Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology

Giuseppe Valetto

Submitted in partial fulfillment of the
Requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2004

ABSTRACT

Orchestrating the Dynamic Adaptation of Distributed Software with Process Technology

Giuseppe Valetto

Software systems are becoming increasingly complex to develop, understand, analyze, validate, deploy, configure, manage and maintain. Much of that complexity is related to ensuring adequate quality levels to services provided by software systems after they are deployed in the field, in particular when those systems are built from and operated as a mix of proprietary and non-proprietary components. That translates to increasing costs and difficulties when trying to operate large-scale distributed software ensembles in a way that continuously guarantees satisfactory levels of service.

A solution can be to exert some form of dynamic adaptation upon running software systems: dynamic adaptation can be defined as a set of automated and coordinated actions that aim at modifying the structure, behavior and performance of a target software system, at run time and without service interruption, typically in response to the occurrence of some condition(s). To achieve dynamic adaptation upon a given target software system, a set of capabilities, including monitoring, diagnostics, decision, actuation and coordination, must be put in place.

This research addresses the automation of decision and coordination in the context of an end-to-end and externalized approach to dynamic adaptation, which allows to address as its targets legacy and component-based systems, as well as new systems developed from scratch. In this approach, adaptation provisions are superimposed by

a separate software platform, which operates from the outside of and orthogonally to the target application as a whole; furthermore, a single adaptation possibly spans concerted interventions on a multiplicity of target components. To properly orchestrate those interventions, decentralized process technology is employed for describing, activating and coordinating the work of a cohort of software actuators, towards the intended end-to-end dynamic adaptation.

The approach outlined above, has been implemented in a prototype, code-named Workflakes, within the Kinesthetics eXtreme project investigating externalized dynamic adaptation, carried out by the Programming Systems Laboratory of Columbia University, and has been employed in a set of diverse case studies. This dissertation discusses and evaluates the concept of process-based orchestration of dynamic adaptation and the Workflakes prototype on the basis of the results of those case studies.

Table of Contents

Table of Figures

Acknowledgements

I would like to thank many PSL members and alumni, in particular Dan Phung for his outstanding work on AI^{2}TV and for his reliability, kindness and availability, Matias Pelenur for his work on the Little-JIL interpreter, Gaurav Kc for early discussions on Workflakes and Worklets, Phil Gross, Janak Parekh and Suhit Gupta for teaming up at various junctures (I will remember those transatlantic test and demo sessions for a long time, and I am afraid they will, too).

I would also like to thank Nathan Combs of BBN for his assistance with Cougaar and many fruitful discussions. At University of Massachusetts at Amherst, many thanks to Lee Osterweil for his encouragement and insights, and Sandy Wise for his support with Little-JIL. Thanks also to my colleagues at Telecom Italia Lab, in particular Luigi Licciardi and Mario Costamagna for believing in my work, and Elio Paschetta and Matteo De Michelis for their precious help.

During the course of this research, the Programming Systems Laboratory has been funded in part by DARPA, monitored by AFRL, F30602-00-2-0611 (DARPA Order K503) and F30602-97-2-0022 (DARPA Order E101); ONR in cooperation with NRL N000140110441; National Science Foundation grants CCR-0203876, EIA-0202063, EIA-0071954, and CCR-9970790; and by Microsoft Research, IBM and NEC Computers. The work at Telecom Italia Lab was funded in part by EURESCOM (project P-1108).

The views and conclusions contained in this document are those of the author, and have not been endorsed by and should not be interpreted as representing the official policies, either expressed or implied, of DARPA, AFRL, the Air Force, ONR, NRL,

Dedication

To Roberta, whose unconditional love makes everything possible.

To Pietro and Matteo, who make every day new and precious.

To my parents, for their invaluable support throughout the years.

1 Introduction

Software systems and services pervade our lives at an unprecedented scale, in great part thanks to the popularization of the Internet and the distribution and componentization of a variety of software on top of such a global networking environment. Software applications are becoming increasingly interconnected and interoperable, and provide us with a multitude of value-added services, which can be increasingly devised and offered by composing pre-existing software in new ways.

That trend is likely to continue and become even more explosive in the next few years, with the emergence of ubiquitous, interconnected communication and computing facilities, pervading our living environments and embedded in a variety of devices, as well as of newer distributed computing models that push or transcend the traditional client/server paradigm (such as multi-tiered architectures, Web Services, peer-to-peer computing, and Grid computing).

One downside of that scenario is that software systems that provide the value-added services we are becoming accustomed to are rapidly becoming extremely complex to develop, understand, analyze, validate, deploy, configure, manage and maintain. The reasons are twofold. Firstly, each service is likely to rely on a number of integrated software components, as well as resources (computational power, networking and data), which may be heterogeneous, loosely coupled, and dispersed to various extents. Notice how this integration complexity is not only horizontal, i.e., at the application level, but also vertical, since – in accord with the middleware approach – the final application sits on top of multiple layers of infrastructure software, each of which is designed to abstract the layer below and hide its idiosyncrasies. Moreover, analyzing

and understanding all the inter-component dependencies of complex, heterogeneous distributed software systems and their impact on quality is increasingly difficult, and can become outright impossible if componentization is pushed to its limits. For example, in a componentized scenario of global scale, multiple services could dynamically find, bind to, and invoke remotely deployed components and resources that match their computing needs; thus, they might introduce extremely variable usage and interaction patterns for said components and resources, whose consequences would become hardly predictable and even difficult to replicate within a testing lab.

It is also noticeable that this multifold growth in software complexity occurs at both the development and the execution phase of the software life cycle. Complexities that surface at development time can be at times mitigated by taking advantage of certain insights and best practices in software development, such as component-based software engineering, which aims at promoting re-use, interoperability and standardization, or formal specification and design methods, which can help with describing and reasoning about the various facets of especially complex systems.

Complexities that are related to the post-development phases of the software life cycle regard managing and ensuring proper quality of service to software systems once they operate in field conditions. Those complexities are particularly intensified in a scenario where services are built from a mix of proprietary and third-party components, so that the control of and the knowledge about the overall system and its runtime environment does not belong to a single stakeholder.

It is well-known that it is quite hard to carry out systematized testing during the development phase, in order to properly profile and validate the usage of single components once they are deployed and function on-line in a widely distributed execution environment; it is even harder to come up with tests for the many possible interaction patterns of integratable component sets, which may include some third-party, legacy or COTS elements. That difficulty limits the level of assurance that can be achieved, in particular with respect to the non-functional characteristics of the computing entities under test. As a consequence, critical conditions, errors and failures become manifest only in the field, rather than in the lab, and corrective maintenance has become an intensive, continuous activity that spans the whole product lifetime and accounts these days for the majority of software costs. The abovementioned current trends in large-scale distributed computing are likely to aggravate this situation.

Furthermore, traditional software management practices occurring in the post-deployment phase are quite labor-intensive and in the current state of the art still rely heavily on human analysis and intervention. As such, they are (and will increasingly become) slow and error-prone, as they struggle to cope with the rate of growth in systems' complexity. As a consequence, the reaction to and resolution of faults, mis-configurations, overload, or other common run-time software mishaps typically comports some period of service interruption, or at least significant degradation of the service quality. That translates into further increasing costs and difficulties when trying to operate large-scale distributed software infrastructures and applications that

must continuously abide with certain levels of service, according to their functional and extra-functional requirements.

The need to respond to the complexity challenge outlined above is gaining considerable attention as one of the major problems to be faced in Information Technology today and in the next future, with respect to the engineering of complex software systems. A number of research initiatives that advocate and investigate new methods and tools to cope with it have recently been launched under a variety of denominations, such as on-line validation [56], recovery-oriented computing [121], steering systems [55], dynamic systems [202], autonomic computing [32], etc.

Those initiatives vary in scope and differ in the conceptual and technical approaches they advocate. For instance, on-line validation suggests a vision in which systems are continuously supervised and kept functional and in good shape by some external means. Recovery-oriented computing emphasizes preventing system faults, or overcoming them by keeping or returning systems to their full functionality as before the occurrence of the fault. Steering systems investigate how to develop software whose operation dynamics and parameters can change as a function of its execution. Dynamic systems and autonomic computing are principally concerned with making the software infrastructure and the crucial Information Technology assets of an organization intrinsically and automatically manageable, taking in account not only the technical but possibly also the business-related aspects that impact systems management.

Notwithstanding the differences, the common concept at the basis of all those initiatives is to automatically and transparently handle complexity as soon as it

displays its adverse effects on a system in operation, or even preemptively; the goal is to keep systems running and providing service within their intended functional and extra-functional boundaries at all times and in all conditions – possibly save for the most critical, extreme or unexpected faults. The means is the introduction of computing provisions for the *dynamic adaptation* of single components as well as the overall service. With the term dynamic adaptation, we intend some automated and coordinated set of actions (expressed as computations) aimed at modifying the structure, behavior and/or performance of a target software system, *at run time*, with no service interruption, and minimal service perturbation. Dynamic adaptation typically intervenes in response to the occurrence of some condition (or a complex mix thereof), and has the purpose to ensure the continuous provision of service with acceptable levels of quality. Examples of adaptations may range from tuning functioning parameters within a single component in order to influence its isolated performance, to concerted (re-)configurations of multiple components and connectors, to component instantiation or migration, to architecture-wide interventions, such as on-the-fly (re-)instantiation of the service as a whole.

Dynamic adaptation can be seen both as an on-line extension of software maintenance practices, and as an automation of existing post-deployment management practices. It can address a set of issues that is potentially quite vast: (re-)deployment, leading to automated system and service staging and evolution; dynamic (re-)configuration at different levels of granularity, leading to Quality of Service (QoS) optimizations of various kinds and self-management, such as availability, scalability, performance;

fault recovery and prevention, leading to self-healing; the activation of security countermeasures, leading to protection from attacks; etc.

The autonomic computing initiative categorizes the various features it seeks to develop and promote in autonomic systems as *self-configuration, self-healing, self-optimization and self-protection* [32]. The "self-" prefix indicates the focus of autonomic computing – which is shared by many other similar initiatives - on building new kinds of systems with intrinsic adaptation provisions that are embedded into their implementation. Such an internalized approach, however, may suffer from two major drawbacks: firstly, it promotes an "egotistic" stance, in which each element in a composite system decides upon and effects dynamic adaptation on its own, overlooking any end-to-end perspective that embraces the overall system; moreover, it may prove unrealistic in an IT world in which systems in operation are a mix of new and old software, with new software a possibly minor portion of the whole, and with old software often impervious (unless running prohibitive costs) to the kind of re-hauling needed to make it intrinsically adaptive.

An alternative approach can be called *externalized dynamic adaptation*, which operates from the outside of and orthogonally to target applications, without making any assumptions about the targets' implementation, internal communication and computation mechanisms, source code availability, etc. As such, externalized dynamic adaptation is applicable also to legacy systems. Notice that "legacy" is taken here in a rather broad sense, to encompass not only pre-existing and "ancient" software, but also any third-party components, subsystems and entire, self-standing systems. The targets of the approach can be hence generally characterized as *systems*

of legacy systems, that is, heterogeneous, possibly very large and loosely-coupled ensembles of components of different origin and varying granularity that work together towards providing a given service, which must be made autonomic as a whole. In the remainder, a system of that kind is often referred to as the *target system* of a dynamic adaptation facility.

In order to effectively carry out externalized dynamic adaptation on that kind of target systems, a set of capabilities, or roles, must be present. Among them: *monitoring*, i.e., the ability to provide snapshots of the state of the system and its constituents (architectural components and connectors, or even finer-grained modules), which must be sufficiently detailed to capture and expose enough information about any run-time criticalities to be addressed by the adaptation facilities; *diagnostics*, i.e., the ability to analyze said snapshots and find out whether critical conditions have occurred (or are about to occur) and to point out their cause; *decision*, i.e., the ability to figure out what among multiple possible adaptation strategies is the most suitable for the diagnosed situation; *actuation*, i.e., the ability to summon and effect on demand some (re-)configurations or other controlled modifications onto the running system implementation; and *coordination*, i.e., the ability to carry out actuations that impact multiple components in a concerted fashion, as required by the chosen dynamic adaptation strategy.

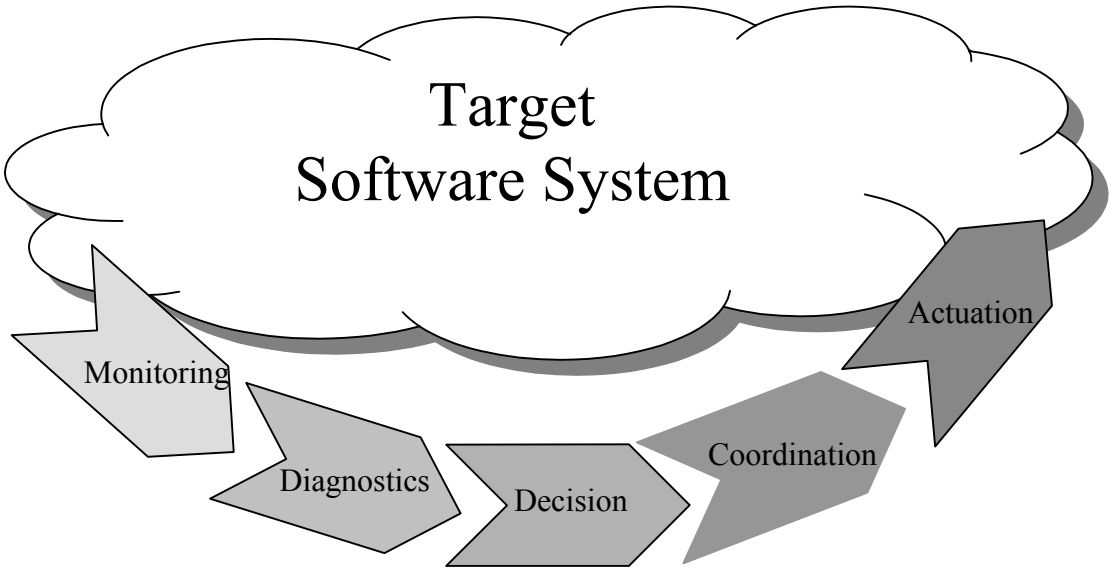

Figure 1: Roles in Dynamic Adaptation of Software.

