Chapter 1

Process Programming in the Service Age: Old Problems and New Challenges

Gianpaolo Cugola, Carlo Ghezzi, and Leandro Sales Pinto

Abstract Most modern software systems have a decentralized, modular, distributed, and dynamic structure. They are often composed of heterogeneous components and operate on heterogeneous infrastructures. They are increasingly built by composing *services*; that is, components owned (designed, deployed, maintained, and run) by remote and independent stakeholders. The quality of service perceived by the clients of such a composite application depends directly on the individual services that are integrated in it, but also on the way they are composed. At the same time, the world in which applications are situated (in particular, the remote services upon which they can rely) change continuously. These requirements ask for an ability of applications to self-adapt to dynamic changes, especially when they need to run for a long time without interruption. This, in turn, has an impact on the way service compositions are defined using ad-hoc process languages that are defined to support compositions. This paper discusses how the service setting has revamped the field of process (workflow) programming: where old problems that were identified in the past still exist now, how we can learn from past work, and where and why new challenges instead require additional research.

1.1 Introduction and Historical Perspective

Software is the driving engine of modern society. Most human activities are either software enabled or entirely managed by software. Examples range from health-care and transportation to commerce and manufacturing to entertainment and education. We are today in a stage where existing Web technology

G. Cugola · C. Ghezzi · L. Sales Pinto
Dip. di Elettronica e Informazione, Politecnico di Milano, Italy, e-mail: `\{cugola,ghezzi, pinto\}@elet.polimi.it`

allows the data available in every node to be accessed from any other node, being it static or mobile, through the network fabric. We are moving to a stage where functionalities (services) may be openly accessed and integrated to provide new functionality and serve different users.

Although the terms *(software) service* and *service-oriented computing (SOC)* are becoming widely used, they should be made more precise to better understand the nature of the problems we are currently facing. A service is a software component that provides some functionality of possible general use. Functionalities can be of different levels of complexity, generality, and granularity. Services and SOC differ with respect to *(software) components* and *component-based computing* in that services are owned (developed, deployed, run, and maintained) by independent stakeholders, who make them available for external use by multiple potential clients. Their use may be possible under certain *conditions* and may be subject to a *service-level agreement (SLA)*. The conditions and the SLA are part of a *contract* that binds service users and service providers. The conditions may include a price that the user has to pay for use. The SLA should state both the functional and non-functional properties that the service declares to offer. For example, it may indicate that the service offers a localization function having a given precision and that the average time to update the coordinates is 100 msec. In the current state of practice, contracts are often loose. Many research efforts are currently directed towards making them formal, and hence enforceable. The push to progressing in this direction has both economic and legal motivations.

Service-oriented computing has promise, but it also raises new problems. The promise is that in the future, one may expect a real service marketplace to become available, where even inexperienced users might be empowered by their ability to access a large variety of useful functionalities. The problem is, realizing this promise requires significant technological advances. For example, the way existing services may be discovered and the process languages or notations through which users may compose them are still quite primitive and hard to master. The hope that even non-technical users might become directly engaged is far from being real. A key obstacle is the openness and instability of the environment in which clients and services are immersed today. The environment changes continuously and unpredictably. Existing services may be discontinued or they may change in a way that violates their SLA and invalidates the client's expectations. They may change the conditions under which clients may use them. Clients may also change their expectations, or the context in which a service is requested may change, requiring a different service to be selected to address the new needs. New services may also become available, which might give better response to the clients' needs. How can this dynamic complexity be managed?

Defining and managing *service orchestrations* in an open and evolving environments is hard. It is especially hard if the proposed solution is based on traditional "programming" approaches adopted by current mainstream workflow languages. The "orchestration code" needs to take care of an intricate control flow in which one tries to capture all possible ways things can go wrong and react to exceptional conditions to continue to meet the requirements in the presence of anticipated and unanticipated changes.

This situation has very strong similarities to what was discovered in the late 1980s and in the 1990s in the research area on *software processes*. This area was mostly boosted by Osterweil's seminal work [17]. Osterweil recognized the need to formalize the software development process so that it could be analyzed, improved, and automated. This area was sometimes referred to using the term *process programming*, although the low-level term "programming" does not do justice to the real essence of Osterweil's proposal. Rather, the idea was that software processes were important conceptual entities to understand, model, and possibly automate. Indeed, the same concept was later applied to other human-intensive domains besides software development, where the term *workflow* instead of *process* became more commonly used[1].

