Reliable Execution of Business Processes on Dynamic Networks of Service-Enabled Devices

P. Spiess, S. Karnouskos, L.Souza, D. Savio, D. Guinard, V. Trifa, O. Baecker, M. Koehler
SAP Research (www.sap.com)
Email contact: patrik.spiess@sap.com

***Abstract*—It is expected that future shop-floors will be populated by thousands of networked embedded devices. Those will not only communicate using IP (as in TCP/IP), but also feature some autonomy, allowing them to collaborate among themselves and with enterprise systems. As they can offer both their mechatronic and higher-level functions as a service and support dynamic deployment of new code, they can execute business logic locally, allowing for new classes of business processes that are executed collaboratively by back-end and embedded systems. While some parts of a process will still be executed in the data centre, the rest will execute directly on embedded devices on the shop-floor. Business Process execution will therefore be more dynamic and context-based. We introduce an approach to manage efficient business process execution over such highly dynamic infrastructures.**

I. Motivation and Problem Statement

With technological advances, networks of embedded devices have become more powerful and capable of executing tasks in a peer to peer fashion. The infrastructure envisioned by the Internet of Things [4] is a heterogeneous one, where millions devices are interconnected, are ready to receive instructions and create event notifications, and where the most advanced ones can self-organize and collaborate. With this new paradigm, business logic can be pushed down and be distributed to several layers such as the network or even the device layer, thus reducing the information load that has to be processed by the enterprise system, increasing scalability and response time as the business logic is executed at the point of action [10].

Service Oriented Architecture (SOA) has emerged as the de-facto standard approach of handling business processes. Despite severe heterogeneity of devices and embedded software, SOA is starting to be applied also to networks of networked embedded systems [3]. Standards that define web service communication and management for embedded devices, such as Device Profile for Web Services (DPWS [2]) and OPC Unified Architecture (OPC-UA [12]) act as enablers for this ambitious step forward. They enable business process management and execution at the device level with interfaces that are independent of programming languages and operating.

Although SOA at the device level simplifies business process composition using services that execute on dynamic networks of devices, new challenges rise due to the characteristics of such networks. With networked embedded devices, context information is commonly used to decide if a given device should execute a service or not. For instance it can be desirable that only devices in a certain location or under a certain environment condition execute a specific task of a business process.

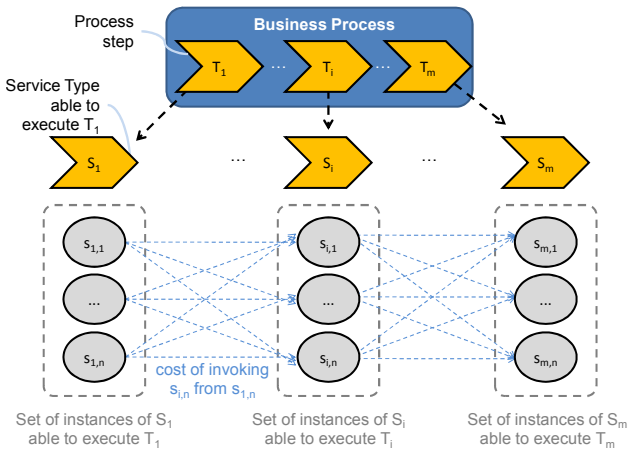

Fig. 1. Business Process Composition

Hence, to compose a business process that is partially executed in a network of embedded devices, it is necessary to adequately consider the dynamics of the system and verify changes in context information, in order to ensure reliable execution of the process. This paper outlines our approach that enables dynamic business process execution for networked embedded systems based on devices that offer their functionality as services.

II. Overview of our solution

The approach proposed abstracts the business process as a sequence of semantically annotated tasks $(T_1, T_2, ..., T_n)$ as depicted in Figure 1. We further differentiate between service types $(S_1, S_2, ..., S_n)$ and service instances $(s_1, s_2, ..., s_n)$. A service type is mainly determined by its interface while a service instance, i.e. an installed, running, and connected copy of the service, has a unique identity, a network address (e.g. a WS endpoint reference), and context information.