Figure 1 shows abstractly how those capabilities defining the dynamic adaptation roles can be integrated into an end-to-end closed control loop that enables the superimposition of dynamic adaptation onto the target system from the outside. That control loop may provide a form of feedback (*detect-and-respond*, i.e., the capability to take some action, as a reaction to the occurrence and detection of a target system condition) and also feed-forward (*detect-and-anticipate*, i.e., the capability of take some preventive action, on the basis of the occurrence and detection of events that are anticipatory of a target system condition, and before it actually occurs). Such an end-to-end, externalized control loop can address equally well, as its targets, new systems developed from scratch, legacy systems and systems built by composition, whether or not they natively take in account dynamic adaptation concerns and features.

The research work presented here proposes an approach for dealing with the aforementioned coordination role – and in part with the decision role as well - in the framework of externalized dynamic adaptation. It does not directly address - but

rather assumes - the existence of the monitoring and diagnostic capabilities of dynamic adaptation. Those other two roles are seen as input sources providing data and triggers to a decision and coordination facility which selects proper adaptation policies and oversees their execution. Actuation capabilities are also outside the main conceptual focus of this work, although of course there is a necessarily tight integration between coordination and actuation capabilities at the implementation level, given that actuators are the natural subjects of coordination in the afore mentioned adaptation policies.

The motivation for the focus on coordination comes from the observation that many existing approaches to dynamic adaptation are local, i.e., they provide adaptive provisions for a single computing entity in isolation (e.g., a Web server), or at most for tightly coupled subsystems that are designed and bound to work together (e.g., a cluster of Web servers). Local adaptations intend to achieve and maintain the functionality and performance of those entities continuously optimal (or at least adequate) under variable conditions. Those local optima are irrespective of any larger application context, according to which an adaptive computing entity may become a component in a more complex, often distributed, system. To exert dynamic adaptation internally to a single component, monitoring, diagnostics, decision and actuation capabilities are therefore sufficient, and coordination can be forsaken. Coordination becomes instead critical when the target of dynamic adaptation is a multi-component application, which may or may not include some intrinsically adaptive elements (e.g., a classic three-tiered system made of a front-end clustering facility, multiple Web and application servers participating in the cluster as the mid-tier, and a back-end data

storage). The goal in that case is to maintain the target application as a whole within adequate quality parameters Such an end-to-end adaptation can emerge from finer-grained adaptations such as those carried out on single components, only if they are opportunely orchestrated towards that global goal. Notice how also the connotation of the decision capability changes, since it must also become global, to take in account the "bigger picture" of the overall target application, and may thus need a whole new degree of sophistication and knowledge: per-component decisions leading to local optima may not be adequate anymore, or may even be counter-productive in the light of an end-to-end dynamic adaptation scenario.

To investigate those themes, this research proposes using distributed process technology as a software coordination paradigm that allows to automate the orchestration of a cohort of software actuators (also known as *effectors*), which must work together to establish or maintain the intended configuration, functionality and behavior of the system that is subject to the dynamic adaptation.

Process (often referred to also as "workflow")[1] technology provides suitably sophisticated coordination facilities, since it provides the high-level, abstract and explicit concept of a multi-step process, with each step representing a *task, activity,* or *unit of work,* and with steps connected by control and data flows. Processes can appropriately capture complex, end-to-end adaptation strategies, composed of a number of inter-related actuations that bring some intended side effects on various parts of the target system, and that need to respect complex logic and sequencing dependencies: a process lays out an explicit, global picture of the entire adaptation strategy in terms of more refined activities and fine-grained interventions. Process

[1] In the remainder, the terms process and workflow will be used in an interchangeable manner.

description formalisms are flexible with respect to the type of coordination model enforced, and often offer powerful constructs to handle dynamic dependencies, contingency planning and compensating actions. Furthermore, the state of the art in process technology offers enactment engines that can integrate a variety of actors, support heterogeneous environments and technologies, and address large-scale distribution issues. Therefore, via the enactment of an appropriately codified process, a process engine can orchestrate the execution of actuations by a variety of diverse effectors on dispersed components.

In this work, we present and discuss thoroughly the model, architecture, and implementation of such a process engine – named Workflakes [7] [8]. Workflakes has been developed in the context of a dynamic adaptation platform named Kinesthetics eXtreme (KX) [5] [6], developed by the Programming Systems Laboratory of Columbia University. The KX architecture embodies the externalized control loop shown in Figure 1, and includes – besides Workflakes – complementary facilities for monitoring, diagnostics and actuation [122]. KX remains orthogonal and disjoint from the target systems it is superimposed onto; hence, it promotes the separation of dynamic adaptation from other concerns intrinsic to the target application, can be applied to legacy systems, and can still cooperate with and take advantage of any built-in autonomic features in an end-to-end perspective.

Workflakes is located at the core of KX, that is, at the juncture between the monitoring / diagnostics “front end”, and the actuation “back end” of the control loop provided by the KX platform. Workflakes is implemented as a decentralized process enactment engine based on the open-source Cougaar project [25]. Workflakes

includes a general-purpose application programmatic interface (API) to guide computational units that implement KX effectors, via a number of abstract control and reporting primitives. Furthermore, it fulfills the decision role of dynamic adaptation either internally or by calling external decision-making facilities that capture and evaluate domain-dependent knowledge to elaborate decisions.

In the context of KX, Workflakes has been experimented with and validated in a number of case studies. Those experiments pertain to a variety of application domains, from large-scale information systems, to e-commerce, to personal communication services, to group multimedia provisioning, etc. They tackle different aspects of the dynamic adaptation problem space, from improving QoS, to handling management complexities, to enhancing performance, enforce correct behavior, etc.

Those case studies are presented in detail in this document; their evaluation provides qualitative and quantitative information about the benefits that externalized dynamic adaptation in general, and more specifically the process-based coordination of adaptation, can have on their targets. Those benefits are described in terms of various quality factors that pertain to the goals of each case study and to the operation of the target application and the service it provides, such as reduced efforts and costs, increased efficiency for activities like deployment, management and maintenance, improved service reliability and availability at run time, enforcement of correct system behavior, improved performance and so on. The achieved benefits are also evaluated with respect to the amount of additional development effort, system complexity, and performance overhead introduced by superimposing externalized dynamic adaptation upon the original system.

The experimental work on Workflakes intends to demonstrate two major hypotheses that have originally motivated this research [9]: in the first place, that it is feasible and effective to employ an external infrastructure to retrofit pre-existing software systems and components thereof with dynamic adaptation features; furthermore, that state-of-the-art decentralized process / workflow technology can fulfill the requirements of the coordination role of such an externalized infrastructure, and can exert highly complex forms of orchestration and control on distributed software ensembles, as required for dynamic adaptation. This latter result can be generalized to other application domains that have in common with the realm of dynamic adaptation similarly demanding coordination requirements; for example, how to dynamically determine, initiate and guide some form of "impromptu" cooperation within a group of existing, distributed software entities, in order to satisfy the provision of some service on demand (see for example [123], [124], [125]).

This document addresses the various issues outlined in this Introduction. First of all it provides an exhaustive presentation of the approach: it starts with an overview of the most important conceptual aspects of dynamic adaptation; it continues with the description of a generic externalized architecture for dynamic adaptation [57], as it has emerged from the joint design work carried out by participants in the DARPA DASADA program [56], under which much of this research was developed, and with a discussion of process / workflow formalisms and technology in the context of the major recognized paradigms employed for software coordination; it then analyzes the requirements of coordination related to the domain of dynamic adaptation and how process technology can fulfill them.

The document continues with the presentation of the choices taken in this research, and the model that derives from them, discussing its rationale, advantages and limitations. The description of the design and implementation of the Workflakes process enactment engine for the orchestration of dynamic software adaptation, in compliance with that model, follows.

Coming to the evaluation of the work, the document firstly describes a selected set of case studies in dynamic adaptation involving Workflakes and KX, and then uses their results to assess the major strengths and weaknesses of the approach. Finally, it outlines the contribution of this work in comparison to the state of the art, and forecasts possible paths for future research.

2 Characterization of the Approach

The purpose of this Section is to introduce the major concepts that underlie this research. It begins by providing an overview of dynamic adaptation. Then it describes a reference architecture for a platform that aims at superimposing dynamic adaptation from the outside of a target system. To provide motivation for the usage of processes for the orchestration of dynamic adaptation, it discusses a variety of results in software coordination, and the fit of process technology as a software coordination paradigm. Finally, it outlines the major requirements for a coordination facility for dynamic adaptation and matches them with the characteristics of process technology and of other candidate paradigms, in particular rule-based and agent-based systems.

2.1 A Conceptual Overview of Dynamic Adaptation

Dynamic adaptation can be exerted in a number of ways, which can considerably vary in granularity and scope; moreover, a number of options exist on how their implementation can interrelate with the implementation of the system to be adapted.

With respect to granularity, each single software component may be developed to be adaptive in itself, i.e., for self-diagnosis, self-configuration, self-tuning, self-repair, and so on. An example regards a simple client/server architecture: the request handling component in a server (such as a Web or application server) can be designed to efficiently handle peaks in incoming requests from clients, by activating particular threading and scheduling policies that replace those employed in normal conditions, as a consequence of the detection and for the duration of a peak.

Adaptive features at the granularity of the single component are likely to do a good job if employed in isolation, i.e., to achieve and maintain a "local optimum" with respect to the quality of service provided by that component under a variety of circumstances. For complex systems built from multiple inter-connected components, however, it is not always the case that a combination of local optimizations provides the best adaptation solution across the board. Extending on the example above, if two separate server components, which happen to live on the same host, both react to a request peak for the overall service by increasing their parallelism and spawning additional threads for servicing their respective request queues faster, they might in fact end up in a resource contention situation with respect to the host CPU, thus possibly contributing to deteriorate the quality of service, rather than enhance it. That can happen even if those co-located components belong to different applications, which may be a common case for example in server farms or data centers devoted to the provision of multiple services (in fact, many autonomic computing efforts – in particular industrial ones - are directed towards the automated managements of those data centers, as well as the various applications that are hosted there).

In response to certain conditions, internal adaptive mechanisms are therefore not sufficient and can be, on the contrary, counter-productive; it may be necessary to come up with a global adaptation strategy – at the granularity of the target system as a whole - in which finer-grained, local adaptations assume merely a tactical role and are orchestrated and balanced with respect to one another in accord with an overall strategy. In the example above, an alternative form of dynamic entailing more global re-configurations and optimizations would be deploying other server components and

clustering them together with the one that has trouble servicing incoming requests fast enough.

Notice that there is an interesting parallel here: dynamic adaptation at the target system level builds upon adaptive features made available at the component level, very much like the target system itself is built on top of the various functional features of components; furthermore, like an application logic is necessary to provide the glue that holds the system together and makes it work in the intended way, some overarching logic is similarly necessary, to express and guide the dynamic adaptation of the whole system as a combination of adaptations impacting subsets of components and their connectors in an orderly way.

Another interesting aspect is whether dynamic adaptation features should be embedded within or superimposed upon the target software, i.e., i*nternalized vs. externalized dynamic adaptation*. The former approach assumes significant planning-ahead and effort on the part of the design and development team and is thus particularly effective for new developments, while the latter intends to remain orthogonal to the development of the target system, as well as to its main computation, control and communication facilities, and can be in principle superimposed a posteriori on non-adaptive as well as partially adaptive target systems.

Internalizing adaptive features at the component granularity level can be achieved in a variety of ways, which range from hardwiring fault-tolerance features within the code of a recognizably critical component (perhaps a posteriori, as a result of corrective or perfective maintenance), to more systematic approaches, which can be characterized

as “design for adaptation”. That practice attempts to analyze and address at an early stage the criticalities within a component, and to equip it by design with flexible mechanisms that allow to effect suitable adaptation policies, possibly from a portfolio of options. For instance, aspect-oriented development methodologies [126] can be employed to that end: adaptive features would be then regarded from the start as *software aspects*, that is, concerns that need to remain separated and orthogonal in the development of a given component. They could then be designed and implemented in a modular fashion, with respect to other functional or extra-functional concerns relevant to the same component. That approach is advocated for example by [190]. A number of other approaches are being actively investigated, and design techniques and architectures that explicitly support adaptive concerns gets increasing levels of attention (as demonstrated, for example, by the success of a recent forum like the ICSE Workshop on Software Architectures for Dependable Systems [191] [192]).

Internalizing dynamic adaptation features at coarser granularity levels than a single component, that is, encompassing functional sub-systems or even the target system as a whole, is a more complicated endeavor, and the attention and effort devoted to it at design time is critical. A way to achieve it is embedding adaptive facilities within the very computing infrastructure – the middleware - upon which the distributed target system is built. A number of middleware platforms have been conceived, which offer some set of dynamic adaptation capabilities as a premium for applications built with and operating on top of themselves. An adaptive middleware can either be developed ad hoc, such as Conic [3] and Polylith [1], which are among the earliest middleware prototypes providing support for the dynamic reconfiguration of the architectural

layout and the interconnections of distributed applications; or it can represent an enhancement of some established or standard computing platform with additional features to address certain aspects of dynamic adaptation. For example, 2K / dynamic TAO [2] can reconfigure the real-time TAO ORB [127], which in turn offers features and policies for the optimization of basic CORBA services; BARK [4], instead, can be used for dynamic (re-)deployment of Enterprise Java Beans components.