One of the important findings of the work on (software) processes was that because of the active and creative role of humans in the process, *deviations* [8] were important to handle [20, 3]. The software process, in fact, supports humans and manual activities as well as automated tools. Unlike tools, humans cannot be seen as "subroutines" to invoke to get fully predictable results. Moreover, humans can tolerate inconsistencies, whereas tools seldom can. Finally, because processes are long-running entities, they need to evolve as the situation may change during the course of execution. Having recognized these distinctive feature, the process work in the 1990s sought ways to model flexible processes through sophisticated mechanisms and studied how to manage deviations and inconsistencies arising in the process enactment. This past work can be classified in three main directions:

Process programming with exceptions. A number of approaches investigated how to adapt the exception handling constructs that are supported by standard programming languages for inclusion in languages intended for process definition and automation. The emphasis here is on using a process language, as in Osterweil's original proposal, to program the process. Perhaps the most completely developed approach is the APPL/A language [20], which is based on an imperative paradigm. The idea of using exceptions has the obvious advantage that the *normal* process flows are clearly distinguishable from the *exceptional* flows in the process description. This allows for a certain degree of separation of concerns and supports a cleaner programming style than handling exceptional conditions through conventional *if–then–else* constructs. The main drawback of this approach is that it requires all possible exceptional conditions to be identified before writing the process code. This can be quite restrictive in highly dynamic contexts in which new and unanticipated cases may arise.

[1] In this paper, the terms *process* and *workflow* will be used interchangeably.

Reflective mechanisms. Through reflection, languages support reasoning about, and possibly modification of, programs. Reflective features are often available in conventional programming languages. They have been also proposed and experimented within process languages. As an example, in our past work on the SPADE environment [4], we developed a fully reflective process modeling language (SLANG) based on Petri nets, which allows meta-programming. That is, in SLANG one can develop a process whose objective is to modify an existing process or even an existing process instance. The potential advantage of such an approach over the previous one is clear: the process model does not need to anticipate all possible exceptional situations, since it can include the (formal) description of how the process model itself can be modified at execution-time to cope with unexpected situations. The main drawback of this approach is that it may bring further rigidity into the approach: not only the process must be modeled (or "programmed") in all detail, but so also must the meta-process, i.e., the process of modifying the model itself.

Flexible approaches. Both previous cases are based on the assumption that a precise and enforceable process model is available and there is no way to violate the prescribed process. In other terms, there is no way to treat a deviation from the process within the formal system. Reflective languages support changes to the process, but all possible changes must follow a predefined change process, i.e., again there is no way to "escape" from a fully defined, prescriptive model. The key idea to overcome this limitation was to abandon the ambitious but unrealistic goal of modeling every aspect of the process in advance, following an imperative, prescriptive style, to focus on certain constraints that should be preserved by the process, without explicitly forcing a pre-defined course of actions. Any process that satisfies the constraints would thus be acceptable. This brings a great flexibility in process enactment, avoiding micro-management of every specific issue while focusing on the important properties that should be maintained. Usually, these approaches are coupled with advanced runtime systems that support the users in finding their way through the actual situations toward the process goals, while remaining within the boundaries determined by the process model. An early example of this approach is described in [7].

In the reminder of this paper, we describe our current work, which focuses on process programming for service compositions. We developed a language, called DSOL, which can be classified in the last category. We propose a *declarative* approach to model service orchestrations, i.e., workflows that compose existing services to build new ones. As a result, we obtain a language that is easier to use and results in more flexible and self-adapting orchestrations than the existing mainstream languages adopted in the area, like BPEL and BPMN. An ad-hoc engine, leveraging well-known planning techniques, interprets such models to support automatic dynamic service orchestration at run-time.

The rest of this contribution is organized as follows. Section 1.2 presents a deeper analysis of the deficiencies of current mainstream service composition languages. Section 1.3 looks back to identify similar issues that were recognized in the past and the solutions provided by research at the time. We also identify what is new in the current setting and how all this may drive the search for new solutions, like DSOL (Section 1.4). Section 1.5 draws some conclusions and discusses future work directions.