We only consider those tasks that involve an interaction with a service that is not part of the process itself, e.g. an invocation of a service (such as a web service, a REST-like service [13], a remote procedure call, etc.) or the receipt of an asynchronous message from such a service. In BPEL semantics, this corresponds to *invoke* and *receive* operations.

For the invocation tasks, control is handed over from the executing process instance to the service and is returned to the process instance after the external service replied. For the reception task, process execution is suspended until the expected message arrives. The services are provided by devices that are capable of executing them. The proposed approach has its greatest benefit if the devices are connected in an unreliable fashion and therefore provide their services in an unreliable way, which we assume to be typical characteristic of networked embedded devices.

We further assume that context information is available for each device (e.g. by monitoring or self-announcement of the devices and/or the services instances they host) and that the context information can be used to calculate technical cost of sending a one-way message from any service instance to any target service instance (cost of communication $cc_{i,j}$) and executing the corresponding action within the target (cost of execution $ce_{i,j}$). The context information is expected to vary over time, hence so is the calculated cost. For instance, the communication cost between two hosted service instances will be dependent on the location of the devices, network load, network media etc. In another example, executing an action on an overheated machine will be assigned a higher cost than executing it on an idle device.

Our approach tries to find and continuously adapt a service orchestration that executes a given process (a sequence k determining a sequence of instances $s_{1,k_1}, s_{2,k_2}, ..., s_{n,k_n}$) that is optimal w.r.t. the technical cost involved, i.e. where the total cost $c = \sum_{n=0}^{m}(cc_{m,k_m} + ce_{m,k_m})$ is minimal.

To handle the dynamic aspects of such systems, this solution proposes the use of several components that not only define and execute an optimized control flow between services implementing individual tasks, but also monitor the respective context information of the selected devices and hosted services in order to track changes and redefine the path of the control flow when necessary.

III. Architecture

The business process life-cycle is composed of five main phases: (1) Design, (2) Path Assignment, (3) Execution, (4) Path reassignment, and (5) Removal. A software architecture supporting all of these phases is presented in Figure 2.

During Design time, a Process Modelling Application (e.g. a BPMN, BPEL, or generic work flow modelling tool) will assist the business process designer to create the structure and connections of the tasks involved in the process. This is usually performed using an intuitive GUI that provides the required support for users. However, a mere text editor can also serve as a "'modelling tool"', where the user creates a textual representation of the process that follows a given standard (e.g. a given XML format as in the case of BPEL). The output of the modelling application is an abstract process that contains a description of the services type it is going to use, but contains no links to concrete instances of that service type.

Once the process has been designed, it is necessary to assign the subset of its tasks that are supposed to interact with

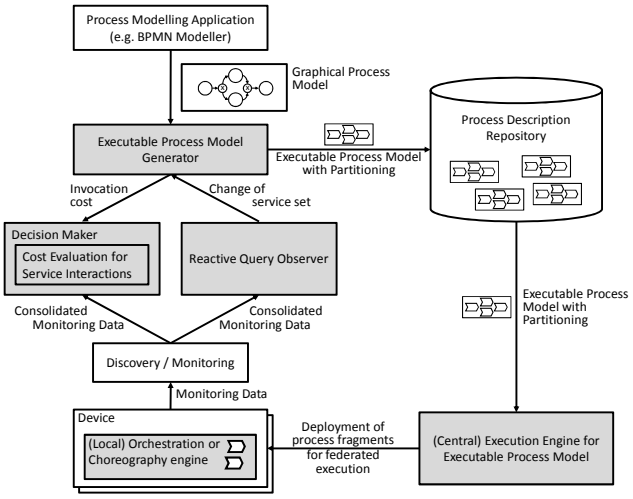

Fig. 2. Architecture for Business Process Partitioning

embedded services, to the service instances running on the smart devices. The Decision Maker component is responsible for this phase (2). It makes use of previously incoming, cached discovery/monitoring data to compute and report services instances that match the requirements of each defined task. Considering contextual information, a cost estimation of the execution of a task in a specific service instance $ce_{i,j}$is calculated. Additionally, the interaction costs between each task and the different possible subsequent services instances $cc_{i+1,0}, cc_{i+1,1}, ..., cc_{i+1,n}$ are also calculated. Finally, the Decision Maker component chooses the path with the least cost $(k_1, k_2, ..., k_n)$ and returns it to the Executable Process Model Generator (EPMG).