Internalized solutions, especially if properly accounted for since early design, can extensively cover and keep under control a wide spectrum of dynamic adaptation concerns . However, they also have several limitations, in particular when viewed in the context of large-scale, heterogeneous, component-based systems.

For example, the hardcoding of specific adaptive provisions at the component granularity can many times limit the set of adaptations that can be carried out without re-building the target, whereas they may need to change for various reasons, for instance because of unexpected component usage, or because they may not be flexible enough to harmonize with system-wide adaptation policies. Built-in dynamic adaptation code also tends to make each component more complex, and thus intensify maintenance and evolution difficulties.

Moreover, when internalized adaptation is implemented end-to-end – for example via an adaptive middleware - all service components need to be assembled from the start according to that middleware and the computing model it offers. This introduces a rather strong dependency between the actors and subjects of dynamic adaptation. And, of course, the spectrum and granularity of possible adaptations remains still

restricted by the set of adaptation primitives made available by the specific middleware.

But possibly the most important criticism concerning internalized dynamic adaptation regards third-party composition. When the target system includes legacy, commercial off-the-shelf (COTS) or otherwise third-party components, which may often be the case in large-scale distributed systems, internalized adaptation can be exerted only on those portions of the overall system that are developed either to be intrinsically adaptive, or to comply with the computing model of an underlying adaptive middleware. Third-party elements that might be critical for the overall system may be left out: in such a scenario, achieving a comprehensive and coherent end-to-end dynamic adaptation of the target becomes harder.

In contrast, an externalized dynamic adaptation solution aims at retrofitting components and entire systems with the desired reconfiguration, self-healing, self-management, etc. capabilities, independently from ownership considerations. The externalized approach applies in principle equally to legacy systems or systems built by composition, and to newly developed systems, since its characteristic is to remain orthogonal with respect to the adaptation target.

Although externalized dynamic adaptation is quite general in principle, its feasibility is limited by a couple of critical pre-requisites: the availability of mechanisms to carry out the monitoring and actuation upon the target system. Among the major capabilities needed for dynamic adaptation mentioned in Section 1, monitoring and actuation are crucial since they represent unavoidable points of contact with the adaptation target. While internally adaptive systems and components provide those

capabilities by nature, an externalized dynamic adaptation facility must assume and count on either the availability of accessible monitoring and actuation features (which can be built in the legacy components to be adapted, or offered by their execution environment), or alternatively the possibility to programmatically extend those components to expose enough monitoring and actuation points for its purposes.

The granularity of the monitoring and actuation functionality exposed to an externalized facility is also very important. Dynamic adaptation must be able – in the most general case – to acquire data and intervene at all of the following granularity levels:

- On entities that can be recognized inside a single component; for example, on single parameters or modules that influence some aspect of the component functionality.
- On a component in isolation, for example to instantiate or take down a component.
- On subsystems, i.e., set of interrelated components: for example on the connector managing the interactions of two communicating components.
- On the target system as a whole: for example for the (re-)deployment of the system in a given configuration.

Therefore, an externalized dynamic adaptation platform should strive to have monitoring and actuation facilities that cover all the levels above.

Those pre-requisites are less demanding than they may appear, as a spectrum of options to comply with them is available most of the times. First of all, monitoring and actuation may be offered natively to a certain degree. That happens for example

in a large amount of commercial software products that choose to implement and expose some management facilities, either constructed ad hoc or – increasingly common – in compliance with established frameworks, such as SNMP [128] which defines general-purpose networked entities for passive (monitoring) and active (actuation) management of hardware and software, JMX [129], which provides similar management facilities specifically for Java-based software platforms and applications, WBEM [130], which defines guidelines and technologies for standardized Web-based management of enterprise computing environments, WMI [131], which adopts WBEM for the unified management of Windows environments and applications running upon them, or others. Those frameworks also typically provide means to extend and customize the native basic facilities, to cover particular needs with relative effort and without modifications to the target components.

For software that sits on top of some middleware platform, it is also generally possible to come up with other components devoted to intercepting and manipulating middleware interactions as needed, as shown for example in [10], thus enabling the monitoring and actuation of the architectural connectors of the target system.

Even more commonly, a lot of software offers integration or extension means, in the form of some APIs, which allow the interconnection with other software. That way, wrappers of different kinds can be developed to exert some form of monitoring and actuation, limited to whatever features of the target can be reached through the exposed APIs.

Furthermore, numerous techniques for code instrumentation that augment the target system can be used, either as an alternative or a supplement to the ones outlined

above. Although such lower-level techniques may represent the most powerful kind of tool for exerting *ad hoc* monitoring and actuation on generic software that does not natively provides those capabilities in any other way, they typically demand a significant deal of knowledge of the innards of the target. Instrumentation often assumes source code availability, such as for instance in AIDE [63]. Other techniques enable to work on object code (like for example ProbeMeister [64] for Java byte code, or mediating connectors [65] for WIN32 library wrapping), rather than source code; however, even those techniques may need to be guided by a detailed knowledge of the software to be instrumented, down to the level of how certain invocation chains relate to the behavior and functionality of a given component.

Limited to the monitoring role, increasingly used logging and log inspection facilities [133], as well as tools for the inspection of network traffic (such as Antura [132]), can also provide a wealth of raw data about various facets of application behavior. Finally, operating system-level facilities can be typically exploited as low-level means for coarse-grained monitoring and actuation, with respect to processes and main system resources.

These issues will not be discussed in further detail – except when presenting the implementation of the Workflakes within the KX prototype and the relative case studies - since they do not represent a major focus of this research. However, notice how the availability of any of the approaches outlined above, or of a combination thereof, may satisfy the externalized dynamic adaptation pre-requisites of monitoring and actuation in a large number of cases. In practice, only software components that constitute complete black boxes, use totally proprietary interaction protocols, and

furthermore do not permit any interactions except the ones mandated by their role within the target application are impervious to be monitored or actuated.

While – as discussed above – monitoring and actuation are necessarily tightly coupled with the target system, its features, its technological underpinnings and its implementation, the other major capabilities of dynamic adaptation, i.e., diagnostics, decision and coordination, can remain disjoint from any such consideration.

Figure 2 is a variation of Figure 1 that highlights the dependencies and the data / control flow between the various dynamic adaptation roles and the target system; it graphically suggests how the diagnostics, decision and coordination roles are those that mark most clearly the separation between the system to be adapted and the system exerting the dynamic adaptation. That separation also means that in an externalized dynamic adaptation platform, multiple approaches to achieve diagnostics, decision and coordination can be used, and that the options chosen would strongly characterize that platform. Conversely, the choice of how to fulfill the monitoring and actuation role may often be dictated by the nature and technology of the target system

That vision of target-independent diagnostics, decision, and coordination does not imply that those capabilities can be achieved within a specific dynamic adaptation application independently of the characteristics of the software system to be adapted. In fact, the implementation of the diagnostic, decision and coordination mechanisms must be customized and informed each time with knowledge modeling the problem logic and the environment at hand. That knowledge about the target system must be represented in a format that is understood by all of those roles, and kept in a

repository that is accessible to, but remain independent from them. That way, it is not necessary to embed knowledge about the target system directly in the diagnostics, decision and coordination role, but it is possible to develop generic facilities which are supported by *behavioral models* and corresponding tools. (More details on behavioral models and their importance, in particular in the context of externalized dynamic adaptation, are provided in Section 2.2).

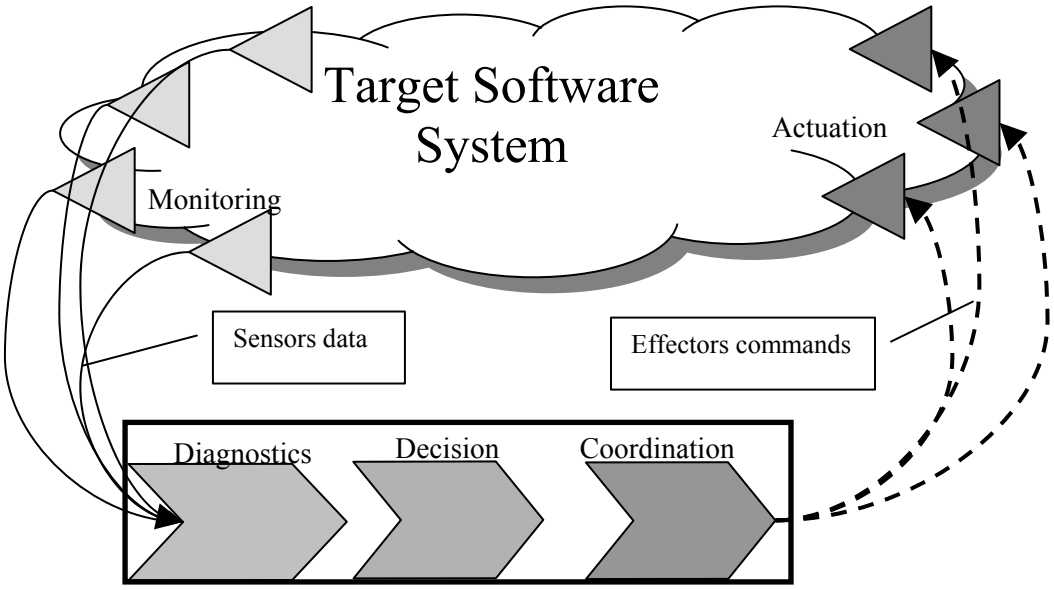

Figure 2: Interactions of dynamic adaptation roles with the target system.

2.2 Reference Architecture

It is useful at this point to introduce a reference model for the architecture of an externalized dynamic adaptation platform, in order to make more concrete the intuition at the basis of the closed control loop vision, as well as the discussion of the various roles participating in it and of their inter-relationships (shown in Figure 2). That reference model is also referred to in the remainder as a *conceptual architecture* for externalized dynamic adaptation, since it provides high-level as well as

operational blueprints, to which the design and the implementation of concrete software platforms should adhere.

One of the major joint undertakings in the DARPA DASADA program [56], under which this research was developed, was to come up with such a conceptual architecture, with an additional requirement regarding its generality. An externalized dynamic adaptation infrastructure needs to be applicable in diverse usage and technological contexts; therefore great attention must be paid to its interoperability with a variety of adaptation targets. Such generality in turn can be achieved via standardization of the interactions and – consequently – the interfaces between the platform components. Standardization enables to choose among possibly different approaches and techniques that can be used to fulfill each of the major dynamic adaptation roles, and to accommodate more easily within the model those that best suit the target system. In the DARPA DASADA program, much work has been devoted to the development of proposals for standard, target- and implementation-independent APIs for *sensors*[2] [58] and *gauges* [59], which are, as we will see, the platform elements fulfilling respectively the monitoring and diagnostic role. The decision and control roles in a dynamic adaptation platform are however less well understood, thus remain further from standardization.

As a result, a reference model was originated, as a common proposal by a consortium of researchers participating in DASADA, as explained in [6] [57].The goal of this model is the full automation of adaptations that must be carried out on the target system. Therefore, it operates at level 4 (out of 5) of the autonomic capability model, as defined in [32], which addresses the resolution of technical aspects relative to

[2] Sometimes also referred to as *probes*.

automation, but does not take in account or integrate in the control loop any business or organization-wide concerns (addressed instead in level 5).

Figure 3 depicts the resulting conceptual architecture: the major elements constituting the architecture are identified, while no assumption of any kind about the target system is made.

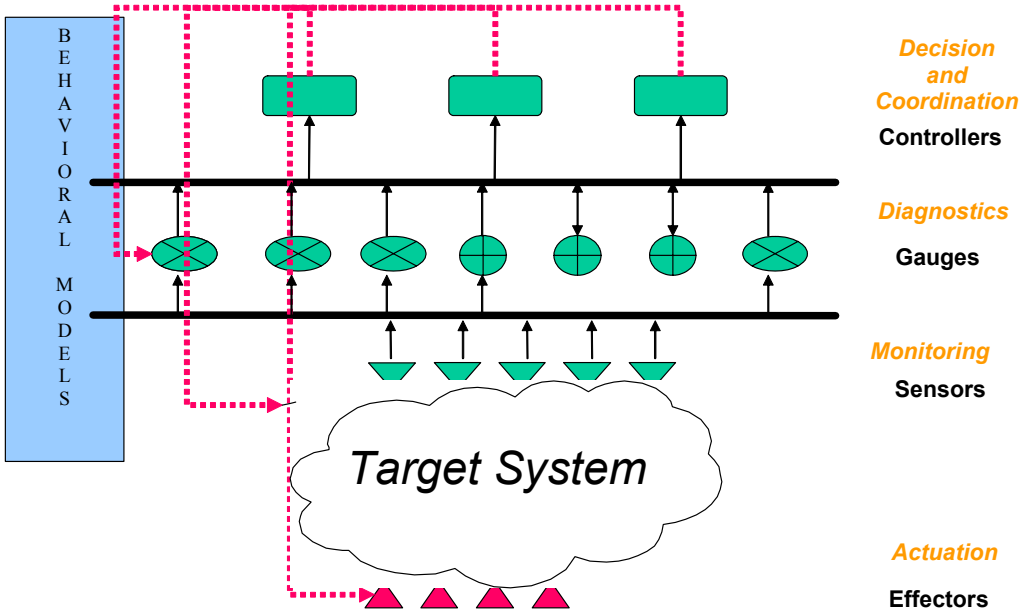

Figure 3: Layout of an externalized platform for dynamic adaptation.