1.2 Limitations of Currently Available Orchestration Languages

Throughout the last two decades, different approaches were taken towards socalled *process programming* using workflow languages. Several programming and modeling languages were defined in an attempt to best define and automate different kinds of processes. More recently, the advent of SOC has attracted much research into the area of business processes, to provide foundations for formalization, automation, and support to business-to-business integration, where services provided by different organizations are combined to provide new added-value services that can be made available to end users.

Two languages emerged as the de-facto standards for modeling service orchestrations: BPEL [1] and BPMN [21]. Although the two have some differences [18], they share a number of commonalities that result in the same limitations in modeling complex processes. In particular, both adopt an imperative style, in which service orchestrations are modeled as monolithic programs that must capture the entire flow of execution. This requires service architects to address every detail in the flow among services — they must explicitly program all the sequences of activities and take care of all dependencies among them, consider all the different alternatives to accomplish the orchestration goal, and forecast and manage in advance every possible fault and exception that may occur at run-time.

To be more precise about this issue, consider the following example. Suppose we have to orchestrate some external services to buy tickets for a (night) event, and to arrange for transportation and accommodation for those participating in the event. Initially, a participant provides the name of the city where she lives, the event she wants to attend, the relevant payment information, and her desired transportation and accommodation types. The first action to perform is buying the ticket, followed by booking transportation, which can be arranged either by plane, train, or bus (the participant may express a preference). After booking the transportation, the accommodation (hotel or hostel, in this order of preference, unless explicitly chosen by the user) must be booked. In general, the preferred option is to book the transportation in such a way that the participant arrives at the event's location the day before the event and departs the day after, booking two nights at

```
...
<scope name="EventPlanning">
 <scope name="BookTransportation">
  <if>
   <condition>
    <!-- preferredTrans equals airplane or  preferredTrans is null -->
   </condition>
   <scope name="BookFlight">
    <compensationHandler>
     <!-- Cancel flight reservation -->
    </compensationHandler>
    <invoke operation="bookFlight"
            inputVariable="transDetails" outputVariable="flightBooked" .../>
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals train or
          (preferredTrans is null and not flightBooked) -->
   </condition>
   <scope name="BookTrain">
    <compensationHandler>
     <!-- Cancel train reservation -->
    </compensationHandler>
    <invoke operation="bookTrain"
            inputVariable="transDetails" outputVariable="trainBooked" ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- preferredTrans equals bus or (preferredTrans is null and
           not flightBooked and not trainBooked -->
   </condition>
   <scope name="BookTrain">
    <compensationHandler>
     <!-- Cancel bus reservation -->
    </compensationHandler>
    <invoke operation="bookBus"
            inputVariable="transDetails" outputVariable="busBooked" ... />
   </scope>
  </if>
  <if>
   <condition>
    <!-- not (trainBooked or flightBooked or busBooked) -->
   </condition>
   <throw faultName="TransNotBooked" />
  </if>
 </scope>
</scope>
...
```

Listing 1.1 Booking transportation in BPEL

a nearby hotel/hostel, to allow for free time to visit the place. It is also acceptable to stay a single day (the day of the event) if this is the only way to successfully organize the trip.

To model this orchestration using BPEL (and the same is true for BPMN), we must explicitly code all possible action flows. Unfortunately, there are many. Indeed, even if we do not consider possible exceptions to the mainstream process, this requires addressing alternative actions (e.g., booking a train is not required if a plane has already been booked), actions that must be done in sequence (e.g., buying the ticket before finding transportation), and actions which depend on the result of other actions (e.g., the choice of transportation depends from the preference of the user). This is quite significant and complex.

Listing 1.1 shows a code snippet that expresses the alternatives for booking the transportation in BPEL. It is easy to observe how convoluted and hard to read it is, especially if we consider that this is just a small fragment of quite a simple case study and that we have not considered possible exceptions. Indeed, the situation becomes much more complex when run-time exceptions, such as a failure while invoking an external service, have to be considered. We need to be able to forecast these and add code to manage them, designing alternative paths and including code to undo actions that must be retracted when alternative paths are followed.