The EPMG generates an executable process model with partitioning instructions and stores it in the Process Description Repository. When a running version of the process is needed (phase (3)), the executable process description is inserted into a Central Execution Engine which will coordinate the execution of the process between the different service instances as described in section III-F.

The set of currently active process instances determines, which part of all data monitored about the devices and service instances is relevant. Hence, upon process activation, the Reactive Query Observer is armed to monitor the subset of monitoring data that is relevant to the active processes in the network. A (possibly complex) rule set is installed for each process. In case any of the rules is violated, the reactive query observer triggers the EPMG to re-calculate the optimal path for the respective process (phase(4)).

When a human administrator or an application decides to remove a process from the network (phase (5)), it is removed from the central orchestration engine and the local ones.

A. Executable Process Model Generator

The Process Modelling Application (e.g. BPMN Modeller) helps building an abstract model of the process. This model reflects e.g. the business aspects of the process; it defines the

actors, the actions to be performed, the sequence in which tasks are performed, which tasks can be executed in parallel, forks and joins from sequential to parallel execution and back to sequential as well as how the outputs of previous service interactions are used in subsequent service interactions.

An example model at that level would be an the work flow of an order through a company from coming in, processing by a particular chain of workers/machines, storage, to delivery. A process expert would model this process in a modelling tool. The Executable Process Model Generator takes that model and maps its tasks to concrete service calls. A number of languages can be used as that level such as XPDL or BPEL. But the Executable Process Model Generator (simply referred to as generator hereafter) goes beyond simple translation. To establish the execution model it works collaboratively with the Decision Maker to split the execution into smaller execution units. These units are constructed both based on partitioning instructions, and availability of devices able to execute the semantically described sub-tasks. The Executable Process Model Generator also asks the Decision Maker to find the optimal execution path of the process (for a detailed analysis see III-B). This information is also taken into account when establishing the partitioned executable process model. This model is then sent to the Process Description Repository.

Because the processes we are considering are partly executed on a network of embedded devices (i.e. the tasks of the process are interactions with services provided by these devices), the partitioned execution model can not be statically defined: the execution context and hence the cost of execution c is likely to change over time. In order to keep the process requirements continuously satisfied, the generator is informed of changes in the optimal execution path by the Reactive Query Observer by means of asynchronous messages. The continuous queries (realized by a rule set) are instantiated every time a process instance is activated. Whenever it identifies a change in the results of the stored queries, the Reactive Query Observer sends a message to the generator. Upon arrival of the message the generator will decide whether a re-evaluation of some parts of the execution path is required, e.g. because the technical cost of the current path now exceeds a given threshold or has increased by a given percentage compared to the original cost. If this is the case, the generator will again use the Decision Maker to find a better execution path.

1) Partitioning the Process: After the generator has obtained the information, which services will be used to realize each of the tasks in the process description, it evaluates if it is beneficial to partition the process. To do so, it must be aware of all process execution engines that are available in the system. We assume that there is a central execution engine available to execute the process as a whole. But to make process execution more local, more execution engines are available which are lightweight enough to run on gateways to devices or on the embedded devices themselves. In the most extreme case, each device not only hosts services, but also a lightweight execution engine ready to execute process fractions.

The reason for partitioning the process is that the interaction between the central execution engine and the service instances on the devices might involve higher technical cost than interactions between the services directly. If, for example, two services on the same network or on the same device are used in a sequence, it may be more beneficial to execute this interaction locally. Figure 3 shows an example, where process partitioning replaces 8 expensive operations by 2 expensive an 3 cheap ones (at the cost of message size, if tasks depend on outcome of previous tasks). Since the central execution engine might also be remote (e.g. because it is part of a hosted application), the communication from the central engine to the service leads to longer latency, added unreliability, and communication overhead.