Notice how the externalized dynamic adaptation platform remains physically and logically distinct and separated from its target, although some of the platform elements may be co-located with target components; in particular, sensors and effectors, which respectively fulfill the monitoring and the actuation roles of dynamic adaptation, represent the contact points between the platform and the target system and are most likely co-located. However, since the feedback loop is handled outside of the target application, it is possible to maintain a clear separation between a wealth of reusable, common adaptation mechanisms and the target system specifics.

The conceptual architecture in Figure 3 follows a layered style, which allows to clearly separate – visually but, more importantly, logically - the various roles (a similar separation of concerns is advocated in other dynamic adaptation initiatives, such as [60]). The layered architecture also enables to highlight the interactions among the dynamic adaptation roles and the corresponding platform components. In the Figure, data exchanges are represented by solid arrows and by horizontal lines of communication (or *buses*) among layers; the two buses in the conceptual architecture (the Sensor and Gauge Bus) represent logically distinct communication elements, each devoted to interfacing only certain architectural components through the transport of specific kinds of information. In an implementation, the same physical communication facility could be used for all logical buses. Control interactions are represented instead by dotted arrows, and indicate the path along which the dynamic adaptation interventions occur.

The monitoring layer first gathers information from the running target system, by instrumenting it with sensors. Sensors, which should be minimally invasive, typically generate times series of events containing raw local data, and report via a Sensor Bus to the diagnostic layer. There, information is filtered, aggregated, correlated and evaluated by *gauges*, and findings – that is, abstract semantic events recognized from complex event patterns - are reported to the Gauge Bus. Then the decision and coordination layer analyzes the implications of the gauge findings with respect to the target system functioning and performance, and makes decisions on whether to carry out some dynamic adaptation(s). Adaptation actions would be performed at the actuation layer, under the orchestration of one or more *controllers*. Implementation-

level *effectors* would thus adapt (i.e., reconfigure, repair, tune, etc.) individual components, as well as connectors and other substructures, of the target system.

Notice how according to this design, actuations can in principle occur not only on the target system, but also on elements of the same dynamic adaptation platform, such as gauges and probes. That is intended to provide the dynamic adaptation platform with dynamic *meta-* or *self-adaptation* capabilities (for instance, for on-the-fly re-configuration of sensors or gauges), whose significance will be discussed further in Section 6.3.

Notice also how the view provided in Figure 3 in fact combines the decision and coordination roles within the *controller* components of the platform. Controllers receive and interpret gauge output, perform decision analysis, choose adaptation strategies, and coordinate appropriately the work of effectors. Decision and coordination can be in principle as well as in practice kept separated in the architecture. Anyhow, they are inherently closely inter-related, and combining them offers the potential for continual and incremental *steering* of the adaptation: the controller can immediately consider intermediate outputs originated from effectors' work as well as gauges, which can potentially lead to on-the-fly modifications of the adaptation plan itself.

The conceptual architecture also highlights the importance and pervasiveness of knowledge coming from formal (i.e., machine-readable) behavioral models of the target system: in fact, that knowledge can be employed to drive the target instrumentation with sensors; to contextualize the interpretation of collected information by gauges; to inform decisions taken by controllers; to guide how they

must orchestrate the work of the effectors; and to store the actual repair plans to be executed. That formal knowledge can be captured and made available to the dynamic adaptation architecture via suitable notations and models, which must encompass numerous aspects of the target system, like functional and non-functional properties, protocols, architecture, distribution layout, etc.

One would be hard pressed to indicate a single form of modeling as the most suitable for capturing such a multi-faceted knowledge; it is indeed possible that those various concerns are better captured not by a single, but by multiple complementary representations. Also, dynamic adaptation does not have complete, a priori target analysis and modeling as a pre-requisite. Models can instead be developed piecemeal and selectively, with respect to those target substructures and facets that are relevant to each dynamic adaptation application. Model knowledge could also emerge as a result of the accumulation of monitoring information, upon which inference about the run-time structure, properties and behavior of the target system could be conducted dynamically, like in Software Surveyor [66]. Furthermore, although it is certainly desirable to leverage any pre-existing codified knowledge (deriving, for example, from design artifacts of the target), or to develop exchangeable models in some unified formalism or set thereof, that effort may not be necessary. In particular, since at the current stage of the art no consensus exists on standards for modeling distributed software applications in relation to their dynamic adaptation, even proprietary models constructed and maintained internally by each platform element that necessitates them can serve well. Software modeling connected to dynamic adaptation is the subject of a lot of active research; for example, like in [61] [62], it

can be fruitfully approached from the perspective of Architecture Description Languages (ADLs) and tools, by extending them to cover issues related to capturing, checking and guiding the evolution of run-time software architectures. Further discussion of such modeling issues is in general outside the scope of this work, except what regards enactable models of coordination, which are discussed extensively in Sections 2.3 to 2.5.

The Kinestethics eXtreme (KX) platform developed at the Programming Systems Lab of Columbia University complies with the conceptual architecture discussed above. In particular, in KX, the decentralized Workflakes process enactment engine has been chosen for the controllers. That approach effectively constitutes an application of process / workflow technology as a *software coordination paradigm*. The motivation for that choice can be explained by describing the major requirements of a coordination facility for externalized dynamic adaptation, and how workflow fulfills them. To introduce that discussion, an overview of some other approaches to the coordination of the behavior of software applications must be provided first.

2.3 Perspectives on software coordination

In Computer Science, there have been numerous efforts devoted to studying general-purpose abstractions and formalisms that can be employed to express coordination separately from computation concerns, and that can be applied in multiple application domains. This trend was perhaps initiated by Carriero and Gelernter [96], who proposed the strict separation of concerns between coordination and computation in programming languages. They define coordination as "*the process of building programs by gluing together ensembles of active entities";* a coordination model

takes the role of the glue that binds together the computational activities carried out by the entities in the ensemble, and a coordination language is the linguistic embodiment of a coordination model, offering facilities to express synchronization, communication, creation and termination of the coordinated computations. A seminal example of a pure, general-purpose coordination language is Linda [97] [98]. Linda provides a set of simple, generic but powerful linguistic constructs and architectural abstractions for the coordination of distributed systems and parallel programs. The Linda coordination model is founded on the concept of a tuple space, i.e., a global shared data structure that serves as the only mediator of the interactions among all components of the system. The tuple space model, which owes much to the classic blackboard architecture [99] [100] of many Distributed Artificial Intelligence (DAI) systems, effectively provides an elegant architectural style for distributed systems.

Since the coordination model promoted by Linda can be implemented easily on top of most conventional programming languages [137] and is application domain-neutral, Linda has become a reference point for new coordination models and languages, and the origin of numerous variations, derivations, and specializations in a myriad of Linda-based models and systems, including commercial implementations, such as the JavaSpaces™ [101] by SUN Microsystems.

Those models all share with the original Linda the trait of being data-driven, as opposed to control-driven [102]. In data-driven approaches, the coordination facilities are typically added on top of a "host" computational language: the coordination statements result often intertwined with the computational statements and usually rely on data coming from computation results to implement and regulate communication,

synchronization etc. As a consequence, the coordination model is likely to remain implicit. In control-driven approaches, instead, coordination means remain linguistically separate from computational ones: that forces a clear separation between coordination and computation, and "pure" coordination-based controllers can be explicitly developed.

Many programming languages tackling data- as well as control-driven coordination have been conceived in the last decade, warranting – among other things - an international conference series (see [146]). However, some interesting perspectives to the study and application of coordination in computing have also been contributed by other disciplines, besides programming languages.

For example, across the years, a number of initiatives in the Software Engineering community have been devoted to specifying, modeling and developing computing systems by focusing on describing their interactions, following the recognition of the importance of the concept of architectural *connectors* between components [104]. Those approaches have evolved from Module Interconnection Languages (MILs) [138], to Megaprogramming [140], to Architecture Description Languages [103], and – at the same time – from bottom-up to top-down, from imperative to declarative, and from implementation- to specification-oriented tools. MILs were intended as tools for programming-in-the-large [139]; they operated at the implementation level, and would generate system-specific code for tying together already implemented components.

Megaprogramming languages, such as CLAM [193], have a similar approach, but focus on the interoperation within large meta-systems made of megamodules (that is,

systems-of-systems), and take a more abstract perspective, with a few primitives that describe mainly how to schedule the invocations among megamodules. Both MILs and megaprogramming advocate a largely imperative, compositional and bottom-up approach to the specification of coordination, addressing mainly the development of glue code among computational components, which implements the coordination directives.

ADLs provide formalisms that predicate and reason about software architectures [141] [142], in terms of components, connectors and their instantiated configurations. Their main goal is to provide a high-level, top-down view – a blueprint - of a distributed software system. As observed in [104], connectors are the loci in ADLs for expressing coordination: since the nature of all interactions within an architecture is captured by connectors, different types of connectors can be modeled with enough detail to define and support different coordination models, and, once instantiated in a given system configuration, to determine the coordination aspects regulating the behavior of a distributed software system. ADLs have mainly declarative connotations, and are primarily used as specifications tools, even if they incrementally tend to extend their guidance from design onto the later phases of the SW development process [203]. A further push towards the investigation and the extension of ADLs as languages that enforce the features of the architectural model (including coordination) onto a running implementation of that model in the post-deployment phase is only at the beginning [61] [62].

Another discipline that has been investigating software coordination themes is that of multi-agent systems. Multi-agent systems are rooted in Distributed Artificial

Intelligence (DAI); however, in the context of the affirmation of the Internet as the dominant information as well as computational global infrastructure, agents are increasingly being applied also to mainstream application domains, including the gathering and processing of widely distributed information, data mining, document management, electronic commerce, and others [14].

Therefore, nowadays, numerous state-of-the-art distributed systems are organized and operate as a community of *software agents*: agents are "smart" and "active" components, which may have characteristics such as substantial autonomy, awareness and knowledge of the application domain, some degree of reasoning and decisional power, sometimes code mobility, and more [143]. In such a scenario, in which agent communities may be self-organizing to a degree, the coordination model can be dynamically influenced by the very subjects of coordination. That contrasts with more traditional coordination approaches that adhere to a view in which components are only passive subjects of coordination.

In multi-agent systems, coordination concerns remain well separated from computations by nature: an agent application is largely defined in terms of the cooperation pattern that spans the various agents. Each agent has its functional specificity and a set of computational capabilities, which may be very different from those of other agents: the agents in a community contribute those capabilities in a coordinated way, in order to perform distributed computations and to achieve some overarching result or goal, or offer some service.

The goals of an agent-based application and the ways to pursue them can be expressed in many ways. Generally speaking, the cooperation among the agents

towards their goal is carried out via a series of agent-to-agent interactions, with agents requiring services to each other on the basis of their current knowledge about the other agents' capabilities, their state, the state of the distributed computation, and its "distance" from the intended goal.

Strategies (or *plans*) are the typical means to express the converging behavior of an agent community towards its computational goal. How agents interpret and execute a plan depends on the underlying coordination model and the corresponding Agent Coordination Language (ACL) [17]. The theoretical foundations of most agent coordination languages are generically rooted in speech-act theory [115] [116]; in some of the most prominent ACLs, such as KQML [18], or FIPA [110], that derivation is clearly visible, since they are based upon a set of semantically standardized communication acts. However, the spectrum of coordination paradigms in use in agent-based systems is wide [13]. Depending on the characteristics of the agent infrastructure as well as of the application domain, agent coordination may be implemented with – among others - general-purpose coordination models such as tuple spaces, scripting languages, rule bases, and also decentralized process planning and enactment (as will be discussed in Section 2.4).

Depending on the coordination model of choice, the level of flexibility and dynamism in executing an agent plan may vary from rather inflexible *organizational structuring* [15] (i.e., a coordination plan is defined a priori and superimposed by a master coordinator over the agent community) on one extreme, to fully *dynamic* or *run-time negotiation* [16], (i.e., the plan is continuously evaluated and decided among the

agents throughout execution, according to some self-organizing scheme of the community [13]) at the other extreme.

In run-time negotiation, the coordination scheme is often expressed as a form of declarative knowledge, and the resulting plan towards the goal is said to *emerge* from that know-how, as well as the operating conditions and the input of the agent community. Those plans are very open-ended: two different runs of the same agent community, aiming at solving the same computational problem, are likely to differ even considerably under a full negotiation model. The level of sophistication of the knowledge codification and of the mechanisms employed to make good use of that knowledge greatly varies - of course - with the complexity of the problem at hand: at times it can be captured with some deterministic script loaded in each agent; other times, however, it may be necessary to provide agents with significant semantic and reasoning capabilities, and a lot of autonomy in determining their own course of action.

In the organizational structuring scheme, instead, a rather prescriptive form of plan is assumed, which – instead of emerging bottom-up - is explicated a priori in a top-down fashion and assigned to the responsibility of a master coordinator. The plan is then carried out in a centralized fashion by that coordinator, which orchestrates the operation of a number of peripheral agents. Notice how centralization here is not so much physical or topological, but rather logical and organizational: it is the function of control that the coordinator provides in an organizational structuring scheme that is logically centralized, which means very limited autonomy is left to other the participating agents.

Many agent coordination systems, in practice, employ approaches that lie in between the two extremes described above, and try to variously reconcile bottom-up autonomy and top-down guidance: one possibility is to make organizational structuring hierarchical, through the explicit delegation of portions of the plan to different *manager* agents (see for example [174]); another variant is to have a high-level centralized plan, which enforces top-down guidance to a certain level of detail, but leaves to the autonomy of agents the resolution of the finer-grained parts of the plan, which emerge from a network of agent-to-agent interactions.

Another paradigm that can be fruitfully employed to implement coordination models is that of rule-based programming. Rule-based approaches have been extensively used in Artificial Intelligence expert systems, with prominent examples such as OPS5 [105] and CLIPS [106], principally to provide automated reasoning and decision support. From there, they have extended their reach to the implementation of flexible decision systems, widely used in application domains such as Telecommunications Management Networks [147], data management [144], and others.

