Process-based Software Tweaking with Mobile Agents

Giuseppe Valetto (CSELT, Columbia University)
Gail Kaiser (Columbia University)
{valetto, kaiser}@cs.columbia.edu

Abstract

We describe an approach based upon software process technology to on-the-fly monitoring, redeployment, reconfiguration, and in general adaptation of distributed software applications, in short "software tweaking". We choose the term tweaking to refer to modifications in structure and behavior that can be made to individual components, as well as sets thereof, or the overall target system configuration, such as adding, removing or substituting components, while the system is running and without bringing it down. The goal of software tweaking is manifold: supporting run-time software composition, enforcing adherence to requirements, ensuring uptime and quality of service of mission-critical systems, recovering from and preventing faults, seamless system upgrading, etc. Our approach involves dispatching and coordinating software agents - named Worklets – via a process engine, since successful tweaking of a complex distributed software system often requires the concerted action of multiple agents on multiple components. The software tweaking process must incorporate and decide upon knowledge about the specifications and architecture of the target software, as well as Worklets capabilities. Software tweaking is correlated to a variety of other software processes - such as configuration management, deployment, validation and evolution - and allows to address at run time a number of related concerns that are normally dealt with only at development time.

1 Motivation

Distributed software systems are becoming increasingly large and difficult to understand, build and evolve. The trend towards integrating legacy/COTS heterogeneous components and facilities of varying granularity into "systems of systems" can often aggravate the problem, by introducing dependencies that are hard to analyze and track and can cause unexpected effects on the overall system functioning and performance.

A number of best software engineering practices attack this challenge, such as component-based frameworks, Architectural Description Languages (ADLs), Aspect-Oriented Programming (AOP), etc. They mainly operate on the specification of the software at some level, aiming at better describing, understanding and validating software artifacts and their interrelationships. Those practices must be incorporated into and enacted by the development process in order to enable the creation and maintenance of quality systems, and in general result in new iterations of the lifecycle spiral when any corrective, adaptive and perfective needs arise.

A complementary avenue for addressing the aforementioned complexity is represented by the introduction of run-time automated facilities that enable some form of monitoring, control and adaptation of the software behavior – including re-deployment and reconfiguration of components without bringing the system down. We refer to such an approach as to "software tweaking". Software tweaking assumes that systems can gauge their own "health" (i.e. run-time quality parameters) and take action to preserve or recover it, by rapidly performing suitable integration and reconfiguration actions.

The scope of software adaptation made possible by tweaking is necessarily limited with respect to full-fledged re-engineering, but it can still alleviate or resolve at lesser costs a wide range of maintenance, evolution and operation problems, such as

supporting run-time software composition, enforcing adherence to requirements, ensuring uptime and quality of service of mission-critical systems, recovering from and preventing faults, seamless system upgrading, etc.

Numerous research initiatives are active on these themes. The DARPA DASADA program [23] – for instance - has recently promoted a large number of efforts aimed at achieving and maintaining high levels of assurance, dependability and adaptability of complex, component-based software at all phases of the system's life cycle: before, during and after system assembly time, including provisions for on-the-fly re-assembly and adaptation. The ESPRIT project C3DS [30] similarly tackles dynamic composition, (re)configuration and execution control of componentized services. Also other research efforts address various facets of the same challenge, such as automatic software configuration and deployment (SoftwareDock [16]), rapid composition of heterogeneous software with federations (PIE [21]) or workflow-aware middleware (Opera [14]), dynamic layout (re)programming for distributed systems (FarGo [29]), and many others.

Successful software tweaking requires a considerable knowledge of the specifications and architecture of the system, in order to detect inconsistent or undesired structure/behavior and decide what to do. Hence, it builds upon approaches that encourage the formalization of such knowledge during development, but aims at extending their usefulness to the realm of software operation. Moreover, automating the adaptation of a non-trivial system requires the presence of a software tweaking process to handle the intricacies and dependencies of adaptation procedures. Hence, we present an approach to software tweaking that is based upon process and agent technologies. On the one hand, it proposes to exploit distributed process technology to efficiently build and operate reliable and robust software products; on the other hand, it is positioned at the intersection between software processes such as evolution, configuration management, deployment, and validation, and allows to address several of their concerns not only at development time but also at run time.

2 Approach

We envision an approach that explicitly takes advantage of process technology to automate the software tweaking process with a community of software agents, whose activities are orchestrated by a distributed process engine.

We employ Worklets [1] as our tweaking agents: Worklets carry self-contained mobile code that can act upon target components and follow directives indicating their route and operation parameters. Worklets were originally conceived as a means for flexible software (re)configuration, with effects local to the target component. Each Worklet would work with the component needing configuration, deciding what to do on the basis of the component state and its own capabilities. Moreover, a Worklet could "hop" from a component to another, carrying out at each location a portion of a predetermined multi-step configuration sequence. A very simple example of this kind of reconfiguration would be dispatching a Worklet to modify the ports used for inter-communication by two components of the target system: the Worklet would carry information about the various port numbers and code to activate new ports and deactivate old ones.