It is our belief that this is mainly a consequence of the imperative paradigm adopted by mainstream orchestration languages, which closely resemble (imperative) programming languages, forcing service architects to precisely and explicitly enumerate all possible flows of actions, with the additional drawback that the code for fault and compensation handling is mixed with the main process flow.

As we mentioned in the introduction, we believe that possible solutions to mitigate these problems can be found by examining research solutions in the past decade in the area of software process modeling. We survey the relevant part of this research in the next section, while in Section 1.4 we present a novel approach to service orchestration, which leverages our experience in that area [4, 7] to abandon the imperative way of modeling orchestrations in favor of a declarative style. The run-time system of the declarative language may use known planning techniques to automatically determine the exact order in which different steps can be performed and how to operate in case of both expected and unexpected faults.

1.3 Looking to the Past to Take Inspiration for the Future

Although service composition languages, like BPEL and BPMN, were born in a different environment from the one in which software process modeling languages were proposed, they share many commonalities. Indeed, both software processes and generic business processes are long-lived, complex, dynamic entities that must interact with an external environment that they usually cannot fully control. Such environments are inevitably subject to changes, which are hard to predict and almost impossible to prevent. Thus, changes often lead to unexpected situations that force the process to deviate from the originally intended course of actions.

This commonality suggests that current research on service composition languages could profit from taking inspiration from past research on software process modeling, in particular with respect to the mechanisms to model and handle exceptions and deviations.

The most common and often used solution to the problem of managing exceptional situations is by providing specific language constructs to describe them and to model the actions to manage them, as in languages like AP-PL/A [20] and, later, Little-JIL [15]. As we mentioned in Section 1.1, this approach, also used by mainstream business process languages like BPEL and BPMN, and by several Workflow Management Systems [19], is limited in that it can only handle expected exceptions, forcing the process modeler to forecast at design time all possible situations that may lead to a deviation from the standard course of actions. This limitation is exacerbated by the fact that languages which follow this approach usually adopt a normative paradigm of modeling and a rigid runtime system, which do not allow to deviate from the model at process execution time, if something unexpected happens.

Even if we ignore the difficulty of anticipating, at design-time, everything that could go wrong at run-time, just modeling the forecasted exceptions is a cumbersome and error-prone task. To address this problem, a number of exception handling patterns were proposed [16]. They help process designers to identify and reuse existing solutions, simplifying the development and maintenance of process models. These patterns could be reused easily in the domain of service compositions. However, the approach would still be based on an explicit enumeration of all expected exceptions and on explicitly programming how they can be handled.

Some software process execution environments—such as SPADE [4], OASIS [12], Endeavors [5], EPOS [11], and IPSE 2.5 [6]—adopted reflective languages, through which process models and even their running instances may be accessed as data items to be inspected and modified at process enactment time. This approach allows the *meta-process*, i.e., the process of changing the running instance of the model, to be also modeled, as a special step of the process itself. While this brings an unprecedented level of expressiveness to the language, it also requires a lot of effort from the process modeler, who is asked not only to model the software development process, but also the meta-process, in all details. For this reason, reflection is considered an effective approach to manage major exceptional situations, which require a radical departure from the originally modeled process, and particularly those situations that are expected to occur again. For the other (minor but more frequent) cases, which happen sporadically and require quick responses, other approaches are required.

One example of such approaches in the area of software process modeling is a system we developed a few years ago: PROSYT [7]. Three intuitions guided the design of PROSYT:

- the need to abandon the normative approach to process modeling and consequently, the imperative style that was typical of previous process languages;
- the need to add flexibility into the runtime system, to allow the users involved in a process to deviate readily from the expected course of actions if an unexpected situation arises; and
- the need to abandon the activity-oriented approach to modeling, which often results in focusing only and too early on those aspects of the process that are directly related to the specific course of actions that the process

modeler had in mind. Conversely, we preferred an approach focusing on the general constraints that guide and govern the domain in which the process operates.