Among rule programming paradigms, one of the most widely used is the *Event-Action* paradigm. According to it, rules are composed of a left-hand side (the Event), which is a declarative description of a pattern that defines some situation of interest, and a right-hand side (the Action), which is an imperative program to be performed when that situation occurs. In its basic form, the Event-Action paradigm is particularly suited to specify reactive behaviors in a system, and it is practically stateless. However, numerous variations that introduce and exploit a notion of state in the rule-based system exist, through the definition of so-called Event-Condition-

Action (ECA) rules [145]. Conditions are predicates over the state of the system – as well as the content of the received event: the state must be somehow available and known to components that receive events and must execute actions: only if the condition attached to a matching rule is verified, the corresponding action gets fired. A further enhancement is to add Alternative Actions, moving from the ECA to the ECAA rules paradigm [194], which allows defining actions that are fired in case the condition of a matching rule is NOT satisfied.

Rule-based programming with paradigms such as ECA can be used to express coordination: a rule execution engine managing a set of rules (a *rule base*) can direct the work of a score of computational subjects, for example distributed objects [107], by enforcing the execution of actions by those subjects whenever certain situations occur. While rules in their basic form are eminently reactive, they can also provide forms of proactive coordination, when the actions in the right-hand sides of rules bring side effects also on the internal state of the rule system, and when they are enriched with mechanisms for backward- and forward-chaining, such as pre-and post conditions, also known sometimes as *guards*.

In rule-based systems, the overall logic is very fragmented and is defined bottom up. The overall pattern of coordination remains thus largely implicit: it can be derived only by evaluating how rules can be chained to one another, that is, how imperative right-hand sides of some rules can bring side effects that match the declarative left-hand side of other rules. As the rule base grows in size, that task becomes increasingly difficult. Correspondingly, it may also become hard and counter-intuitive

to translate a top-down view of a complex coordination plan in terms of a set of rules that implement that plan.

2.4 Employing processes for software coordination

Workflow technology aims at the support of complex collaborative *processes*, composed of activities, in which the synchronization and coordination of the activities and the actors having a part in them (i.e., the stakeholders of the process) is an essential characteristic.

Traditionally, workflow technology provides paradigms, techniques and tools that support, guide and automate the management of business practices. Among the common domains of workflow applications, there are: clerical work, administrative procedures, commercial transactions (e.g., business-to-business transactions), document management, product development (e.g., software development), etc. From those examples, it can be seen how traditional workflow applications see a central role for humans, whose work is guided by the process model, and facilitated and automated via a set of computer tools that get integrated into the process: typical goals are to increase productivity and ensure consistent levels of quality to human-intensive practice that can benefit from automation as well as the organized use of computerized tools.

Workflow technology is based on the concept of an explicit *process model* that describes process to be followed, and on facilities (collectively termed the *workflow* or *process enactment engine*) for supporting, guiding and automating the collaborative work of stakeholders according to that model [134].

Activities, also referred to as *tasks*, or *steps*, among other denominations, are usually at the basis of workflow modeling: activities allow for process construction, reasoning, and composition. A model includes multiple activities, which are linked together by a set of explicit dependencies, such as temporal and causal relationships, constrained transitioning, synchronization, conditional execution, and more. Those dependencies define how the process flows, in terms of data as well as control (i.e., coordination among activities): at any time during the enactment of a process, a number of activities can be taking place concurrently, provided that their dependencies as defined in the model are satisfied. Activities can be simply a synchronization point for the data and control flow, but more often they represent actual units of work, which need to be carried out for the process to proceed. The execution of the work associated to an activity can be thought of as its side effect outside the realm of the model and within the "real world", that is, the environment in which the process unfolds and upon which it predicates. Depending on the application at hand, a side effect can for instance be a computation by an helper application, the invocation of an external tool, the allocation and use of resources, the assignment of duties to stakeholders, the initiation or conclusion of some transaction, the production of a document, the filling of an order, and so on.

As implied by the examples above, carrying out the work of an activity can require the acquisition, use and manipulation of a combination of *artifacts* that must be indicated in the activity definition, such as input and output data, tools and resources. The work of an activity is also typically associated to some *actor,* which may have a specific *role* or responsibility in the process.

For instance, in a software development process, an activity can represent the building of a new version of a software product. For such a task, the input artifacts would be the source files and the output artifacts would be the binary files produced by the build; the tools could be a compiler and a linker, the resources could be a make script and the source code repository, and the actor in charge of the task would have the role of a software developer.

In Figure 4, the representation of a single activity in the graphic IDEF0 formalism [197] is shown for illustration. It includes the definition of input and output artifacts manipulated by the activity, the resources the activity needs to engage to carry out the corresponding unit of work, and control stimuli that can originate from other activities or external entities.

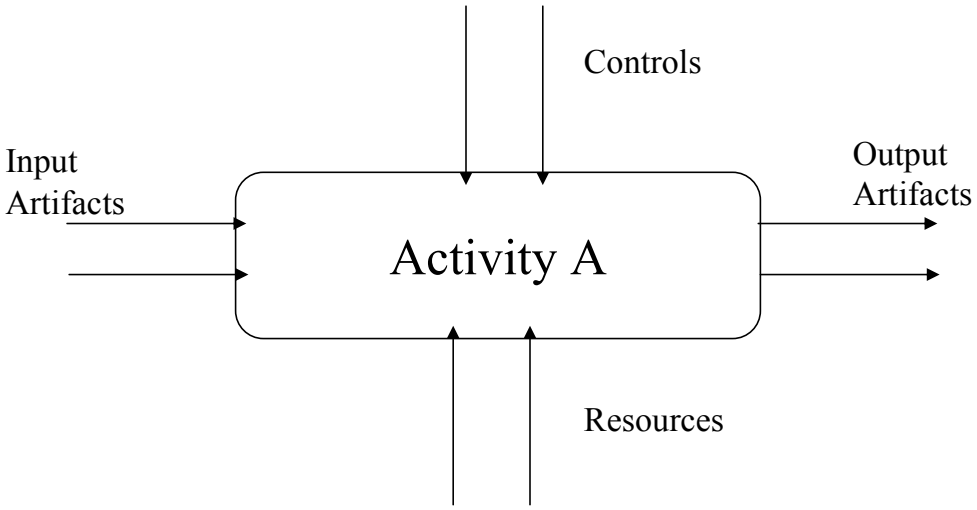

Figure 4: Representation of a generic process activity.

In Figure 5, an illustrative example of a workflow is also shown, with sequential and parallel dependencies between activities that are established directly from the data flow. For the sake of simplicity, control flow constructs are omitted in this example.

By modeling processes along the lines described above, workflow technology is able to describe complex, collaborative work practices in the form of explicit, top-down plans that break down the overall work into a multiplicity of finer-grained steps and a network of inter-dependencies among steps.

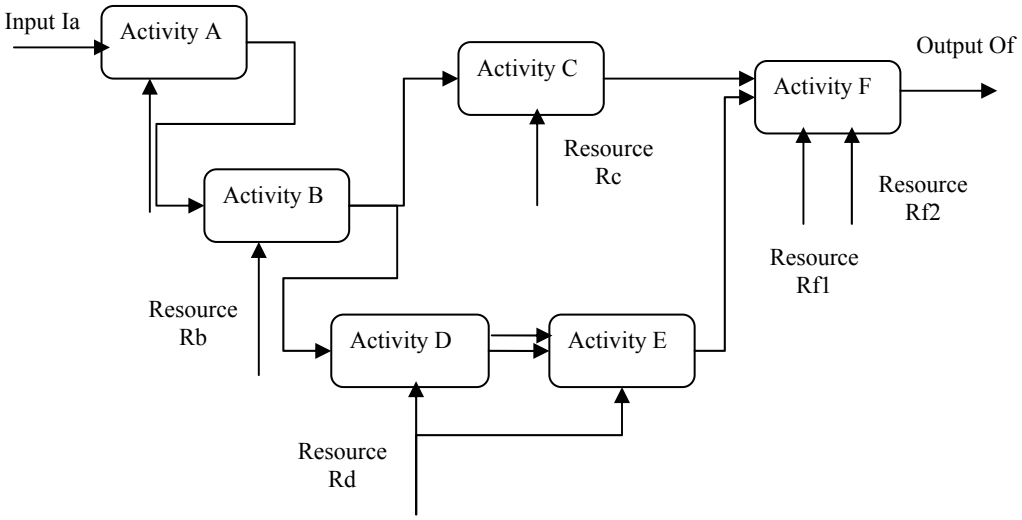

Figure 5: Example of workflow specification.

The workflow paradigm has along the years reached a significant level of maturity. For example, the Workflow Management Coalition (WfMC, see http://www.wfmc.org/) was established in 1993 to establish interoperability among then emerging workflow formalisms and software products; today it counts more than 300 member organizations, including most developers of commercial workflow systems and major IT product developers in general. Along the years, the workflow community has delivered a number of results, several of which relate to coordination languages and models. For example, a number of workflow specification formalisms have been conceived (some of the latest examples are BPEL4WS [95], XLANG [152] and XPDL [198]): those languages effectively provide high-level coordination

models and allow to express, author and maintain those models. Those formalisms may have both declarative and imperative connotations, and accommodate ways to enact processes both reactively (that is, initiate a workflow on the basis of some external stimulus) and proactively (that is, initiate a workflow on the basis of some specific internal state configuration) . Workflow formalisms, one the one hand, make possible and easy to carry out abstract analysis and reasoning, for discussion and communication of the process among human stakeholders, and, on the other hand, are sufficiently formal and precise (i.e., machine-executable), to provide guidance and enforcement of the coordination model by enactment engines.

From a technological point of view, workflow enactment engines have been evolving in the last decade from *centralized* to *de-centralized* architectures. Centralized enactment engines follows some variant of the classic client/server paradigm, which is in fact effective only for systems running on LANs and having a limited number of relatively clustered users and computer hosts. Decentralized enactment engines, instead, have more dispersed computing architectures (for example, hierarchical or peer-to-peer combinations of *task processors,* which collectively form the enactment engine), coupled with distributed information infrastructures as well as distributed organizational and work structures, which commonly leverage the Internet as a substrate, and its related standards and applications (such as the WWW) as a paradigm. Some early examples of decentralized workflow engines are ProcessWall [135], Oz [21] and Endeavors [27]. Current decentralized industrial products include Biztalk Server [195] by Microsoft and WebSphere MQ Workflow [196] by IBM. Decentralization greatly expands the usability of workflow technology, for example

to cooperative processes involving multiple organizations; it also opens the way to innovative means for the definitions, assignment and execution of processes and fragments thereof among task processors.

In the last few years, in part due to the maturity and the insights achieved in the field, approaches and techniques that have been established for and have become typical of workflow have begun to be applied for other purposes, besides the support and guidance of human-intensive collaborative practices. In particular, a number of problems regarding the orchestration of multi-party interactions of software applications or components may be mapped to the execution of some kind of process. In those problems, workflow takes effectively the role of a software coordination paradigm – similar to the approaches seen in Section 2.3 - which leverages the process model for the specification of coordination. Among workflow formalisms, one that has pioneered the application of process semantics specifically as a coordination paradigm, and has a focus on the coordination of software ensembles, is Little-JIL [136].

Some of the distinguishing characteristics that can be offered by workflow as a software coordination paradigm are the following:

- A process specification provides a top-down view of the coordination model.
- The coordination model adopted is very explicit.
- Being top-down and explicit, process specifications tend to provide coordination in the form of prescriptive guidance (as opposed to open-ended negotiation).

The applicability of workflow concepts to software coordination goes beyond dynamic software adaptation, which is the focus of this work; fields of application are

possible and are being explored. To highlight the general issues and provide motivating support our choice of workflow-based coordination in the framework of the KX project, we hereby discuss two other such applications: how workflow can be employed for the automated composition of value-added software services from pre-existing computing entities; and how it can represent a valid paradigm to coordinate the work of a community of software agents towards a common computational goal, according to the discussion about agents in Section 2.3. Those two domains[3] are disjoint from the domain of dynamic adaptation; however, they both present characteristics that can also be recognized as issues relevant to the coordination of dynamic software adaptation.

Orchestration of software composition

A first important class of problems that seeks to use workflow for software coordination can be characterized as the orchestration of the composition of software systems and services. In the context of software composition, the word "orchestration" indicates the automation, in accord to some application-level logic, of the interactions among multiple computing entities, with the aim to come up with a composite computational entity that provides a new service. With respect to orchestrated software composition, workflow can be employed for the definition and enactment of the dynamics regulating the automated wiring of the different software components or applications that must work together towards the composite service. The modalities of such an orchestrated composition may greatly vary, for instance with respect to the degree of dynamism allowed (e.g., pre-defined vs. on-the-fly

[3] Notice that the choice of those two particular problems in this discussion does not imply that other automated software coordination problems are not suitable to be addressed with workflow techniques.

composition), the level of granularity of the participating software entities (e.g., fine-grained components vs. full-fledged, self-contained applications), the intended lifespan of the newly composed service (e.g., opportunistic composition of a one-shot new service vs. persistent composition of a permanent new service). The kind and the complexity of the coordination that must be exerted varies accordingly.

A major contribution to the trend towards automated software composition originates from the increasing degree of pervasiveness reached by software assets within the enterprise, and by the need to integrate those assets together coherently. That is at the basis of Enterprise Application Integration (EAI) [199]. EAI, as a discipline, has two major, complementary concerns: a technological concern, which addresses the integration of all the information systems and productivity applications in a company onto a common integration substrate; and a business support concern, which addresses the automation of the interactions among the interoperating applications in accord with some logic that reflects the nature and the business model of the enterprise.