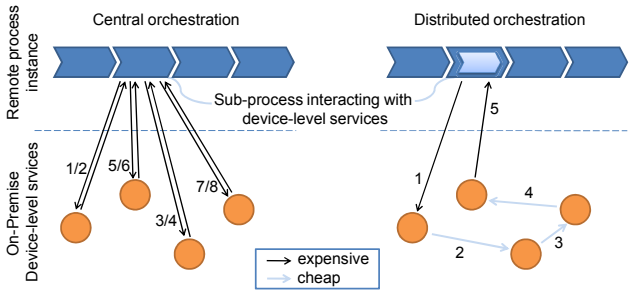

Fig. 3. Relocation of Process Parts

B. Decision Maker

The Executable Process Model Generator issues a request to the Decision Maker to find the optimal path (as depicted in Figure 4). This will trigger a cost evaluation of task execution on specific service instances. The evaluation is done based on key performance indicators (KPI) that are either selected on-the-fly or are defined in the system configuration. Such KPIs are realized by calculating technical cost (as described earlier). The costs look at the usage of different resources like communication cost, latency, communication density among tasks, computation cost, memory usage, energy consumption or even other related constraints like e.g. time, software version constraint cost etc. All of these are considered as criteria based on which decisions are made. According to the previous cost model, there is a c for all of these dimensions, so you would add another index to c and the $cc_{i,j}$ and cei, j and introduce a weight function, combining the costs along the different dimensions: $c_{total} = \sum_{k=1}^{n}(\alpha_k \cdot c_k)$. Once a final decision is made, the Executable Process Model Generator will insert the executable process model with partitioning instructions in the Process Description Repository. Later this executable is inserted in an Execution Engine which will coordinate the execution of the process.

We can approximate the possible process task allocation to a weighted graph, where the vertexes are the services instances and the edge weight is directly related to the cost. In its simplest form the weight is equal to one. The exact formation of the graph will have an effect on the algorithms that will be needed to find the optimal path. As an example this could be a complete graph or a directed graph etc.

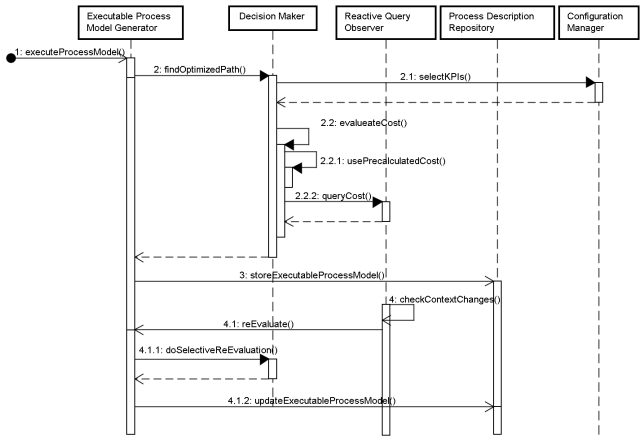

Fig. 4. Control flow for creating executable process from abstract process

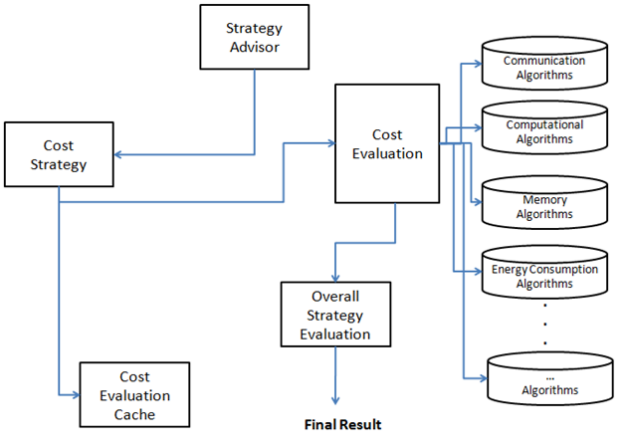

Fig. 5. Example of different strategies for cost evaluation

We assume that all metrics representing the costs (the c_ks) can be merged and successfully be represented by a single cost c_{total} (i.e. a real positive number). With this assumption, we can approximate the business process task allocation to known graph problems, such as the shortest path problem. For instance, given a connected weighted graph, we seek the shortest path i.e. trying to solve the all-pairs shortest path problem, which will provide the business process allocation with minimum overall cost. Example algorithms to solve this graph problem include the Johnson [9] and the Floyd-Warshall [5] algorithms. Another example is provided at a later section.