Moving from these ideas, we designed a language called PLAN–Prosyt LANguage–which adopts an *artifact-based* approach to modeling. PLAN allows process designers to focus on the artifacts produced during the process (together with the basic operations to handle them), rather than on the activities that fragment the process into elementary steps. This shift is similar to the transition from imperative programming languages to the object-oriented paradigm. Moreover, in PLAN the expected flow of actions is never stated explicitly. Instead,the constraints (i.e., pre-conditions) to invoke operations on each artifact type, and a set of invariants that have to hold for each artifact, are the basis for building a process model and for guide the control flow.

At process enactment time, an advanced runtime support system interprets the PLAN model, allowing the users involved in the process to execute actions in the order they find more effective to pursue the process goals under the actual circumstances. When something unexpected happens, users are allowed to deviate from the modeled process temporarily, by violating some of the pre-conditions to invoke actions, as long as the overall invariants hold. This allows small deviations to be handled without the need to modify the process model. At the same time, PROSYT allows different consistency-checking and deviation-handling policies to be specified and changed at runtime on a per-user, per-artifact basis, to precisely control the level of deviation allowed. This brings great flexibility in process enactment and avoids micro-management of every specific issue of the process, while focusing on the fundamental properties that have to be guaranteed.

Other software process execution environments adopted approaches similar to those introduced in PROSYT. For example, SENTINEL [9] adopts an activity-based approach, in which software processes are modeled as a collection of state machines. State transitions are guarded by preconditions. To guarantee safe behaviors, transitions may legally occur when all the preconditions hold; however, state transitions can also be triggered by human interaction, and in this case, some preconditions can be explicitly violated, allowing minor deviations to the process model. The process is allowed to continue enactment as long as no invariant assertions — which define safe states — are violated. If violations occur, a reconciling process must be carried out to fix corrupted state variables, after which the process execution can resume in a safe way.

Provence [13] adopts a different approach, in which the process model is used to monitor the process, rather than automating it. Provence is based on an event-action system, called Yeast [14]. When relevant events occur in the actual process, Yeast notifies Provence. Provence matches those events with the process model to trigger actions and state changes. Although Provence does not have any specific support to handle unforeseen situations, its approach provides an interesting way to support detection of changes in the process state (which can become inconsistent with respect to the process model) as soon as they occur.

PEACE [2] and GRAPPLE [10] follow a goal-oriented approach, implemented using a logic-based formalism (in PEACE) and ad-hoc planning techniques (in GRAPPLE). In both systems, the process model is defined only through the objectives that must be satisfied, while at enactment time the runtime support system finds the best ordering of activities to accomplish the given goals. This approach increases the environment flexibility, reducing the need for deviations.

The lessons learned by looking at the systems above were at the core of our recent work in the area of service orchestration, which lead to the development of DSOL, the subject of the next section.

1.4 The DSOL Approach

Our main motivation in defining a new language for service orchestrations was the fact that none of the currently available languages designed for this purpose were able to cope efficiently with unforeseen exceptions, a feature we consider fundamental for a system that has to operate in an open, dynamic world.

After some initial research, looking also at past experience in process modeling, we realized that to achieve this goal, we have to rethink the way in which service orchestrations are defined: we need a paradigm shift. As we already noted, the imperative programming style adopted by most process languages seems to be inappropriate to supporting flexible orchestrations for several reasons: (i) processes are modeled in a normative and rigid form, making runtime adaptations hard to achieve; (ii) they must capture completely different aspects within a single monolithic model, from control flow to exception and compensation handlers; (iii) they require sophisticated programming skills, precluding SOP from reaching a key goal: empowering even non-technical users to build their own service orchestrations.

The language DSOL—*Declarative Service Orchestration Language*—that we defined to support service orchestration and its runtime support system, adopts a radically different, *declarative* approach. With DSOL, we aim to achieve two different goals: (i) simplify the definition of complex service orchestrations, in order to empower even non-technical users, such as domain experts; and (ii) increase the possibility of runtime adaptations by letting orchestrations evolve when unforeseen situations happen, or when the orchestration's requirements change.