Technically, EAI seeks a form of large-scale software composition, in which the software to be composed tends to be rather heterogeneous, self-contained and coarse-grained, and the composition logic is largely derived from the business processes of the enterprise. EAI is often promoted and facilitated by the exposition of a wealth of enterprise functions and the corresponding information systems on a common and uniform computing and communications environment, such as that offered nowadays by corporate Intranets. Upon that basis, EAI platforms are typically founded on some form of middleware that pervades the company and supports long-duration and

complex transactions among participating applications. Furthermore, to capture and automate business processes, leading EAI platforms, such as BEA WebLogic Integration [148], or TIBCO BusinessWorks [149], often include a workflow notation and a corresponding enactment engine

Notice how we are talking here primarily about *application-to-application* workflow, as opposed to the traditional workflow focus of supporting and automating human-centered activities with tools (sometimes defined as *human-to-application* workflow). Application-to-application workflow glues enterprise software together by specifying some scripted control and data flow among them, including application bindings, composition constraints, data transformations, etc. The sophistication of the coordination facilities needed for carrying out the kind of automated composition sought in the EAI application domain is relative, since a lot is pre-defined and rather stable, at least for application integration within a single enterprise: for instance, the applications to be composed are well-known, hence there is no need for on-the-fly component lookup and recruitment, nor to dynamically overcome any unforeseen impedance mismatch between components. The major difficulty lies instead in modeling a potentially complex business process correctly in all its facets, and possibly in reconciling within that process any known idiosyncrasies of the participating applications (such as any process-in-the tool syndrome [37]).

EAI initiatives, these days, also strive to surpass the boundaries of a single enterprise and its Information Technology infrastructure, in order to interconnect multiple enterprises that interact with and service each other, for instance in commercial supply chains. Application-to-application workflow thus extends onto larger-scale

company-to-company workflows, often by means of *federation* [150], that is, the composition of the separate and self-standing workflows of multiple organizations, some parts of which are made accessible as entry or composition points from outside each enterprise.

Of course, beyond the boundaries of a single organization, the technical concerns about the integration of enterprise applications are greatly intensified, since a high level of interoperability between the IT infrastructures of the enterprises involved is necessary. That may still represent a considerable technical hurdle, in particular when trying to compose together largely diverse information systems and tool sets, hardly compatible middleware platforms and computing environments, etc.

Once those "hard" (i.e., technology-based) interoperability issues can be reconciled, EAI – irrespective of scale – can be seen mostly as a "soft" (i.e., logical) interoperability question, which can be posed in terms of correctly expressing and carrying out the interactions among the various enterprise applications that need to work together for the task at hand. That is a coordination problem.

Nowadays, the necessary level of interoperability can be achieved by exploiting - besides basic Internet protocols and services - recent advancements regarding standard and open means for the description, lookup and interaction of heterogeneous components over the Internet (such as the family of protocols and programmatic interfaces commonly known under the collective name of Web Services, initiated by industrial partnerships and now embraced and promoted by the World Wide Web Consortium - W3C [12]).

That trend has recently prompted a number of initiatives in WWW-based EAI that assume the availability of Web Services as the technology of choice for the generic interoperability substrate – within and across enterprises - and propose standard languages and frameworks (*de jure* or *de facto*) for modeling and developing business processes. Among the most noticeable initiatives there are BPEL4WS [95] (sponsored among others by corporations like IBM, BEA and Microsoft, and emerging from earlier efforts, such as WSFL [151] and XLANG [152]), and ebBPSS [153] (promoted by the ebXML consortium).

Although the aforementioned initiatives aim specifically at defining formalisms and techniques that enable to wire together enterprise applications according to business processes, it is easy to observe how their relevance goes beyond the domain of EAI. Since they employ a single, uniform way to indistinctively wrap and invoke as Web Services components of any granularity, from simple function calls, to services, to entire applications, to the entry points of other complex, federated business processes, they address in fact the definition of coordination models for orchestrating generic networked computational units, at least those that can be exposed as Web Services.

It remains to be seen how well BPEL or other solutions proposed in the specific EAI arena can gracefully extend to such a more generic view. The solution for many of the various issues related to orchestrated software composition is still of course very much an open research field, for example in cases that are characterized by a need for particularly flexible composition plans and for on-the-fly recruitment of service components, either in impromptu, one-shot compositions, like in DISCUS [124], or in services that are intended as more permanently available, like in DySCo [158]. Some

broad initiatives for the investigation of those open issues have been launched in the Web Services community: one prominent example is the Web Services Choreography Working Group of the W3C [94].

Outside the WWW-based world of Web Services, moreover, other initiatives exist, which see workflow as the glue of complex distributed applications in other computing contexts, such as for example GridFlow [50] for Grids. It remains to be seen whether all of those efforts can be consolidated, extended and generalized.

However, for the purpose of this discussion, it is important to notice how workflow formalisms and techniques applied to application-to-application integration have gradually achieved a degree of maturity and recognition, which makes them a natural technological choice for the orchestrated composition of software.

Orchestration of agent communities

In Section 2.3, agent-based systems were discussed as a specific context in Computer Science in which the investigation of software coordination paradigms and languages is particularly active and relevant. Workflow-based coordination has been applied also in that context, to define and enact the plan employed by an agent community to reach its computational goal.

Since workflow-based coordination naturally leans towards a form of prescriptive guidance, the orchestration of agent communities with an organizational structuring organization - which calls for a master coordinator – is particularly suitable to be expressed as a workflow, and to be enforced at run time by a process enactment engine.

Technically speaking, a de-centralized enactment engine composed of multiple task processors can be employed: many in the first wave of decentralized process enactment engines that have been extensively researched in the early 90's (such as Adele [20], Oz [21], or Serendipity-II [22]), their current commercial-strength counterparts, such as BPWS4J [23], or TIBCO BPM [24], or undergoing academic and open-source initiatives, such as Cougaar [25], or Juliette [26], are suitable candidates, capable to maintain the logical centralization of coordination, while allowing for a physically distributed implementation of the coordinating entity, which scales together with the distribution of the agent community. One typical distribution scheme is a hierarchical organization of task processors. Sub-processes can be delegated to the various task processors, and all dependencies (such as causality or precedence) between sub-processes are to be resolved by a master task processor, taking care of the higher level of the process specification hierarchy. Each task processor thus takes the role of a delegated master coordinator and oversees a subset of the agents, which are regarded by the workflow as pools of resources of computational nature.

Another approach is to associate in the distribution architecture the task processors to the software agents (that is the approach taken for example in Cougaar [25], which integrates software agents and task processors, and to a large extent in Juliette [26], which tends to co-locate task processors with computational executors of work associated to process steps).

Workflow technology is also suitable for a number of agent coordination schemes that fall in between the two extremes of strict organizational structuring and full run-

time plan negotiation. When a plan for reaching the goal is expressed as an explicit multi-participant process, such process can indicate in a proactive way what work stages must be executed at a given moment, and at the same time handle in a reactive way events and situations (including unexpected ones) that occur in the course of the cooperative work. The distinction between proactive and reactive behavior in a workflow that coordinates software agents is particularly important, since it closely mirrors the other distinction, between guidance and autonomy. In the distinction of responsibilities between the workflow engine and the software agents, proactivity maps to guidance: for example, the workflow engine assigns a certain task to a given software agent, which is put in charge of its execution. Reactivity instead maps to autonomy: for example, a software agent can carry out a certain portion of the plan as a reaction to some event, thus exerting some discretional power.

In a case in which autonomy and discretional capabilities on the part of the agents can be exploited, substantial amounts of complexity in the workflow can be deferred from the design time to the enactment time. In that case, the workflow specification does not need to be excruciatingly prescriptive, and may describe the coordination patterns among agents only at a relatively high level of abstraction. Consequentially, many of the more dynamic aspects of the plan may become variable, in accordance with the degree of discretional autonomy enjoyed by the various agents in the community.

In general, how to reach the most effective trade-off between autonomy and guidance depends on the characteristics of the cooperation capabilities built in the agent framework, as well as on the workflow paradigm and its implementation within the workflow enactment engine of choice. That trade-off constitutes one of the major

design decisions to be solved in order to adequately exploit workflow techniques to coordinate a group of distributed agents.

From the discussion above, it should appear evident how workflow formalisms and techniques are complementary to a significant degree and sometimes even overlapping with software agents, in particular as far as coordination is concerned (see also [28]). By hybridizing the two domains, it becomes possible, on the one hand, to employ software agents to represent certain workflow actors, which can contribute to enhance the functionality of decentralized workflow enactment engines, as shown in [22] and [27]. On the other hand, it becomes feasible to co-opt within a workflow paradigm a substantial part of the mechanisms and information that are used for cooperation in an agent community, so that the workflow can be employed to orchestrate that cooperation. That task – as we discussed – is conceptually quite simple when the agent coordination model of choice is close to the organizational structuring extreme of the spectrum, and becomes instead increasingly more complex, as the coordination model drifts towards the other extreme of full run-time negotiation.

2.5 Characteristics of coordination for dynamic adaptation

Having discussed in Section 2.3 some of the leading software coordination paradigms, and in Section 2.4 how workflow technology can be regarded as a means for software coordination, it is possible now to assess its fit with respect to the orchestration of dynamic software adaptation and the requirements imposed on its coordination role.

Automatically adapting a generic distributed application requires the ability to select and apply a plan that brings about some intended changes to the run-time state of that application. That occurs typically as a reaction to some significant piece of information which serves as a *trigger* for the adaptation. That trigger is typically relayed by the diagnostic role, although adaptation could be also triggered willingly by some stakeholder of either the target system or the dynamic adaptation platform.

Regarding the selection of a certain policy, in the simplest cases the trigger may assert a fact that already carries with it unequivocally defined consequences; other times, a variety of tools - which may or may not need to take in account the current state of the target system – could be exploited for the support of the best decision among multiple alternatives. For instance, for a typical dynamic adaptation task such as the on-the-fly modification of the architectural layout of the target system, formal architectural modeling and constraint analysis, coupled with transformation tools, such as [53] [54], can be effectively exploited. A discussion on how to incorporate generic decision tools in the control loop of externalized dynamic adaptation can be found in Section 3.2, where the architecture and design of Workflakes are described.

When a decision to apply a certain adaptation is taken, a single action will sometimes suffice to fulfill it. That is the simplest example of adaptation, and it is the assumption of a number of systems, such as Falcon [55], which is devoted to the interactive or automated steering of a computer program. When the target is a multi-component application, however, the decision will often have to be mapped onto a multiplicity of fine-grained interventions, impacting various separate elements of the target. In that case, the adaptation needs to be represented as a set of concerted and inter-dependent

activities, and some mechanism is needed to take up the coordination role, thus ensuring that their side effects on the target system (i.e., the actuation of the adaptation) occur in a coherent and consistent way. Those activities may have well-defined causal relationships, and they may be conditional, or dependent on others; besides, during the course of the actuation, certain activities may fail, calling for some form of contingency planning; etc.

In general, the more complex the adaptation and the more sophisticated the actuation it calls for on the target, the more involved and well-concerted the corresponding plan needs to be; that, in turn, obviously imposes a set of requirements on the coordination facility. Those requirements regard a number of aspects:

- the **power** of the coordination constructs made available to specify the adaptation plan;
- their level of **abstraction**, i.e., independence from the peculiarities of the application domain and the execution environment;
- the **explicitness** of the coordination specifications, which must be easy to **reason about**, **maintain**, **evolve** and **reuse;**
- to enable automation, those specifications must be **executable** within a computerized environment, thus they have a significant level of **formality** and semantic **precision**;
- the execution of the specifications must highly **repeatable**, yielding results that can be verifiably consistent over time, since automation naturally calls for the ability to carry out validation and auditing, either at run time or “post mortem”.

Workflow technology is one viable choice as the coordination paradigm for dynamic adaptation since it significantly complies with the above mentioned requirements:

- The concept of a process model provides an explicit and abstract way to express sophisticated patterns of coordination. To date, no universal consensus exists on the set of constructs a process modeling facility should encompass. However, sets of process definition constructs (or *patterns* [154]), powerful enough to produce highly detailed specifications of complex coordination logic are supported to a sufficient extent by the state of the art of process modeling (for example, see an analysis of BPEL4WS [155]). That makes feasible and many times even simple to define multi-party, multi-step dynamic adaptation plans as processes.
- High-level process description languages or formalisms exist, which represent valid vehicles to specify, document and reason about dynamic adaptation processes. In particular, the top-down nature of most process specifications is apt to capture human knowledge about what needs to be done to adapt a system, in terms of the sequence of steps that must be followed.
- Process specifications are easily reusable and maintainable, which enables the evolution of the dynamic adaptation process, together with the changing needs of a dynamic adaptation application, as well as the controlled target system.
- Most process representations are sufficiently formal to be executable within an apt process enactment engine that automates the execution of coordination.
- Process enactment software provides out of the box the means for the repeatable enforcement of the adaptation process.

Furthermore, there are some technical features offered as a commodity by state-of-the-art process technology, which are convenient in the context of a dynamic adaptation platform:

- State-of-the-art decentralized process engines ensure the scalability of the approach, and enable to efficiently pursue the dynamic adaptation of widely distributed software applications.
- Software integration mechanisms are typically offered by process enactment engines, in order to facilitate the interaction of processes with a variety of software tools and resources that can be used to carry out the various activities mandated by the process. Those facilities can be exploited for dynamic adaptation to integrate one or more effector technologies.

Other approaches have the potential to fulfill the coordination role in dynamic software adaptation. It is interesting to compare the level of requirements compliance and support offered by process / workflow technology, with that of those other approaches. In particular alternatives that seem to be popular in the domain of dynamic adaptation are based on rule or agent systems. Some discussion on the principled similarities and differences between process technology and those two approaches is reported below; the analysis of related work in Section 6.4 will expand on these issues, by comparing and contrasting concrete examples of works that use those approaches.