C. Cost Strategy

The cost strategy is based on cost dimensions that can be measured on the service host. Each cost dimension is associated with specific constraints, which define the limits of changes in this variable before triggering a re-calculation of the tasks allocation.

As depicted in Figure 5, according to different cost calculation strategies, each cost dimension will have a different contribution to the overall cost, which is weighted according to its significance at the specified time or business process.

1. Separate process P into tasks T_1, ..., T_m
2. For each task T_i do
   a. Find the complete set of service instances $S_i=\{s_{i,1}, ..., s_{i,n}\}$. Each element in S_i represents a service instance capable of executing task T_i
3. For each task T_i from T_m to T_1 do
   // going backwards through the graph
   a. leastCost = $+\infty$
   b. servicePath[i] = null
   c. For each service instance $s_{i,j}$ element(S_i)
      i. leastConnectionCost = $+\infty$
      ii. For all outgoing connections l_k
         1. leastConnectionCost = min(leastConnectionCost , cost of connection l_k + c_{total}[i+1, k])
      iii. c_{total}[i, j] = cost to execute service $s_{i,j}$ + leastConnectionCost
      iv. if(c_{total}[i, j] < leastCost)
         1. leastCost = c_{total}[i, j]
         2. servicePath[i] = $s_{i,j}$
4. Return servicePath

Fig. 6. Business Process Task Allocation Algorithm

The Strategy Advisor is a component that contains several pre-defined strategies that can be selected based on specific criteria (e.g. topology, nature of process, density of the network, communication). Each strategy uses a specific set of algorithms to calculate the cost associated with the KPIs (e.g. performance, time-to-deliver results etc).

The consideration of the Strategy Advisor is optional, as is also the Cost Evaluation cache which contains partial evaluations already done in order to speed-up the process. Once the cost evaluation process starts several algorithms and their respective implementations are selected.

1) Example for Business Process Task Allocation Algorithm: One implementation example of a business process task allocation algorithm is presented in the pseudo code in Figure 3. This algorithm is based on the algorithm defined by Dijkstra and seeks to find the overall lowest cost for the task allocation problem.

Note that each service instance can either run on a conventional server (information service) or on a constrained, embedded device (real world service). An example for service execution on a constrained device is a temperature service. If someone wants to read a temperature value of a specific location some cost of sensing is involved. Alternatively it could be cheaper to use an information service, e.g. from a weather information service provider.

The algorithms can be either executed in a distributed manner close to the location of service execution, i.e. on particular smart devices, or alternatively in a central manner. For the second option it would be useful to have outgoing cost of connecting $s_{i,j}$with any $s_{i+1,k}$ readily available, e.g. in the service monitor. The pseudo code of the algorithm is shown in Figure 6.

This algorithm has a complexity of $O(max(deg(s_{i,j})+|T|)$.

D. Reactive Query Observer

The Reactive Query Observer's role is the passive monitoring of cost dimensions from services and devices that are available in the system. This information is useful when the

availability and properties of services change once a specific process description has been created. The Reactive Query Observer is programmed using a set of rules that specify constraints that need to be followed at any time. Once the rules are programmed into the system, the systems will continuously receive messages from various devices and services within the system. This information is available any time to be used by the Executable Process Model Generator so that processes can be re-composed using the information available. The newest information is persisted into a database so that it is accessible to different components in the system, consisting of e.g. events and up-to-date status information about all the services available to the system. The the different services that have been allocated to a specific business process task are monitored and their status and quality of service are continuously updated. This information will be directly made available to the Decision Maker module, which ensures that the newest information available is used to react to changes in the network of devices.

When new event notifications are received, the Reactive Query Observer must check all the existing business processes that could be affected by changes in services, and triggers an event to the Executable Process Generator to update the previous process using the newest information. All these notifications will be done using lightweight, yet reliable messaging systems. The core of the Reactive Query Observer integrates a module for performing Complex Event Processing (CEP), so that meaningful events can be identified and correlated across all the events received by the query observer. This not only allows a drastic reduction of data that will be output by the Reactive Query Observer to other modules in the system, but also the possibility to identify meaningful eventing patterns and correlation analysis.