A service orchestration modeled in DSOL includes different aspects, which are defined separately using different idioms, possibly by different stakehold-

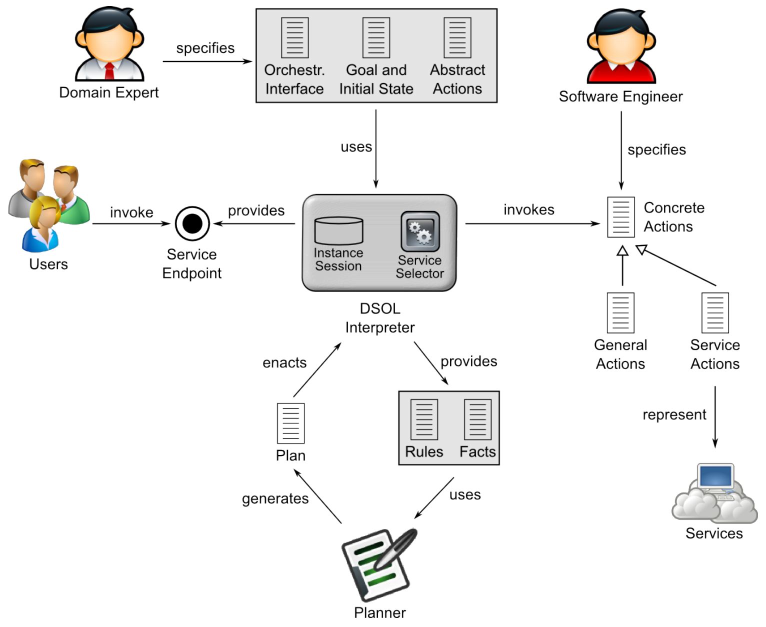

Fig. 1.1 The DSOL approach to service orchestration

ers who bring their own competencies. Specifically, as shown in Figure 1.1, a service orchestration in DSOL includes the following elements:

- the definition of the *orchestration interface*, i.e., the signature of the service that represents the entry point to the orchestration;
- the *goal* of the orchestration, as a set of facts that are required to be true at the end of the orchestration. This is is usually expressed by a domain expert, who is not necessarily competent in software development;
- the *initial state*, which models the set of facts that one can assume to be true at orchestration invocation time (usually described by the same domain expert who formulates the goal);
- a set of *abstract actions*, which model the primitive operations that can be invoked to achieve a certain goal and are typical of a certain domain. They are described using a simple, logic-like language that can be mastered even by non-technical domain experts;
- a set of *concrete actions*, one or more for each abstract action, written by a software engineer to map abstract actions into the concrete steps required to implement the operation modeled by the abstract action, e.g., by invoking an external service or executing some code.

At orchestration invocation time, the *DSOL Interpreter* translates the goal, the initial state, and the abstract actions into a set of *rules* and *facts* used by the *Planner* to build an abstract plan of execution, which lists the logical steps through which the desired goal may be reached. The Interpreter then enacts the plan by associating each step (i.e., each *abstract action*) with a *concrete action* that is executed, possibly interacting with external services. If everything goes as expected and all the steps are executed successfully, the workflow terminates and control is transferred back to the client.

Unfortunately, as we have indicated, real world service orchestrations may encounter situations that prevent them from terminating normally. To tolerate exceptions to the standard flow of actions—both expected and unexpected—DSOL provides both specific language constructs and ad-hoc run-time facilities. For the former, it is possible to associate different concrete actions with the same abstract action. This gives the Interpreter the ability to try different options to realize each step of a plan. Indeed, when an abstract action A has to be executed, the Interpreter tries the first concrete action implementing A. If this fails (e.g., the service is unavailable), the Interpreter automatically captures the exception and tries the second concrete action, which invokes a different external service, which hopefully is available and executes correctly.

If, however, none of the available concrete actions can execute correctly, a second mechanism is available: the ability to build an *alternative plan* to execute when something bad happens at run-time. That is, if the Interpreter cannot realize a step (i.e., an abstract action invoked with specific parameters) of the current plan, it invokes the Planner again, forcing it to avoid the failed step. This causes a new plan to be computed that excludes the failed step. By comparing the old and new plans, considering the current state of execution, the Interpreter is able to calculate the set of actions that must be *compensated* (i.e., undone), as they have already been executed but are not part of the new plan.

In summary, by combining the ability to specify different implementations (i.e., concrete actions) for each step of a plan, plus the ability to rebuild failed plans in search of alternative courses of actions, possibly achieving different but still acceptable goals, our language and run-time system allow robust orchestrations to be built in a natural and easy way. Indeed, by combining these mechanisms, DSOL orchestrations are able to work around failures and any other form of unexpected situation automatically, by self-adapting to changes in the external environment.