The careful reader may notice that even this simplistic scenario actually calls for some degree of coordination, since the Worklet must be instructed to avoid disrupting any outstanding communications (or enable their recovery), and the switch to new ports

must happen at both sites in an "atomic" way, to preserve at all times the correctness of the inter-component interaction.

In Section 3.3, we propose a more comprehensive scenario that exemplifies the kind of problems that software tweaking must resolve and technically clarifies our approach. However it is clear that when the target system becomes substantially large and complex, with various interwoven aspects of adaptation that involve a multiplicity of components, individual Worklets cannot simply go off and do their job autonomously, neither the subtleties of that job can be practically hardcoded or scripted into them *a priori*. Successful software tweaking demands that a number of interdependent actions be performed in a concerted and timely way: some kind of "guiding hand" is necessary, which in our belief can conveniently take the form of a process enactment engine.

Process- and agent-based software tweaking derives from considerations advocating the separation of concerns between coordination and computation in distributed programming [2]. On the one hand, Worklets deliver the computational units that carry out the mechanics of modifying the system behavior. Such computational code can be written in principle in any conventional programming language that allows Worklets to interact with the target components: we currently employ Java because Java as a platform inherently offers considerable mileage with respect to the kind of mobile distributed computation that we envision. On the other hand, the tweaking process provides the coordination context: it does not deal with the local operation of a Worklet on a target component, but rather with enacting cooperative reactive and proactive tweaking procedures, scheduling and committing the overall work of cooperating Worklets, handling contingencies and exceptional courses of action, etc.
Tweaking processes are defined by codifying multi-faceted predefined and/or accumulated knowledge about the system (e.g. requirements, composability, architecture, configuration, distribution, operation) and their intended variation in an enactable process language, which becomes the coordination "programming" language for the Worklets community.

Notice that at this stage we are not overly concerned about evaluating or producing process modeling and enacting formalisms particularly suited for software tweaking. Rather, we focus on how an "ideal" or "generic" process technology can be employed to implement systems with the specific kind of *process awareness* [31] required for tweaking.

The target system must be process-aware in the sense that its components and connectors must be able to accommodate tweaking agents and expose to them appropriate internal reconfiguration functionality. However, they do not need to incorporate any process enactment capabilities, nor know anything about the software tweaking process: process-awareness is provided for them by a completely external and separate process enactment engine. The process engine - on its part - must be decentralized in many senses: it must be able to coordinate distributed task processors, typically co-located with the target system components; to distribute fragments of the overall process to those task processors and to maintain a distributed process state; to handle widely dispersed resources and/or artifacts.

Ideal requirements for a process engine for software tweaking can be summed up as multi-dimensional distribution (including full distribution of the process definition, resources, enactment data and enactment architecture) and temporal variability of those distribution dimensions. Software process, workflow and agent coordination

research have produced a number of advanced results in decentralized process technology: among them, multi-agent engines, either peer-to-peer (Juliette [4], Serendipity-II [5], Cougaar [6]), or hierarchical (OPSS [22]); multi-server Web-centered (WebWork [19], Endeavors [20]) or database-centered systems (Opera [14]); federation systems, such as APEL [21] and OzWeb [32]. Those systems and their typologies comply with our requirements at various degrees. An in-depth discussion of those requirements and a synopsis of several process engines in that respect can be found in [7].

3 Technical Description

We are building an infrastructure named KX (Kinesthetics eXtreme) to enable the *Continual Validation* of complex distributed software systems, in the context of DARPA's DASADA program [24]. Continual Validation operates on a running system to ensure that critical functioning and assurance factors are constantly preserved, by rapidly and properly adapting the system whenever the modification of field conditions demand it.

KX aims to achieve continual validation by superimposing a minimally intrusive controlling meta-architecture on top of the target system. Such meta-architecture is in charge to introduce an adaptation feedback and feedforward control loop onto the target system, detecting and responding to the occurrence of certain conditions. Generally, such conditions would indicate errors and failures of some sort, or at least undesirable behavior such as degrading performance, and can arise within components, or at inter-components boundaries (i.e., on the connectors of the target architecture).