E. Process Description Repository

This component is a passive storage container, holding the enhanced process models along with the partitioning instructions, i.e. the fragments of the business process along with the instructions on which execution engine do deploy each of the fragment. This may be implemented by a database or any structured persistent storage method. An administrator or application may choose to take such an enhanced process model with partitioning instructions and deploy it on the central execution engine. That engine may be equipped with a deployment mechanism to deploy the process parts according to the attached partitioning instructions.

F. Execution Engine for Executable Process Model

The execution engine for the executable process model is responsible for performing two tasks: (1) execution of the central part of the partitioned process and (2) deployment of the process fragments that are supposed to be executed by lightweight and/or remote execution engines that are available close to or directly on the devices. No decisions have to be made at this point, since the executable process model generator already calculated and added information to the process model regarding where each process fragment should run. The central execution engine simply sends the process fragment descriptions to the relevant execution engines. A new instance of a process is created when the central execution engine receives a message that is marked as the starting point for a new process instance. It will create the new instance, and (given that it involves interactions with remote process fragments) may hand over control to remote process fragments and obtain back control in case of successful remote execution. The process description may contain a state that is marked to terminate the current instance. If a given instance reaches this state, it is terminated. The central engine, as well as the lightweight distributed engines may implement protocols (like time out mechanisms and message integrity checks) to deal with messages being lost and inconsistency of process state about a process instance between several execution engines.

IV. Application Scenario

In SOA-based supply chain management, operations like are focused to be collaborative and automated.

The comprehensive process management introduced in this paper can e.g. be used to automate tasks in collaborative supply chain management. This includes order creation, processing and inventory, and dependent tasks like procurement, production, and distribution.

Consider a manufacturing site where sensors and actuators are deployed. Examples of sensors include gas sensors, smoke detectors, motion sensors, or water sensors; example actuators could be sprinkler systems or fire shutters. These technologies are used to increase the overall safety of a manufacturing site, which is not only of interest to the plant owner, but also the company's insurer. Insurance companies could recommend the deployment of such technology that prevents incidents and offer reduced insurance rates if certain safety requirements are fulfilled. In addition, such technology of cause also can save lives and reduce loss expenses.

As there are many devices of the same kind available in the given example, the same service type is offered a large set of smart devices. In order to execute the overall business process of monitoring the safety within a production plant, it is necessary to select which services should be used in which sequence. We assume a network of wireless sensor/actuator nodes.

From an economical perspective, it does not make sense to use every available service frequently, as one critical problem of the deployment of sensor/actuator nodes is short battery life. This in turn requires efficient energy-aware algorithms that allow minimizing the frequency of battery replacements. The goal of these algorithms is to find a suitable dynamic business process composition that, based on the available services, reduces the frequency of battery replacements while still guaranteeing a required level of safety. As for the input parameters of the cost evaluation that is part of the algorithm, the following context information can be used: remaining battery power, cost for the invocation of a sensing/actuating service, location of the sensor/actuator, etc. The cost associated

with a specific sensing/actuating service is therefore a function of that context information, which determines whether a given device should execute the service.

Since we face highly dynamic environments where devices join and leave the network, we also need to track the dynamics of the system and react to changing context information (as described in the section about the Reactive Query Observer). Especially, context like the remaining battery power changes frequently. As a concrete example, two water sensors located close-by can execute their sensing service alternately or in a similar pattern based on their remaining battery power instead of sensing simultaneously. If one of them is likely to fail soon because of low battery voltage, the process distribution is re-calculated to switch to its alternative neighbour.

V. Related work

The work in [11] describes a way of decentralizing the execution of BPEL processes by partitioning them into parts and deploying the parts in decentralized locations. This is similar to what is provided by the Executable Process Model Generator and can be considered an example implementation of it. In [1] the authors further talk about design-time and run-time issues of decentralized process execution. The approach does not take into account a changing and unreliable service set as it is offered by a set of devices that we assume in this work: dynamic, potentially mobile and potentially wirelessly coupled devices.