This fundamental characteristics is also achieved thanks to the DSOL approach to modeling orchestrations, which focuses on the primitive actions typical of a given domain more than on the specific flow of a single orchestration. This approach is reminiscent of the solution we experimented with PROSYT [7], which decomposes processes in terms of the artifacts involved in it, leaving aside the traditional top down decomposition into activities. The goal is the same here: to maximize the chance that when something bad happens, even if not explicitly anticipated at modeling time, the actions that may overcome the current situation have been modeled and are available to the Planner and Interpreter (to the user, in the case of PROSYT).

This brief description shows the main advantages of the DSOL approach w.r.t. traditional ones:

1. It achieves a clear separation among the different aspects of an orchestration: from the more abstract ones, captured by goals, initial state, and abstract actions, to those closer to the implementation domain, captured by concrete actions.
2. It meets one of the original goals of service-oriented computing; i.e., it involves users who are not expert in software development into the cycle.
3. By focusing on the primitive actions available and letting the actual flow of execution be built automatically at run-time through the Planner, it allows orchestration designers to focus on those aspects that are typical of a certain domain and remain stable over time, ignoring the peculiarities of a specific orchestration, which may change when requirements change. This last property also holds the promise of increasing reusability, since the same abstract and concrete actions can be reused for different orchestrations within the same domain.
4. By separating abstract and concrete actions, with several concrete actions possibly mapped to a single abstract action, the DSOL Interpreter can find the best implementation for each orchestration step and to try different routes if something goes wrong at run-time, in a fully automated way.
5. Because abstract actions only capture the general rules governing the ordering among primitive actions, the Interpreter, through a careful replanning mechanism, can automatically overcome potentially disruptive and unexpected situations that occur at run-time.
6. The modularity and dynamism inherent in the DSOL approach allow the orchestration model to be changed easily at run-time, by adding new abstract/concrete actions when those available do not allow the orchestration's goal to be reached.

1.5 Conclusions

Service-oriented computing shows great promise. Through it, an open and dynamic world of services becomes accessible for humans, who can be empowered by useful application components that are developed by service providers and exposed for possible use. Service-oriented computing may also generate new business. For example, it supports service provision by brokers who can integrate third-party services and export new added-value services. Because services live in open platforms, the computational environment is continuously evolving. New services may be created, old services may be discontinued, and existing services may be evolved by their owners.

Service-oriented computing raises several important challenges that need to be addressed by research to become successful. In particular, how can the new systems we build by composing existing services be described? How can such descriptions accommodate the need for tolerating the continuous changes that occur in the computational environment?

Service compositions may be achieved through workflow languages. Workflows describe processes through which humans interact with software components and compose them to achieve their goals. The existing workflow languages that have been developed to support service compositions, unfortunately, are still very primitive. In this paper, we argued that much can be learned from the work developed in the past in the area of software processes and software process programming. This area was pioneered by the Osterweil's work and the key challenges were identified in the keynote he delivered at the International Conference on Software Engineering (ICSE) in 1987 ([17]). Although seldom acknowledged, we believe that many of the findings reached by this research area may provide useful inspiration to tackle the problems arising in service-oriented computing.

The work we describe here traces back to the research in software processes that was originated by Osterweil and builds on (some of) the lessons learned at the time to propose a new approach to service composition that we have been recently exploring.

Acknowledgements This work was partially supported by the European Commission, Programme IDEAS-ERC, Project 227977-SMScom; and by the Italian Government under the projects FIRB INSYEME and PRIN D-ASAP.