Generally speaking, a considerable amount of overlapping and also significant hints of convergence can be observed among rule-based, agent-based and process-based

technologies. In the context of that convergence, Figure 6 tries to depict the major factors of commonality among those software coordination approaches.

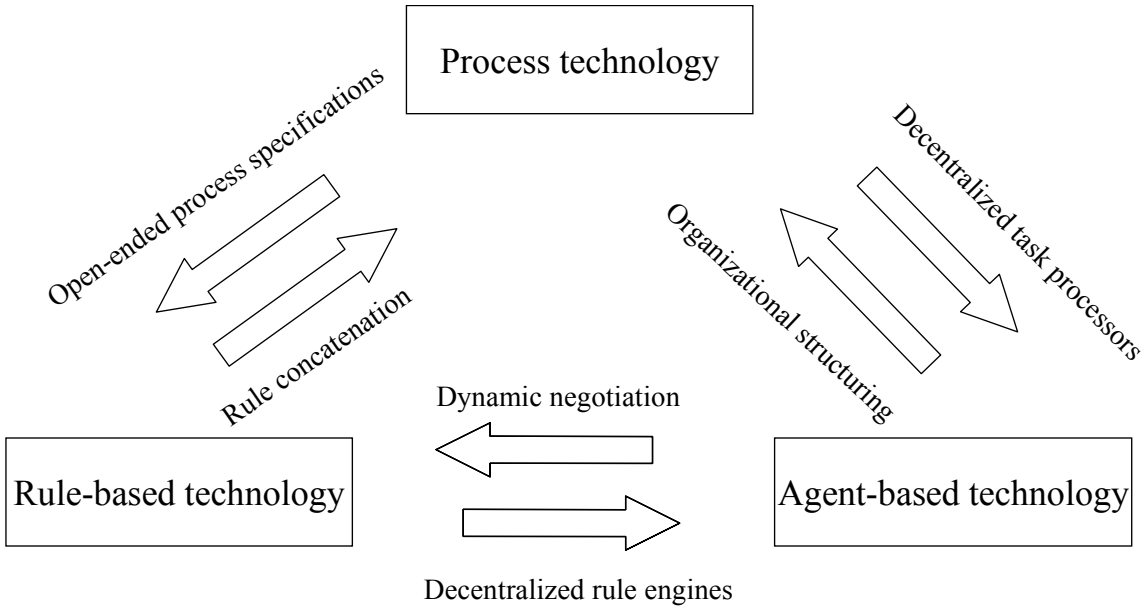

Figure 6: Inter-relationships between rule-, agent- and process-based coordination approaches.

For example, rule-based programming is at the heart of several agent coordination efforts (see for example [109] [111] [112]), in particular when a high degree of autonomy is desired and the coordination model of the agent community leans towards dynamic run time negotiation. The basic idea is that a (portion of a) rule base can be attached to each agent. The autonomous work of the various agents in the community may have – either as a voluntary act or as an implied consequence - side effects that are described in the left-hand side of the rules attached to other agents. It is sufficient to establish within the agent community a communication substrate that propagates the notification of those side effects to come up with decentralized coordination means that can be opportunely programmed depending on the logic of the agent application at hand.

Rule-based programming has also been employed to implement centralized as well as decentralized process enactment engines, such as Marvel [108], Merlin [200] and Oz [21]. A rule-based process specification takes the concept of *process fragmentation* to the extreme: each process fragment is constituted by one single rule: larger process fragments, as well as the overall process, can only emerge bottom-up, via the concatenation of appropriately coded sets of rules.

That approach ensures maximum flexibility and dynamism in constructing open-ended processes; on the other hand, since rule chains are declarative constructs dispersed throughout the rule base, it suffers eminently – as most forms of coordination by means of rules - from implicitness. For that reason, it may be quite hard to turn a coordination plan that can be conceptualized as a step-by-step procedure into a corresponding rule base, in particular when the plan is complex, and the rule base needs to scale up and/or evolve over time. Also maintenance and understanding may become difficult, whenever a rule must be added to a pre-existing process specification, and happens to impact it in some way: for instance, a trigger that is matched multiple times, or process fragments that have overlapping side effects, or interact with each other. However, even if pure rule-based process specifications are perhaps not mainstream in process technology to date, the overlapping between the two techniques and their problem and solution spaces remains evident, and one would be hard pressed to find a single process or workflow engine that does not owe somewhat to rule-based programming. The basic idea of rules remains especially evident in the concept of trigger conditions, which are

included in process specifications to enable the reactive initiation of process enactment.

One important advantage that processes traditionally defined as task flows have on rule-based systems is that they tend to express coordination in a more abstract and explicit way. Because of the implicitness of rule-based approaches, coordination logic expressed that way is usually harder to specify and maintain with respect to a process model. For the same reason, process formalisms are often better-suited for reasoning about and communicating the coordination model. Similarly, also carrying out auditing is normally easier when process-based, as opposed to rule-based, coordination is employed.

With respect to agent-based systems, Section 2.4 discusses how today's decentralized workflow engines can resolve the issue of distributing task processing responsibilities, and how process facilities are suitable and actively being used for agent coordination (see for example [25] [26]), supporting not only strict organizational structuring, but also various degrees of agent autonomy. Notice also that, whereas the full dynamic negotiation capabilities enjoyed by some agent-based systems are hard to achieve by means of process-based coordination, those systems are likely to suffer form an implicitness similar to the one previously discussed for rule-based systems; furthermore, that kind of autonomy may bring about a lack of repeatability, which may hinder auditing and validation.

Based on the considerations above, it can be argued with sufficient confidence that process technology has a number of characteristics that make it a prominent option for the resolution of software coordination challenges inherent in dynamic adaptation.

In Section 3.1, a discussion of how Workflakes tries to take the best advantage of those characteristics can be found.

3 Description of the solution

Workflakes is a process-based facility for externalized dynamic adaptation. Its design has been driven by a certain number early decisions on how a dynamic adaptation process and the corresponding enactment engine should be structured. Together, those decisions make up a model for the representation and the enactment of processes orchestrating the dynamic adaptation of software systems, with particular attention to systems of (legacy) systems, which has been followed and applied in this research. This Section begins by presenting that model, continues with a with a description of the architecture of the Workflakes engine, then discusses its applicability to a spectrum of problems and target systems, and concludes with a critical assessment of the model.

3.1 Model

Various possible alternatives are available when designing a process enactment facility applied to some domain, for example with respect to the kind of process representation to be used, or the distribution architecture of the engine, or the mechanisms for the integration of external tools, and many others.

The discussion that follows aims at describing the decisions taken for the Workflakes process-based orchestration facility. That model emerged by considering, first of all, the coordination role fulfilled by Workflakes within the conceptual architecture described in Section 2.2, as well as its inter-relationships with the other dynamic adaptation roles: in particular, the diagnostic and actuation roles are the ones with the most influence on the orchestration facility. That influence translates into a number of

specificities, in both process specification and process enactment, related to the type and the structure of processes employed to orchestrate the dynamic adaptation of software.

The coordination role is responsible to bridge the more analytical part of the dynamic adaptation control loop, i.e., the monitoring, diagnostic and decision facilities, with its actuation part, in charge to carry out the interventions required for the adaptation. It achieves that goal by expressing control, and providing means to organize and exert it on multiple effectors. Process-based coordination expresses control via the definition of specifications in some process definition formalism, and exerts it via an enactment engine that executes those specifications.

As anticipated in Section 2.5, a dynamic adaptation process follows typically a reactive behavior, following some output from the diagnostic role. The enactment of the process descends from the recognition by the diagnostic role of some significant condition that is occurring in the target system. The relationship between the diagnostic output and the process enactment, although mediated by the decision role, is quite clear: a recognized target condition is an event that may map to a *trigger* and an entry point somewhere in the dynamic adaptation process. The mapping, that is, where the entry point is and what portion of that process (i.e., what *process fragment*) is fired and enacted to orchestrate some interventions on the target system following a given trigger, is a choice under the responsibility of the decision role.

The orchestration of some dynamic adaptation is therefore enacted in a reactive fashion in response to the trigger, and starting from a single *root task,* that is, the entry point associated to that trigger by some automated decision. The root task is

then incrementally expanded into a whole set of steps according to the process specifications: that expansion takes the form of a recursive hierarchy of sub-tasks (a *task decomposition hierarchy*), whose unfolding is in charge in the end of completely handling the target system condition codified within the trigger event.

The original trigger must carry enough input information to allow to initiate the enactment of the process fragment; therefore, both the control and the data flow of the dynamic adaptation process originate from the trigger event, although further information that must be employed during the course of the process may either already be known to the process, or may be produced or acquired as the process fragment unfolds, as a byproduct of the enactment of tasks and the execution of the corresponding work units.

From the discussion about triggering, it follows that a dynamic adaptation process can be represented in purely reactive and compositional terms: it is composed of all the process fragments that respond to and handle some pre-defined triggers, which in turn map back to significant target system conditions. The level of process fragmentation, i.e., the size of the fragments, directly influences the level of open-endedness of the process. Modeling a dynamic adaptation process in that form provides a simple mechanism to close the adaptation loop – specifically the delicate passages between diagnostics and actuation - in a fully automated way.

Such a reactive and compositional approach to the specification of the overall process allows to reach a trade-off, according to which the single process fragments are defined in a top-down, explicit fashion, while the overall process is composed in a bottom-up fashion, from the contributions of the various fragments. The dynamic

adaptation process contains a set of declarative descriptions of *what* conditions are of interest in the target system, each coupled with the specifications of some process sub-structure, which defines a reaction to that condition. All reactions have a clear imperative connotation, since they direct the execution of the adaptation and describe *how* to produce the intended side effects upon the target system via actuation.

In principle, a fully automated and completely reactive process model excludes the possibility of selecting and initiating an adaptation in an interactive way on the part of some agent that is located *outside* the closed control loop of dynamic adaptation. That limitation, however, can be easily overcome, by encompassing in the model of a dynamic adaptation process facility a provision for an external conduit for injecting trigger events into the control loop, either at the monitoring level, or the diagnostic level, or both.

Such a provision is useful for several purposes. When it is used by some human operator, for instance, it enables a degree of *run-time controllability*. The full automation of the coordination of dynamic adaptation brings about a great potential for substantial savings of management resources, prompter response times, and more reliable and consistent interventions. But it also has a downside, with respect to the *controllability* of the adaptation. For instance, the stakeholders of the process and/or the target system might decide to divert the course of the process and guide a certain adaptation towards a different outcome, or to interrupt it altogether. Providing some means for human stakeholders to interact with the process has the important consequence that, while humans may still be completely absent from a dynamic adaptation process, they can also be present, at least for a matter of opportunity, if not

of necessity: they may cover simple and punctual decisional or authorization roles provided for by the process, e.g., to confirm or retract potentially critical or drastic adaptations.

Other entities that can take advantage of a means to issue trigger events can be software systems that are not properly part of the target system or the dynamic adaptation platform, but which might want to proactively initiate some form of adaptation in certain specific cases. Those external systems may be related to other phases of the life cycle of target system, besides on-the-field-operation: a development tool, for example, like a configuration manager, might want to trigger an adaptation process that upgrades the target system, following the release of a new software version (as in Software Dock [36]).

Additional trigger events can be also issued and injected in the same way into the closed control loop of dynamic adaptation as a consequence of side effects included in the process enactment: that can happen either directly (i.e., the side effect by some effector purposely equates to issuing a new trigger), or indirectly (i.e., the effector causes some modifications in the target systems, which are captured by the monitoring role and interpreted as significant new conditions by the diagnostic role, causing the production of a new trigger event). That way, “derivative” proactive behavior can be made part of a process that has primarily a reactive stance.

The interplay between the diagnostic, decision and coordination role, together with the eminently reactive nature of the dynamic adaptation process, resembles the Event-Condition-Action paradigm (see Section 2.3). In fact, the decision role mediates – by properly evaluating conditions predicating over the state of the target system -

between the diagnostic events and the enactment of adaptation provided by the coordination role.

That parallel is valid to some degree. In particular, the significance of the decision role is best appreciated by considering how, for the dynamic adaptation of complex target systems, a simple Event-Action metaphor would be too simplistic and mechanic: it would equate to have completely determined and fixed decisions for each possible occurrence of some diagnostic event, irrespectively of any variability in the operational context of the target system. That is unlikely to be realistic and adequate in field conditions: decision points need many times to incorporate complex and subtle considerations about the current state of the system and its surrounding environment That constitutes one of the major motivations to include an explicit role for automated decision facilities in the dynamic adaptation of software systems.

The parallel between Conditions in ECA rule systems construct and the decision role in dynamic adaptation is furthered by the observation that the complexity of the Condition predicate, as well as the means used to evaluate it, is not constrained in any way in the ECA paradigm: any kind of logic can be accommodated. Such a model remains conceptually agnostic with respect to the decision-making approach and therefore enables to leverage as needed any system that can provide adequate support for the automated decision-making. The decision role in dynamic adaptation is, conceptually, similarly generic and unconstrained.

A difference between the model proposed and ECA rules exists, and is very important. A single process trigger does not simply fire an individual rule, but causes the initiation of a whole sub-process of arbitrary size, duration and complexity, which

may be completely pre-defined in a top-down fashion. In rule-based systems that follow the ECA paradigm, instead, right-hand side actions are atomic and typically short-termed Therefore, to obtain the same effect in a rule-based engine, mechanisms that compute and execute continuation from the rule originally fired are needed, in order to construct chains of rules bottom-up.