In [7] and [8], Hackman et al. present practical work in form of a lightweight BPEL engine that is small enough to be deployed on mobile devices that are able to run the reduced Java ME version of the Java VM (such as mobile phones, PDAs, set-top-boxes and other embedded devices). This open source software could be used to realize local orchestration engines. However, it only covers this aspect of the whole execution mode that we propose.

Friese at al. describe in [6] an approach for peer-to-peer process healing in case of errors. They use a distributed discovery process to create a distributed service repository. All interaction of the process instances with service are not executed directly, but through this distributed repository that will redirect a request to an alternative service if its destination service is not reachable. This layer of indirection hides service failures and increases reliability of process execution. Although this approach has similar goals with respect to process execution healing, it is not applicable for services provided by constrained devices. It requires significant processing power that is not available on distributed devices, but could be realized on more powerful gateway devices that are deployed close to the devices providing the services. All in all, the approach only covers a fraction of our solution (namely Reactive Query Observer and discovery / monitoring) and does not try to partition a process and deploy its parts.

VI. Conclusions and future work

This paper laid out an approach for reliable execution of processes on a networked of unreliable, embedded devices. It introduced a way of assessing the performance of a particular service orchestration by taking into account various state information about devices and services in the network and summing them up to a single *technical cost* figure. This cost function is used as an objective function to identify an optimal solution in the search space of process fragments and their deployment to the central or some distributed orchestration engines. The computed distribution is deployed and monitored; corrective re-calculation is triggered if the current deployment degrades w.r.t. the objective function.

The approach, although conclusive by itself, still needs to be evaluated for its feasibility to find optimal process distributions in large simulated and real installations. An interesting quantitative evaluation would be to compare the additional computational effort spent on establishing and maintaining optimal process distribution, with the amount of resources saved.

Acknowledgment

The authors would like to thank the European Commission and the partners of the SOCRADES (www.socrades.eu) project for their support.

References

[1] G. B. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 134–143, New York, NY, USA, 2004. ACM.

[2] S. Chan, C. Kaler, T. Kuehnel, A. Regnier, B. Roe, D. Sather, J. Schlimmer, H. Sekine, D. Walter, J. Weast, D. Whitehead, and D. Wright. Devices Profile for Web Services. Microsoft Developers Network Library, May 2005. http://specs.xmlsoap.org/ws/2005/05/devprof/devicesprofile.pdf.

[3] L. M. S. de Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. Socrades: A web service based shop floor integration infrastructure. In *IOT*, pages 50–67, 2008.

[4] E. Fleisch and F. Mattern, editors. *Das Internet der Dinge: Ubiquitous Computing und RFID in der Praxis:Visionen, Technologien, Anwendungen, Handlungsanleitungen*. Springer, 2005.

[5] R. W. Floyd. Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345, 1962.

[6] T. Friese, J. Muller, and B. Freisleben. Self-healing execution of business processes based on a peer-to-peer service architecture. In *Proc. of ICAC*. Springer, 2005.

[7] G. Hackmann, C. Gill, and G.-C. Roman. Extending BPEL for interoperable pervasive computing. In *Proceedings of the 2007 IEEE International Conference on Pervasive Services*, pages 204–213, 2007.

[8] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman. Sliver: A bpel workflow process execution engine for mobile devices. In *Lecture Notes in Computer Science*, volume 4294, pages 503–508, 2006.

[9] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *J. ACM*, 24(1):1–13, 1977.

[10] S. Karnouskos, O. Baecker, L. M. S. de Souza, and P. Spiess. Integration of SOA-ready Networked Embedded Devices in Enterprise Systems via a Cross-Layered Web Service Infrastructure. In *12th IEEE Conference on Emerging Technologies and Factory Automation*, 2007.

[11] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing execution of composite web services. In *Proceedings of OOPSLA04, Vancouver, Canada*, 2004.

[12] M. Schleipen. Opc ua supporting the automated engineering of production monitoring and control systems. In *Proc. IEEE International Conference on Emerging Technologies and Factory Automation ETFA 2008*, pages 640–647, 15–18 Sept. 2008.

[13] V. Stirbu. Towards a restful plug and play experience in the web of things. In *Proc. IEEE International Conference on Semantic Computing*, pages 512–517, 4–7 Aug. 2008.