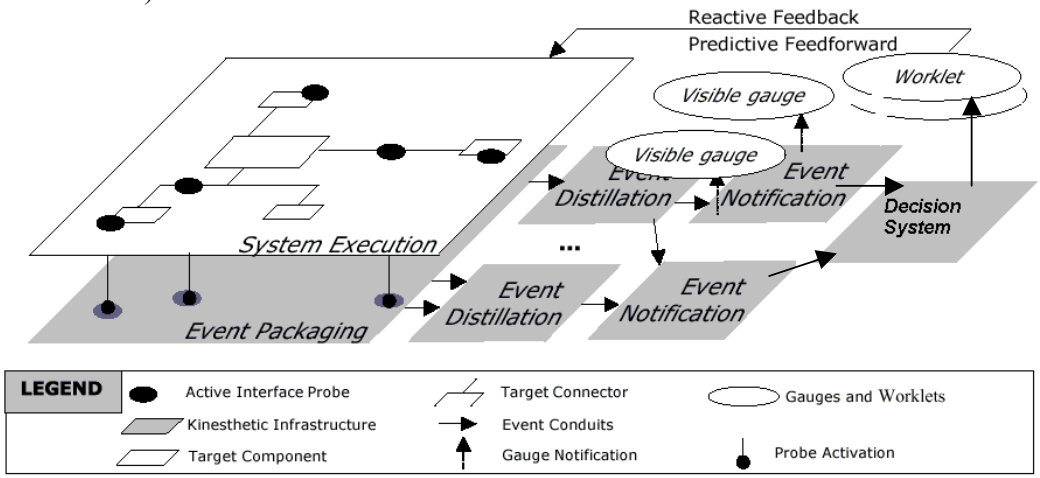

Figure 1: KX meta-architecture.

Compared to the fine-grained level of internal diagnostics and remedies that can be sometimes performed in isolation by a self-assured, fault-tolerant component, the KX meta-architecture aims to handle more global situations, perhaps involving heterogeneous components obtained from multiple sources where it would be difficult if not impossible to retrofit self-assurance. For instance, by monitoring conditions in architectural connectors and correlating them to conditions arising in dependent components, it can correctly interpret and detect mishaps like functional and performance mismatches, breaches of interface contracts and others, which may have far-reaching, domino effects on systems that are built out of a variety of COTS and proprietary components. Notice that, in order to avoid interference to or unnecessary

overriding of any self-assurance facilities present within single components, KX must be enabled to reason about some specifications of what those facilities are likely to do under externally observable circumstances.

The major conceptual elements of KX are shown in Figure 1:

- a set of probes, registering and reporting relevant information on the behavior of the target system. An approach to automatically inserting probes into the source code of a target system via *active interfaces* is found in [3]; another approach is to replace dynamic link libraries [12].
- a distributed asynchronous event bus for receiving target system events, including manufactured events such as "heartbeats", from the probes and directing them through packaging, filtering and notification facilities;
- a set of gauges to describe and measure the progress of the target system (and also the work done "behind the scenes" by the meta-architecture. Gauges in their raw form are machine-readable data packets carrying significant monitoring information that is synthesized from the event traffic in the bus via proper matching and filtering. KX renders gauges either as user-friendly visual or textual panels of Web-based consoles dedicated to human system operators and administrators, or directly as data feeds into automated decision support;
- decision support systems and agents, to respond to the conditions indicated through the gauges by determining appropriate target system adaptations;
- actuation facilities for the delivery and execution of any adaptation measures decided upon on the basis of gauge readings. As outlined in Section 2, our platform of choice is agent-based and employs Worklets as its actuators.

In the KX context, the process engine coordinating software tweaking is one possible implementation of the decision support component. It can be seen as either complementary or altogether alternative to any decision facilities devoted to the monitoring and administration of the running system on the part of humans. Process- and agent-based software tweaking allows KX to automatically implement a tighter closed control loop compared to human decision-making supports, which typically have longer reaction and response times, and lean towards open loop control. (see for example the DASADA MesoMorph [9] project, which relies on a wealth of design time - architecture, requirements - and run time information - gauge readings - to enable a human role – the *Change Administrator* – to take informed decisions on software variation and reconfiguration.) Furthermore, our approach suggests and supports the codification, materialization, and enforcement of sophisticated and explicit software tweaking processes, which sets it apart from other reactive architectures, such as that proposed in [25], in which any process remains implicit.

3.1 Monitoring and Triggering

The enactment of a fragment of the software tweaking process is triggered by the occurrence of significant conditions within the target system, which are detected and reported by the monitoring part of the meta-architecture. Conditions can be simple: for instance, a single event might be sufficient to indicate the raising of a critical exception by a target system component, and require the recovery of the failed computation. Many times, however, conditions are complex, i.e., defined in terms of partially ordered sets - *posets* – of events: for instance, only a sequence of events can hint at the progressive degradation of some service parameter, demanding the preventive replication of some bottleneck target system component in order to preserve QoS; or the (likely) crash of some other component – calling for its re-

instantiation and re-initialization - can be detected by composing timeout events for requests directed to that component, and possibly the lack of a periodic "heartbeat" event originating from the same component.

