreter at run time. External functions are defined by extending the `ExternalFunctions` abstract class, which implements all the mechanisms needed to parse, validate and publish functions. `ExternalFunctions` exploits the Java reflection mechanism, and exposes a method `invoke`, which takes as input the name of the functions to be executed and a list (possibly empty) of parameters. The `invoke` method executes the function in a future block to guarantee asynchronism. In addition, `invoke` checks whether the function is a SMFUN or not. In fact, according to the PaCE semantics (§4.2), SMFUN functions are executed in mutual exclusion, whereas non-SMFUN functions can be executed concurrently.

5.1 Run-Time Adaptation

A primary requirement for P-RESTful applications is to support adaptive and evolutionary situation-aware behaviors [9]. To this extent, PaCE provides four operations on resources – i.e., *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring* – that allow PaCE's scripts to be reconfigured at run time.

According to the PaCE syntax and semantics discussed above, *resource addition/removal* simply refer to the ability of adding/removing a `URI` to/from a list, and *resource rewiring* means changing the value of a variable containing a `URI`. *resource substitution*, instead, consists of rewiring a resource binding, and moving the state of the old resource to the new one. It is worth to notice that, to avoid inconsistencies in the symbol table, the interpreter suspends the execution of scripts before performing any reconfiguration. Indeed, reconfigurations are performed asynchronously and in isolation – i.e., reconfigurations are queued and executed only when the interpreter is in a safe state. While *add*, *remove* and *rewire* operations are entirely implemented within the PaCE interpreter, *substitute* exploits the primitives provided by PRIME to retrieve the state of the old resource (`GET`), and to initialize the new one (`PUT`).

Special attention should be paid whenever the adaptation involves a variable containing observed `URI`s. Indeed, observed variables are referred once, at the beginning of the `observe` structure, when the variable is defined, and the corresponding subscription is generated. Hence, adaptation operations must be carefully examined before their application:

add: the intended semantics concerns the addition of a new resource to the pool of resources already observed. Whether an add is performed, a new subscription is issued to start observing the new resource.

remove: the intended semantics is the opposite of the add operation, i.e., the given resource `URI` should be no longer observed. Therefore, the old subscription is removed so that no further notifications will arrive from a specific URI.

rewire: the intended semantics concerns the substitution of the entire resource pool with a new one. This operation is implemented by removing every `URI` from the list, then adding new `URI` to the list.

substitute: this operation, if applied to the variable containing the observed resources, does not affect the behavior of the `observe` block.

6 Case Study

In this section we want to show how P*a*CE can be used to orchestrate PRIME resources. This section also covers the run-time adaptation facilities of the P*a*CE interpreter.

Let us introduce the following Pervasive Slide Show (PSS) scenario: *Carl, a university professor, is going to give a talk in a conference room, and carries his laptop storing both the slides and related handouts. The conference room provides speakers with a smart-screen available on the local wireless network, whereas the audience is supposed to be equipped with devices (e.g., laptops, smartphones and tablets), which can be used for displaying either the currently-projected slide on the screen or the related handouts. The audience and the speaker always refer to the same slide, and to the same page of the handouts. Every device is required to have a running* PRIME *instance.*

The PSS implementation conforms to the P-REST conceptual model and specifies the following resources: `CurrentSlide` and `CurrentPage` represent the currently-projected slide and the corresponding handout page, respectively. `Presentation` is the composition that implements the interface Carl uses to browse the slide-show. It also encapsulates the slides and the handouts along with the pointers that keep track of the current slide and handout page. `Reader` visualizes the slide show or the handout on the audience's devices. `Projector` handles the smart-screen of the conference room. The `Projector` resource is deployed on the smart-screen PRIME container. The other resources are initially deployed on Carl's container, and made available to the devices in the audience which join the slide-show.