References

1. Alves, A., Arkin, A., Askary, S., Bloch, B., Curbera, F., Goland, Y., Kartha, N., Liu, C.K., Konig, D., Mehta, V., Thatte, S., van der Rijn, D., Yendluri, P., Yiu, A., eds.: Web Services Business Process Execution Language Version 2.0. Tech. rep., OASIS (2006). URL `http://www.oasis-open.org/apps/org/workgroup/wsbpel/`
2. Arbaoui, S., Oquendo, F.: PEACE: goal-oriented logic-based formalism for process modelling, pp. 249–278. Research Studies Press Ltd., Taunton, UK, UK (1994)
3. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th international conference on Software engineering, ICSE '91, pp. 158–165. IEEE Computer Society Press, Los Alamitos, CA, USA (1991)
4. Bandinelli, S.C., Fuggetta, A., Ghezzi, C.: Software process model evolution in the spade environment. IEEE Trans. Softw. Eng. **19**, 1128–1144 (1993)
5. Bolcer, G., Taylor, R.: Endeavors: a process system integration infrastructure. In: Software Process, 1996. Proceedings., Fourth International Conference on the, pp. 76 –89 (1996)
6. Bruynooghe, R., Parker, J., Rowles, J.: Pss: A system for process enactment. In: Proceedings of the 1st International Conference on the Software Process, pp. 128–141 (1991)
7. Cugola, G.: Tolerating deviations in process support systems via flexible enactment of process models. IEEE Trans. Software Eng. **24**(11), 982–1001 (1998)
8. Cugola, G., Di Nitto, E., Fuggetta, A., Ghezzi, C.: A framework for formalizing inconsistencies in human-centered systems. ACM Transactions On Software Engineering and Methodology (TOSEM) **5**(3) (1996)

9. Cugola, G., Di Nitto, E., Ghezzi, C., Mantione, M.: How to deal with deviations during process model enactment. In: Proceedings of the 17th international conference on Software engineering, ICSE '95, pp. 265–273. ACM, New York, NY, USA (1995)
10. Huff, K.E.: Grapple example: processes as plans. In: Proceedings of the 5th international software process workshop on Experience with software process models, ISPW '90, pp. 156–158. IEEE Computer Society Press, Los Alamitos, CA, USA (1990)
11. Jaccheri, L., Larsen, J., Conradi, R.: Software process modeling and evolution in epos. In: Proceedings of the 4th International Conference on Software Engineering and Knowledge Engineering, pp. 574 –581 (1992)
12. Jamart, P., van Lamsweerde, A.: A reflective approach to process model customization, enactment and evolution. In: 'Applying the Software Process' , Proceedings of the 3rd International Conference on the Software Process, pp. 21 –32 (1994)
13. Krishnamurthy, B., Barghouti, N.S.: Provence: A process visualisation and enactment environment. In: Proceedings of the 4th European Software Engineering Conference on Software Engineering, ESEC '93, pp. 451–465. Springer-Verlag, London, UK (1993)
14. Krishnamurthy, B., Rosenblum, D.S.: Yeast: A general purpose event-action system. IEEE Transactions on Software Engineering **21**, 845–857 (1995)
15. Lemer, B.S., McCall, E.K., Wise, A., Cass, A.G., Osterweil, L.J., Stanley M. Sutton, J.: Using little-jil to coordinate agents in software engineering. p. 155. IEEE Computer Society, Los Alamitos, CA, USA (2000)
16. Lerner, B.S., Christov, S., Osterweil, L.J., Bendraou, R., Kannengiesser, U., Wise, A.: Exception handling patterns for process modeling. IEEE Transactions on Software Engineering **99**(RapidPosts), 162–183 (2010)
17. Osterweil, L.: Software processes are software too. In: ICSE '87: Proceedings of the 9th international conference on Software Engineering, pp. 2–13. IEEE Computer Society Press, Los Alamitos, CA, USA (1987)
18. Ouyang, C., Dumas, M., Aalst, W.M.P.V.D., Hofstede, A.H.M.T., Mendling, J.: From business process models to process-oriented software systems. ACM Trans. Softw. Eng. Methodol. **19**, 2:1–2:37 (2009)
19. Russell, N., van der Aalst, W., ter Hofstede, A.: Workflow Exception Patterns. Advanced Information Systems Engineering pp. 288–302 (2006)
20. Sutton Jr., S.M., Heimbigner, D., Osterweil, L.J.: Language constructs for managing change in process-centered environments. SIGSOFT Softw. Eng. Notes **15**, 206–217 (1990)
21. White, S.A.: Business Process Modeling Notation, V1.1. Tech. rep., OMG (2008). URL `http://www.bpmn.org/Documents/BPMN_1-1_Specification.pdf`