A subscription mechanism is a convenient way to declare what posets the software tweaking process is interested into, i.e., what kind of conditions arising in the target system it can handle and what information is sought about them. In practice, however, poset subscription may be complicated or impractical. Therefore, the componentized KX event bus provides *distillers*, which subscribe to those individual events that might appear in a poset, keep notice of what has been seen so far (e.g., via Finite State Automata), and report poset occurrences to *notifiers*. Notifiers then compile other, higher-level events on the basis of the content of the poset, and report them in meaningful forms to the process engine and/or any gauges.

3.2 Instantiation and Dispatching of Tweaking Agents

With respect to the monitoring subsystem, the process engine behaves in a completely reactive way. Once it is notified about a condition, the engine may initiate a complex set of interrelated tweaking operations, which typically require the instantiation of one or more Worklets, their initialization, and finally their dispatching.

Each Worklet can contain one or more mobile code snippets (in our terminology, *worklet junctions*) that are suitable for actuating the required adaptation of the target system. Junctions' data structures can be initialized with data, typically coming from the task definition, the process context, and the information contained in the event(s) that represents the triggering condition. Furthermore, any process-related configuration of Worklets is accounted for by *worklet jackets,* which allow scripting of certain aspects of Worklet behavior in the course of its route. Among them, pre-conditions and timing for junction delivery and activation, repetition of junction execution, exit conditions for the junction's work, directives to supersede, suspend, and reactivate a junction upon delivery of another one, and so on.

One important consideration is that the definition of the software tweaking process must enable the engine to reason about which worklet junctions are applicable to what target components under which conditions. Hence, substantial formalized knowledge about the specifications of the requirements, architecture and dynamics of the target system must be made available to the process. Also, knowledge about worklet junctions and their tweaking capabilities must be characterized accordingly. Formal specifications languages or Architecture Description Languages (ADLs) that allow expressing architectural events and behavior– as well as properties and constraints – may be suitable means to reach these ends. For example, the ABLE project [18] proposes to exploit Acme [26] to define a rule-base that associates "repair strategies" to events (and posets) in the system that signal the violation architectural constraints and properties. Such a rule base can represent a valid basis for defining a software tweaking process. Other languages, like Rapide [27] and Darwin [28], may be equally suitable. (Further discussion of requirements and merits of these and other approaches is out of the scope of this short paper.)

On the basis of the aforementioned knowledge, the process engine requests junctions for the tweaking task at hand from a Worklet Factory, which has access to a categorized semantic catalogue of junction classes and instantiates them on its behalf. Interfaces exposed by junctions in the catalogue must be matched to the kind of

capabilities that are necessary for the task and to descriptions of the target components subject to tweaking.

Once a Worklet gets to a target component, the interaction between the junction(s) it carries and that component is mediated by a *host adaptor*, which semantically resolves any impedance mismatch between the interface of a junction and that of the component (see Figure 2). The purpose of the host adaptor is to provide each worklet junction with a consistent abstract interface to a variety of component types, including COTS or legacy components, that can be subject to forms of software tweaking that are *semantically* equivalent from the worklet junction's perspective.

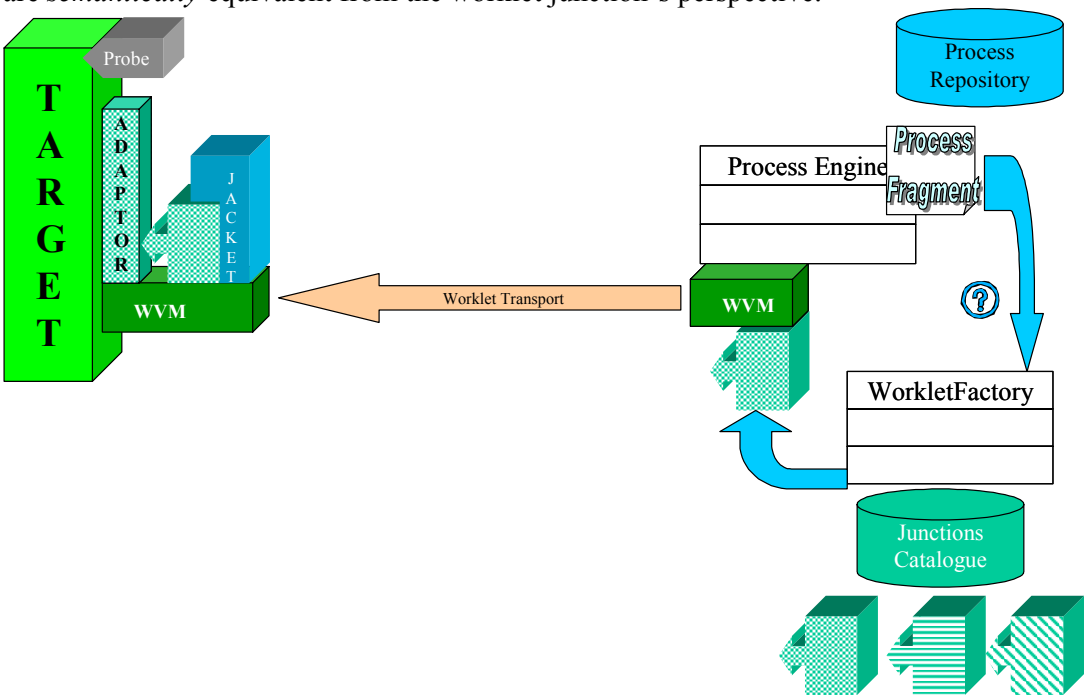

Figure 2: Selecting junctions and shipping Worklets.