When a participant (say, Bob) enters the conference room, he uses the PRIME *resource finder* built-in tool, which lists all of the resources available within the overlay, to explore the environment and find the `Reader` resource. Hence, selecting `Reader` from the list, the PRIME node issues a `GET` operation to retrieve a representation of `Reader`, which, in turn, is used to create a local instance. Figure 7 shows the P*a*CE script for `Presentation` and `Reader`. The `Presentation` resource is a composition meant to aid Carl to project his slides. Thus, the associated P*a*CE script (see Figure 7a) searches the conference center for a projector by using the external function `projSearch`. The resulting URIs are stored in the `proj` variable to be used later. Then, the execution enters in an infinite loop

```
proj = lookup(projSearch)
while (true){
  cmd = GET(stdin)
  if (cmd == ''fwd''){
    rep = getNextSlide()
    PUT(currSlide, rep)
    PUT(proj, rep)
  }
  if (cmd == ''bwd''){
    rep = getPreviousSlide()
    PUT(currSlide, rep)
    PUT(proj, rep)
  }
}
```

(a) Script for `Presentation`

```
observe(slURI){
  slide = GET(obsURI)
  view(slide)
}

while(true){
  cmd = GET(stdin)

  if(cmd == ''ho'')
    rewire(slURI, hoURI)

  if(cmd == ''pres'')
    rewire(slURI, presURI)
}
```

(b) Script for `Reader`

Fig. 7. Scripts for PSS scenario

to serve Carl's commands. In case of a "fwd" ("bwd") command the script calls an SMFUN called `getNextSlide`. It returns the representation of the new slide and, as a side-effect, updates the pointers to the current slide and to the current handout page in the `Presentation` resource. The new representation is stored in the `rep` variable and it is used as a parameter for the following two `PUT` operations: one, issued towards the `CurrSlide` resource, the other, towards the `proj` variable, that is, towards all the the smart-screens found in the conference room. All the `Projector` resources will react by projecting the new slide. As for the `CurrentSlide` resource, it is involved in a more complex interaction. Indeed, looking at Figure 7b, the `Reader` script observes the state of `CurrentSlide` and, whenever it changes, the `Reader` issues a `GET` towards it. When the new slide is retrieved, it is visualized through the `view` external function. Apart from the `observe` body, the script for `Reader` also features an infinite cycle to allow Bob to toggle between the presentation and the handout. This second part makes use of the adaptation primitives described in 5.1. They are used in a reflective way, that is, the script for the `Reader` reconfigures itself to serve Bob's commands. Specifically, when Bob issues the "ho" command, the script invokes the `rewire` functions with `slURI` (the observed URI) as first parameter (the old binding) and the URI of the `CurrentPage` resource (`hoURI`) as second parameter. The `hoURI` variable is hard-wired in the script by the system developer. Being `slURI` an observed variable the special rules presented in 5.1 apply. Specifically, the symbol table is updated so that the `slURI` will denote the `CurrentPage` URI and not the `CurrentSlide` anymore. As a consequence, also the `obsURI` variable is updated accordingly. To make the update effective the `observe` body must be executed once before starting processing the new data stream. After the `rewire` has completed, Bob will be able to follow the handout on his device.

Later on, Bob will be able to switch back to the presentation by issuing the "pres" command.

7 Conclusion

In this paper we addressed the problem of coordinating resources adhering to the P-REST architectural style. Such resources are modeled by PRIME as actors, i.e., an autonomous and asynchronous computational resource reacting to external stimuli. Although Actors are proper abstractions for modeling P-REST resources, asynchronism makes them difficult to deal with when assembled into a workflow. Indeed, resource coordination is a time-consuming and error-prone process for a developer.

To address these problems, in this paper we propose P*a*CE, a data-flow language for composing and coordinating resources built on top of the PRIME middleware [9]. Moreover, P*a*CE exploits PRIME features to enable both run-time adaptation and evolution of compositions. We described P*a*CE's syntax and semantics, and discussed the advantages and issues induced by the adoption of the data-flow paradigm. We also presented the P*a*CE interpreter, which provides reflective capabilities to achieve reconfiguration operations, namely *resource addition*, *resource removal*, *resource substitution*, and *resource rewiring*.

As for the future research directions, we want to improve the language by adding a full support to the error-handling before removing the strict binding between P*a*CE and P-REST, to obtain a general purpose coordination language suitable for every inherently parallel and asynchronous environment. Furthermore, following well known approaches, e.g., Yahoo Pipes [2], Mashlight [7] and JOpera [26], we plan to develop a tool for visually specifying resource compositions. This would further ease the development process, by allowing developers to fully benefit from data-flow paradigm: focus on how things interact, rather than on how things happen.