The transport services, as well as the execution context and the interpretation of jacket scripts, are provided by *Worklet Virtual Machines* (WVMs) located at each target component intended to host worklets. For each tweaking task, the process engine typically schedules the shipping of multiple Worklets. Each Worklet may traverse various hosts in its route, installing junctions in the corresponding WVMs. Execution of the procedural code of the junctions is (optionally) governed by jackets and carries out adaptation of the target component through the programmatic interface provided by the adaptor. Junctions' data structures may be modified as a result of those operations. At the end of its route, the Worklet may go back to its origin, for any reporting and housekeeping needs, which are performed by a specialized *origin junction*.

Each stop on a Worklet's route represents thence the finest unit of process granularity, and the traversal of the whole route can be seen in itself as a micro-workflow; in turn, that is only a single schedulable step in the context of a multi-Worklet software tweaking process fragment.

3.3 Process-based Tweaking: a Practical Example

To solidify our technical discussion, we now consider the case of a mission-critical component-based application, which relies on and integrates a number of external third-party data sources to provide its services.

Of course, modifications to one or more data sources may significantly disrupt our application. Imagine that the provider of a data source has decided to extend its reachability, to be able to service greater data volumes to more clients. The primary source has changed its network location, while the original location is still in use, but as a secondary mirror with somewhat inferior service level. Furthermore, during the upgrade the provider has partially modified the data definition schema, to accommodate a new wealth of information for its new clients. However, the data source provider has devised and negotiated with its clients a mechanism to handle this kind of upgrading situation, and a converter (e.g., between the old and new XML Schema, assuming XML-codified data) is made available for download.

This scenario can be resolved on-the-fly by a community of Worklets with appropriate capabilities that execute a well-planned software tweaking process. Our mission-critical application is faced with a composite situation: degradation of service from the original data source site, coupled with partial or total inability to make proper use of that data. Those two conditions can be recognized at different levels: respectively, analysis of network traffic from/to the data source, and alarms raised by the parsing/processing application component that "wraps" the data source. Notice that there is a process granularity issue here: those conditions can either each trigger an autonomous tweaking reaction, aiming at overcoming that individual situation, or may be recognized (and responded to) together as the signature of a major data source upgrade (perhaps on the basis of accumulated experience and incremental process improvement). In the former case, two process fragments will be enacted autonomously, which will variously overlap and interact. In the latter case, the overall tweaking plan will integrate both fragments and will schedule them in order, as a single complex tweaking operation. Below, we take this option for simplicity sake.

At first, bots-like Worklets can be dispatched in parallel to retrieve the network location of the new primary data source site and the converter that allows migration to the new data interface. The tweaking process can then proceed by sending out a Worklet with the converter to the component in charge of wrapping the data source, which will execute it in order to be able to correctly process information again. Once this adaptation is performed and proves correct, another Worklet can be dispatched to the same component, in order to reconfigure its connection to the data source to point to the new primary site. In addition, if the communication performance still remains below the service level that must be assured, another Worklet can instantiate a new component, i.e., a load-balancing proxy for the two mirrors, perhaps with on-board cache; in that case, the connection needs to be again reconfigured to take advantage of the proxy.

3.4 Levels of application of software tweaking

Software tweaking can be carried out at various stages in the life of a target application, from installation (e.g., remedies in case the installation script fails because a component executable is missing), to deployment (e.g., identify alternative hosts for components that fail to deploy on the default host), operation (e.g. maintain

critical quality parameters in a desired range), and upgrade (e.g. substitute components on-the-fly with new versions).

Notice also that, in KX, software tweaking can happen not only on the target system but also on the meta-architecture itself, thus enabling flexible monitoring and control modalities that adapt KX functioning parameters to changes in the target. We have identified three major contexts in KX in which software tweaking applies:

- Tweaking of the target system: this is essential to implement the actuating part of the feedback/feed forward loop of KX;
- Tweaking of probes and the filtering and dispatching bus: this allows to vary the kind of system information that is sought and exploited by KX, in response to the dynamics of the target system operation;
- Tweaking of the notification and gauging facilities: this allows to vary the amount and type of information that is reported about the target system, as well as the ways in which it is made available to the decision support components.