Acknowledgements. This research has been funded by the European Commission, Programme IDEAS-ERC, Project 227077-SMScom (`http://www.erc-smscom.org`).

References

1. The Scala programming language, `http://www.scala-lang.org/`
2. Yahoo pipes, `http://pipes.yahoo.com/`
3. Ackerman, W.: Data flow languages. Computer 15(2), 15–25 (1982)
4. Agha, G.: Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge (1986)
5. Arvind, Culler, D.E.: Dataflow architectures. Annual Review of Computer Science 1(1), 225–253 (1986)
6. Backus, J.: Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. Commun. ACM 21, 613–641 (1978)
7. Baresi, L., Guinea, S.: Mashups with Mashlight. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 711–712. Springer, Heidelberg (2010)

8. Bernini, M., Mosconi, M.: Vipers: a data flow visual programming environment based on the tcl language. In: AVI: Proceedings of the Workshop on Advanced Visual Interfaces. Association for Computing Machinery, Inc., New York (1994)
9. Caporuscio, M., Funaro, M., Ghezzi, C.: Restful service architectures for pervasive networking environments. In: Wilde, E., Pautasso, C. (eds.) REST: From Research to Practice. Springer (2011)
10. Caporuscio, M., Funaro, M., Ghezzi, C.: Resource-oriented middleware abstractions for pervasive computing. In: IEEE International Conference on Software Science, Technology and Engineering, SWSTE (to appear, June 2012)
11. Davis, A., Keller, R.: Data flow program graphs. Computer 15(2), 26–41 (1982)
12. Fielding, R.T.: REST: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, Irvine (2000)
13. Gelernter, D., Carriero, N.: Coordination languages and their significance. Commun. ACM 35, 97–107 (1992)
14. Jagannathan, R.: Coarse-grain dataflow programming of conventional parallel computers. In: Advanced Topics in Dataflow Computing and Multithreading, pp. 113–129. IEEE Computer Society Press (1995)
15. Joeris, G.: Decentralized and flexible workflow enactment based on task coordination agents. In: 2nd International BiConference Workshop on Agent-Oriented Information Systems (AOIS 2000 - CAiSE 2000), pp. 41–62 (2000)
16. Johnston, W.M., Hanna, J.R.P., Millar, R.J.: Advances in dataflow programming languages. ACM Comput. Surv. 36, 1–34 (2004)
17. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Rosenfeld, J.L. (ed.) Information Processing 1974: Proceedings of the IFIP Congress, pp. 471–475. North-Holland, New York (1974)
18. Khare, R., Taylor, R.N.: Extending the representational state transfer (rest) architectural style for decentralized systems. In: ICSE 2004, pp. 428–437. IEEE Computer Society, Washington, DC (2004)
19. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation, PLDI 1988, pp. 260–267. ACM, New York (1988)
20. Moors, A., Piessens, F., Odersky, M.: Parser combinators in scala. CW Reports, vol. CW491. Department of Computer Science, KU Leuven (2008)
21. Morrison, J.P.: Flow-Based Programming: A New Approach to Application Development, 2nd edn. CreateSpace (2010)
22. Network Working Group. Role of the Domain Name System (DNS). RFC3467
23. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-based runtime software evolution. In: ICSE 1998 (1998)
24. Papadimitriou, D.: Future internet - the cross-etp vision document, version 1.0 (January 2009), `http://www.future-internet.eu`
25. Pautasso, C., Alonso, G.: Jopera: A toolkit for efficient visual composition of web services. Int. J. Electron. Commerce 9, 107–141 (2005)
26. Pautasso, C., Alonso, G.: The jopera visual composition language. Journal of Visual Languages & Computing 16(1-2), 119–152 (2005); 2003 IEEE Symposium on Human Centric Computing Languages and Environments
27. Roman, G.-C., Picco, G.P., Murphy, A.L.: Software engineering for mobility: a roadmap. In: FOSE 2000, pp. 241–258. ACM, New York (2000)
28. Tanenbaum, A.S., Van Renesse, R.: Distributed operating systems. ACM Comput. Surv. 17, 419–470 (1985)
29. Wadge, W.W., Ashcroft, E.A.: LUCID, the dataflow programming language. Academic Press Professional, Inc., San Diego (1985)