4 Related Work

Adaptation of software can be carried out at many degrees: at one extreme, *macro-adaptation* can be achieved only by extensively re-engineering the system; at the other, *micro-adaptation* consists of fine-tuning of running software. Examples of the latter are often found in software (such as drivers) that manages system resources that may have multiple operation modes, and can be optimized on demand or automatically. A more sophisticated flavor of micro-adaptation is implemented in fault-tolerant software by internal application-specific diagnostic code that triggers changes involving a single or a few components.

Software tweaking operates at an intermediate level - meso-adaptation [8] - and as an external facility. Limited, specialized forms of software tweaking are enabled by technologies that are in everyday use. Examples are plug-ins for the reconfiguration and enhancement of WWW browsers and other Internet applications; or software update utilities that operate with mixed push/pull modalities over the Internet to propose, install or upgrade software packages, and negotiate any installation and configuration issues, typically with user supervision.

We are interested in investigating more general-purpose and complex tweaking scenarios, involving a multiplicity of heterogeneous components and a variety of interwoven adaptation procedures. Similar software adaptation granularity is being pursued by a number of other approaches. We focus here – for the sake of brevity – on process and agent-based efforts and their enabling technologies.

Run-time software tweaking can be seen as a specialized case of distributed systems coordination. Research on coordination languages for distributed and more specifically agent-based software has produced a large number of competing approaches and results [10] [11]. Recently, the use of processes as a means for agent coordination has grabbed the attention of researchers [13]. In fact, process formalisms allow describing coordination explicitly and abstractly at the same time. Moreover, they usually combine declarative and imperative connotations; thus, they are feasible for reasoning about the coordination model, as well as implementing it over the distributed software system. Therefore, a number of proposals to employ process enactment facilities as the coordinating core of a distributed system have been put forward.

Coordinating agent applications is one of the goals of Juliette [4]. Juliette is a peer-to-peer distributed process engine, whose agents carry out an overall coordination plan formally defined in the little-JIL visual process language [15], which follows a top-down hierarchical coordination model. Juliette could be seen as an enabler of our software tweaking approach: it can provide "pure" coordination semantics to the tweaking process – encoded in Little-JIL - and distribute them to its multiple decentralized task processors. The computational mechanics associated to those tweaking tasks must be however distributed separately, for example, via Worklets that would be pulled as needed by the task processors and would actuate tweaking procedures onto the target system components. This way, no adaptation functionality needs to be hardcoded into the target system components, nor into the task processors.

Another system that could be similarly employed is Cougaar [6]. Cougaar is a platform for the creation and management of large-scale agent applications, whose centerpiece is a distributed process planning and execution engine. Cougaar's resident process formalism owes much to the domain of military logistics. The timely delivery of the mobile code actuating software tweaking could be approached in this case as a logistics problem, which would allow exploiting some interesting Cougaar features, such as real-time monitoring of the process execution, evaluation of deviations and alternative plans, and selective re-planning. Furthermore, Cougaar supports a plug-in mechanism for the substitution of the default process formalism with others, and composition of applications via process federation. This could lead to process re-use, since fragments of pre-existing software processes dealing with configuration, deployment, validation, etc., could be composed into the software tweaking process.

Our initial experiments with software tweaking indeed regarded the integration of worklets with Juliette. We used our own mockup of the decentralized process enactment facilities of Juliette[1] to request worklets to a worklet factory, dispatch them to Juliette agendas responsible for task scheduling and execution onto target components, and study interaction of worklets with the Little-JIL process paradigm. Our next experiments intend to investigate the use of Cougaar.

While systems like Cougaar and Juliette can be employed for orchestrating a community of agents, which in turn exert control on the target application, other work uses a process engine as a sort of dynamic middleware that directly regulates the intended behavior of the target system. Such an approach aims at defining and enforcing all inter-component interactions as a process. This can enable software tweaking to a certain degree, as far as intended modifications to the system's behavior can be described, either a priori as alternative process courses, or by evolving the process at enactment time.

For example, the TCCS platform delivered by the C3DS project [33] supports on-the-fly composition and reconfiguration of distributed applications. All the operations made available by the interface of each component are described as process tasks. The TCCS process engine defines and automates a *composed service* by sequencing and scheduling some of those operations. The transactionality of the process engine is exploited for run-time evolution of the process, enabling dynamic re-deployment and reconfiguration of the composed service. However, finer-grained tweaking that affects the internal computational logic of one component, or multiple components in a concerted way, remains inaccessible to this approach. PIE [21] takes a similar stance,

[1] We were unable to obtain the real Juliette from U. Massachusetts at that time due to licensing difficulties, which are currently being resolved.

aiming at supporting and managing *federations* of (mostly) COTS components, which together must provide some complex service. PIE provides a middleware, which adds control facilities on top of a range of communication facilities. The control layer implements any process guidance via handlers that react to and manipulate the communications exchanged by federation components. Handlers can be dynamically plugged in the control layer; hence, a specific software tweaking task can be carried out by plugging in an appropriate handler. The granularity of tweaking is somewhat finer than that of TCCS, since besides influencing inter-component interactions, PIE can also modify on-the-fly the semantics of those interactions.

Finally, we consider the Software Dock [16], which combines process and agents technologies in a way similar to our approach, but limited to the automation of distributed deployment activities. An effort aimed at enabling self-adapting software [17] is now under way, which in part builds upon the experience of the Software Dock. It outlines a variety of agent roles for enacting adaptation processes, a contribution that may be relevant to our work, in order to precisely characterize Worklet types for specific levels of tweaking (see Section 3.4). Furthermore, it recognizes the need for an "abstract" coordination service to orchestrate those agents, which we propose to implement by exploiting process technology.

5 Conclusions

Run-time tweaking of complex distributed software systems is both an opportunity and a challenge. It can benefit development, evolution and operation of software (particularly component-based software) in terms of quality and costs; however, it poses various serious difficulties. It demands considerable formal knowledge about the specifications of the system, and ways to express, reason about and exploit that knowledge to come up with appropriate adaptation procedures. It requires facilities to coordinate the actuation upon system components of those – possibly sophisticated - procedures. It must provide computational mechanisms for interfacing to system components and modifying their functionality on-the-fly.

We propose a software tweaking approach that incorporates facets of essential software processes such as configuration management, deployment, evolution and validation in an integrated software tweaking process. Such a process allows to explicitly codify adaptation procedures to be carried out in response to a variety of conditions.

We have conceived a platform based upon decentralized process and software agent technologies for the support and enactment of software tweaking processes. The process engine serves as the coordinator of tweaking agents, which in turn are the actuators of adaptation procedures upon components of the distributed system subject to tweaking. This approach permits to attack separately the coordination and computational aspects of software tweaking.

The first and second generations of the worklets system have already been released to some external users. We are currently developing the first release of our software tweaking platform, which must integrate our worklets mobile agents with a decentralized process engine. We have carried out experiments with some engines, and we are now implementing advanced features such as the worklet factory and jackets (see Section 3.2).

Finally, we are in the process of applying software tweaking to various targets, as diverse as a multimedia-enabled educational groupware platform, a crisis operation

planning application, and mobile localization telecommunications services. This variety of case studies is likely to provide us with considerable insights on the feasibility of our approach, and on a set of questions that remain open at this stage. Among them: the level of accuracy and detail of the target system specifications necessary to the tweaking process; the most appropriate process paradigms and any extensions to them that may be needed for specific software tweaking processes; architectural constraints on the monitoring and control meta-architecture; efficiency issues; responsiveness and real-time requirements.

References

[1] Gail Kaiser, Adam Stone and Stephen Dossick, *A Mobile Agent Approach to Lightweight Process Workflow*, in Proceedings of the International Process Technology Workshop, Villard de Lans, France, September 1-3 1999.

[2] N. Carriero and L. Gelernter, *Coordination Languages and their Significance*, Communications of the ACM, Vol. 35, no. 2, pages 97-107, February 1992.

[3] George T. Heineman, *Adaptation and Software Architecture*, in Proceedings of the 3rd International Workshop on Software Architecture, November 1998.

[4] A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil and A. Wise, *Logically Central, Physically Distributed Control in a Process Runtime Environment*, Technical Report # UM-CS-1999-065, University of Massachusetts, Computer Science Department, Amherst MA., November 11, 1999.

[5] J. Grundy, M. Apperley, J. Hosking and W. Mugridge, *A Decentralized Architecture for Software Process Modeling and Enactment*, IEEE Internet Computing: Special Issue on Software Engineering via the Internet, 2(5): 53-62, September/October 1998.

[6] Cougaar Open Source Agent Architecture http://www.cougaar.org/index.html

[7] Giuseppe Valetto, *Process-Orchestrated Software: Towards a Workflow Approach to the Coordination of Distributed Systems*, Technical Report # CUCS-016-00, Columbia University, Department of Computer Science, New York, NY, May 2000.

[8] *MesoMorph: Meso-Adaptation of Systems* http://www.cis.gsu.edu/~mmoore/MesoMORPH/

[9] Spencer Rugaber, *A Tool Suite for Evolving Legacy Software*, in Proceedings of the International Conference on Software Maintenance'99, Oxford, England, August 30 - September 3, 1999.

[10] G. A. Papadopolous and F. Arbab, *Coordination Models and Languages*, Advances in Computers, Vol. 48, Academic-Press, 1998.

[11] A. Brogi and J. M. Jacquet, *On the Expressiveness of Coordination Models*, in Proceedings of the 3rd International Conference on Coordination Languages and Models (COORDINATION 99), Amsterdam, The Netherlands, April 26-28, 1999.

[12] R. Balzer and N. Goldman, *Mediating Connectors*, in Proceedings the 19th IEEE international Conference on Distributed Computing Systems Workshop, Austin, Texas, May 31 - June 5, 1999, http://www.isi.edu/software-sciences/wrappers/ieeemediatingconnectors.pdf

[13] M. L. Griss, Q. Chen, L. J. Osterweil, G. A. Bolcer, R. R. Kessler, *Agents and Workflow – An Intimate Connection or Just Friends?*, Panel report in Proceedings of the Technology of Object-Oriented Languages and Systems USA Conference (TOOLS-USA 99), Santa Barbara, USA, July 30- August 3, 1999.

[14] G. Alonso, *OPERA: A design and programming paradigm for heterogeneous distributed applications*, in Proceedings of the International Process Technology Workshop, Villard de Lans - Grenoble (France), September 1-3 1999.

[15] Barbara Staudt Lerner, Leon J. Osterweil, Stanley M. Sutton, Jr., and Alexander Wise, *Programming Process Coordination in Little-JIL*, in Proceedings of the 6th European Workshop on Software Process Technology (EWSPT'98), pp. 127-131, September 1998, Weybridge, UK.

[16] Richard S. Hall, Dennis M. Heimbigner, and Alexander L. Wolf, *A Cooperative Approach to Support Software Deployment Using the Software Dock*, in Proceedings of the International Conference on Software Engineering (ICSE'99), Los Angeles, California, May 1999.

[17] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf, *An*

Architecture-Based Approach to Self-Adaptive Software, IEEE Intelligent Systems, vol. 14, no. 3, May/June 1999, pp. 54-62.

[18] The ABLE Project, http://www.cs.cmu.edu/afs/cs.cmu.edu/project/able/www/index.html

[19] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh, *WebWork: METEOR2's Web-based Workflow Management System*, Journal of Intelligent Information Management Systems, pp. 185-215, 1997.

[20] G. Bolcer and R. Taylor, *Endeavors: a Process System Integration Infrastructure*, in Proceedings of the 4th International Conference on Software Process (ICSP4), Brighton, U.K., December 2-6, 1996.

[21] G. Cugola, P.Y. Cunin, S. Dami, J. Estublier, A. Fuggetta, F. Pacull, M. Riviere, H. Verjus, *Support for Software Federations: The Pie Platform*, in Proceedings of the 7th European Workshop on Software Process Technology EWSPT-7, Kaprun, Austria, February 2000.

[22] G. Cugola, E. Di Nitto, A. Fuggetta, *Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems*, In Proceedings of the 20th International Conference on Software Engineering, Kyoto, Japan, April 1998.

[23] DASADA Program Overview, http://www.if.afrl.af.mil/tech/programs/dasada/program-overview.html

[24] Columbia University Programming Systems Laboratory, DARPA-funded DASADA project, http://www.psl.cs.columbia.edu/dasada/

[25] S. Ceri, E. Di Nitto, A. Discenza, A. Fuggetta, and G. Valetto, *DERPA: A Generic Distributed Event-based Reactive Processing Architecture*, Technical report, CEFRIEL, Milan, Italy, March 1998.

[26] David Garlan, Robert T. Monroe, David Wile, *Acme: An Architecture Description Interchange Language*, in Proceedings of CASCON '97, November 1997.

[27] D. C. Luckham and J. Vera. *An Event-Based Architecture Definition Language*, IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717-734, September 1995.

[28] J. Magee and J. Kramer, *Dynamic Structure in Software Architectures*, in Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4), San Francisco, CA, October 1996.

[29] O. Holder, I. Ben-Shaul and H. Gazit, *Dynamic Layout of Distributed Applications in FarGo*, in Proceedings of the 21st International Conference on Software Engineering (ICSE'99), Los Angeles, CA, USA, May 1999.

[30] C3DS Home Page, http://www.newcastle.research.ec.org/c3ds/index.html

[31] A. Wise, A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil and S. M. Sutton, Jr., *Using Little-JIL to Coordinate Agents in Software Engineering*, in Proceedings of the Automated Software Engineering Conference (ASE 2000), Grenoble, France, September 11-15, 2000.

[32] G. E. Kaiser, S. E. Dossick, W. Jiang, J. J. Yang and S. X. Ye, *WWW-based Collaboration Environments with Distributed Tool Services*, World Wide Web Journal, vol. 1, 1998, pages 3-25, http://www.psl.cs.columbia.edu/ftp/psl/CUCS-003-97.ps.gz

[33] S.K. Shirvastava, L. Bellissard, D. Feliot, M.Herrmann, N. De Palma, S. M. Wheater, *A Workflow and Agent based Platform for Service Provisioning*, C3DS Technical Report # 32, 2000.