While the inter-relationship between the diagnostic and the coordination role defines the modality for initiating the orchestration process, that between the coordination and the actuation role regards the operational semantics to be given to enacted process steps. Let us consider a task decomposition hierarchy as it unfolds from its root: it is made by some inner nodes, i.e., tasks that are further decomposed, and by some leaf tasks. Leaf tasks represent atomic units of coordination that cannot be further decomposed; they do not carry in themselves any additional coordination semantics, so they can naturally represent the units of work that are meant to carry side effects onto the target system. As such, they can be the loci for the operational semantics of dynamic adaptation: for example, the definition of leaf tasks can be used to assign certain effectors to them, their input can be used as the input to the effectors' computations, and any assigned resources as resources to be used by the effectors for their work. Similarly, the results of the effectors' execution can be coupled with the state and the outcome of the corresponding leaf task, and as such, they can be relayed back up the task hierarchy. Leaf tasks can effectively connect the process enactment environment to the "real world", i.e., the computing environment where the target system runs and is being adapted.

For that connection to be effective, there needs to be a tight conceptual as well as operational integration between the enactment of leaf tasks and the effectors they use. The model adopted in this work considers effectors as necessary resources that are explicitly indicated in the definition of leaf tasks; since leaf tasks cannot be enacted and do their work without acquiring and using those resources, effectors become first-class entities in the process specifications. Furthermore, the architecture of the dynamic adaptation process engine includes an *actuation API* that allows leaf tasks – as they are enacted - to interact with effectors (independently of how the latter are implemented) by means of a few generic primitives. That API will be further discussed in Section 3.2.

Therefore, in the adopted model the inner task nodes in the process decomposition hierarchy and their dependencies express the logic according to which leaf tasks are planned and enacted in a concerted way. Leaf task, in turn, oversee the practical units of adaptation work by invoking and controlling their associated effectors through the actuation API.

Another point of interest in processes for dynamic software adaptation is the significance of constructs and techniques for handling exceptional courses of action. In any process it is important to account for exceptions and errors that may occur during the enactment of the process, and to be able to express how the control flow must change as a consequence. That becomes paramount in processes that are fully automated, and whose work aims at forcing modifications onto a complex running piece of software. At any point in the process, the actuation by an effector can fail, or

produce wrong results, or even have undesired effects on the target component; therefore, clear and efficient support for those situations is especially necessary.

Several process languages these days include that kind of support, in the form of exception handling [201], such as for example, Little-JIL [19] and BPEL4WS [95]. Exception handling enables to specify certain branches in the workflow that are executed whenever specific conditions (the exceptions) occur in the internal state maintained by the engine during some phase of process enactment: in a process decomposition hierarchy, exceptions force a jump out of the current sub-process and resume the process enactment on an alternative part of the hierarchy, which behaves as an *exception handler*. At the end of the unfolding of the exception handler, existing systems adopt various options with respect to the enactment of the original sub-process: for example, it can be considered finished, it can resume, it can be re-started, etc, depending on the semantics of the exception mechanism.

It is important to consider what an exception handler sub-process should entail in the case of dynamic adaptation. In many applications of workflow technology side effects tend to be confined to the manipulation of data (internally to the process state store, or in a database, or in a document repository of some sort, etc.). In those cases, traditional transactional mechanisms with commit/rollback capabilities for that data may offer sufficient remedies in the face of some exceptional situation.

In dynamic adaptation, however, when an adaptation of the target system requires multiple interventions to be completed, possibly on a variety of elements, the raising of an exception may occur in the middle of those interventions. That is a situation akin to an error occurring in the middle of some transaction; however, the basic

rollback of an atomic transaction is most of the times insufficient, since some of the side effects that may have been caused by a dynamic adaptation process fragment in the "real world" of the target computing environment (such as the shutdown of a target component) could not be simply rolled back anymore. Instead, the exception handler must be designed to *compensate* the earlier invalid adaptation, and to bring the target system and the process into a state that is consistent and enables further operation of both. Compensation may have the goal to either undo a previous side effect, or to bring forth the system to a new consistent state that is different from before, but stable and acceptable.

Designing compensation sub-processes may be particularly involved in the case of dynamic adaptation. Firstly, the set of possible errors that can arise from the interaction between effectors and target system components can be quite large. Also, some compensations may need to carry out a major re-hauling of the target system at large, even in the face of a local induced fault on a single component, if that component is somehow critical. Finally, when some adaptation requires multiple steps to be completed to bring the target system into some desired state, and some step fails in such a way that the multi-step adaptation cannot go further, the exception handling mechanism may need to be able to compensate also previous steps, even if they were completed successfully.

Exception handlers included in the specification of a process – by their nature – typically are meant to deal with *internal contingencies*. Internal contingencies are those whose possibility to occur in the target system as it is adapted according to plan is known, and which should be explicitly provided for in the design of the adaptation

process. A dynamic adaptation system is also particularly exposed to *external contingencies*. Those can arise because of the level of uncertainty inherent to software execution within a distributed computing environment: glitches, faults or other problems can always occur on any of the software entities involved while the process is in execution. External contingencies may correspond either to unforeseeable target system states, or to faults within any part of the dynamic adaptation loop itself, e.g., a communication failure between effectors and the target system components to be adapted. They can occur either as an unforeseen consequence of the adaptation process, or because of some independent circumstances. To ensure robustness against external - in addition to internal - contingencies, the dynamic adaptation process should provide some generic exception handling branches, representing default courses of action to be taken when "all else fails" and the adaptation process needs to reset to some sort of a "safe state".

All the issues discussed so far as characteristics of dynamic adaptation processes can be succinctly summarized as follows:

- the process is fully automated and reactive, composed of multiple process fragments;
- a fragment is a pre-defined sub-process, fired by a corresponding trigger;
- a process fragment unfolds from a root task as a task decomposition hierarchy;
- a means to interact with the process from the outside of the control loop is provided in the form of a conduit for issuing process triggers;
- inner nodes in the task hierarchy are coordination constructs; leaf nodes represent actual units of work;

- interactions with effectors occur during the course of leaf tasks;
- process includes exception handling for internal as well as external contingencies;
- exception handler sub-processes must encompass forms of compensation.

These characteristics have guided the architectural design of the Workflakes engine, which is discussed below.

3.2 Architecture

To introduce the architecture of the Workflakes process-based coordination facility it may be useful to first briefly contextualize it within the "bigger picture" of the KX platform as a whole (for an exhaustive presentation of the overall KX structure and implementation, refer to [122]).

KX covers the entire reference architecture shown in Figure 3 end-to-end. To implement its various layers, KX uses sensors, gauges, controllers and effectors components, which are physically distributed.

In KX, sensors, gauges and controllers communicate solely via publish/subscribe event notification, using content-based asynchronous messaging middleware. Notice that the Sensor and Gauge buses of the conceptual architecture are unified in KX into a single event notification facility: initially, we chose Siena (Scalable Internet Event Notification Architecture) [156], but later added support for alternatives (e.g., Elvin [157]).

Figure 7 shows how KX building blocks are linked to each other via the event notification middleware, at a high level of abstraction, and disregarding distribution aspects[4]. Besides Workflakes, the other major elements are:

- The *Event Packager*, which acts as an event translation service that pre-processes incoming sensor data, since the various sensor technologies do not necessarily output a unified event format that can be consumed by our gauges. The Event Packager also timestamps sensor events according to a globally synchronized clock, and acts as a "flight recorder" to persistently log in a database the incoming events from the sensors, for later replay or data mining.
- The *Event Distiller*, which is the main gauge component. It performs sophisticated, possibly cross-stream temporal event pattern analysis and correlation across continuous data streams from multiple sensors, to capture and diagnose target system conditions of interest for the dynamic adaptation application at hand. The Event Distiller is dynamically configured with correlation rules defining the event patterns of interest: new rules can be added and previous rules can be replaced or removed on the fly.

The Figure also shows an interaction between Workflakes and effectors that occurs outside the bus: for the rather tightly coupled interactions between controllers and the effectors they coordinate, KX advocates point-to-point communication for those interactions, in the form best suited with respect to the technological underpinnings of the effectors employed.

[4] When KX is instantiated on the field, multiple Event Packagers, Event Distillers and Workflakes task processors can be deployed and configured as needed.

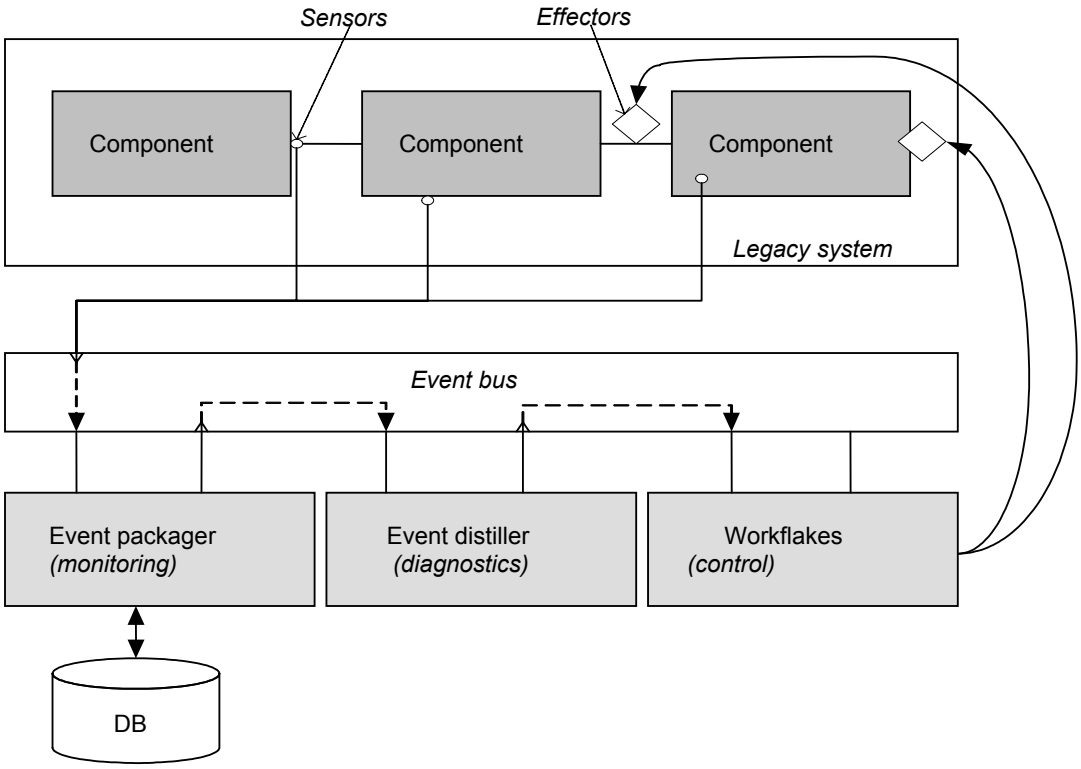

Figure 7: KX architecture.

By leveraging event notification middleware, KX components can be easily rearranged, with multiple instances of KX Event Processors, Event Distillers and Workflakes controllers introduced as needed to address scalability requirements. Furthermore, the actual components that implement sensors, gauges and controllers remain well separated, providing KX with enhanced flexibility: depending on the problem domain, a different set of components may be integrated, and some may be replaced with different, problem-specific alternatives. For example, KX does not formally embrace and include any particular technology for monitoring, since sensors are necessarily highly target-system specific, and thus can vary widely. Among the technologies that have been integrated in KX trials and applications by bridging them to the event-based Sensor Bus there are AIDE [63], library wrappers [65], JMX-based

monitors [129], as well as home-brewed, target-specific sensors directly attached to the event middleware.

The strategy of keeping strongly detached the components that fulfill the various dynamic adaptation roles, and consequently also their respective concerns, applies of course also to controllers, and hence – within KX – to Workflakes. Thanks to that strategy, besides the conceptual inter-relationships already highlighted in Section 3.1, which originate form the conceptual architecture, there are no hard dependencies imposed upon the design of Workfalkes by any design or implementation decisions peculiar to the KX reification of that reference architecture. In fact, in some case studies, as reported in Section 5.2, process-based orchestration by Workflakes has been coupled with monitoring and diagnostic means different from those of KX. Furthermore, the design also enables experimenting with Workflakes as a stand-alone software coordinator, even outside the dynamic adaptation context altogether.

In the remainder of this Section, the design of a controller facility like Workflakes is presented. It encompasses - but keeps logically separated - the decision and coordination roles of dynamic adaptation; it also acknowledges the strong mutual dependency between the orchestration engine and the effectors it coordinates, by providing a tight interface between them with abstract control and reporting primitives, which differs from the loosely coupled interface connecting the other platform elements.

Design of a process-based controller

The core of a process-based controller, such as Workflakes, is of course its enactment engine. Engines can be centralized, or de-centralized, i.e., made up of a multiplicity

of task processors that are interconnected via some distributed communication and state sharing means. For dynamic adaptation, a de-centralized architecture is the better choice, because it allows to locate the task processors together with, or close to, target systems substructures, which can be themselves widely distributed. It also enables to delegate process fragments that pertain to those substructures to the most convenient task processor for "local" execution. Decentralization enhances the performance, robustness, and scalability of the enactment engine, and consequently of the dynamic adaptation platform as a whole.

Decentralization of the enactment architecture implies the presence of mechanisms for distributed data access and management across the task processors, including:

- Distribution of the specifications of the process: in the case of dynamic adaptation, the various process fragments.
- Distribution of the artifacts produced and accessed by the process: in the case of dynamic adaptation, information about the target system and its state.
- Distribution of the process resources: in the case of dynamic adaptation, the effectors to be employed.
- Distribution of the run-time process state; recall that the overall process state encompasses the state of each of its constituent tasks, the dependencies between tasks, and the data and the resources described within the process.

Since – as noted earlier on in Section 2.4 - a number of state-of-the-art process engines make available the necessary mechanisms for de-centralized enactment, in the remainder those aspects are assumed and the focus is placed on the architecture of a task processor in isolation.

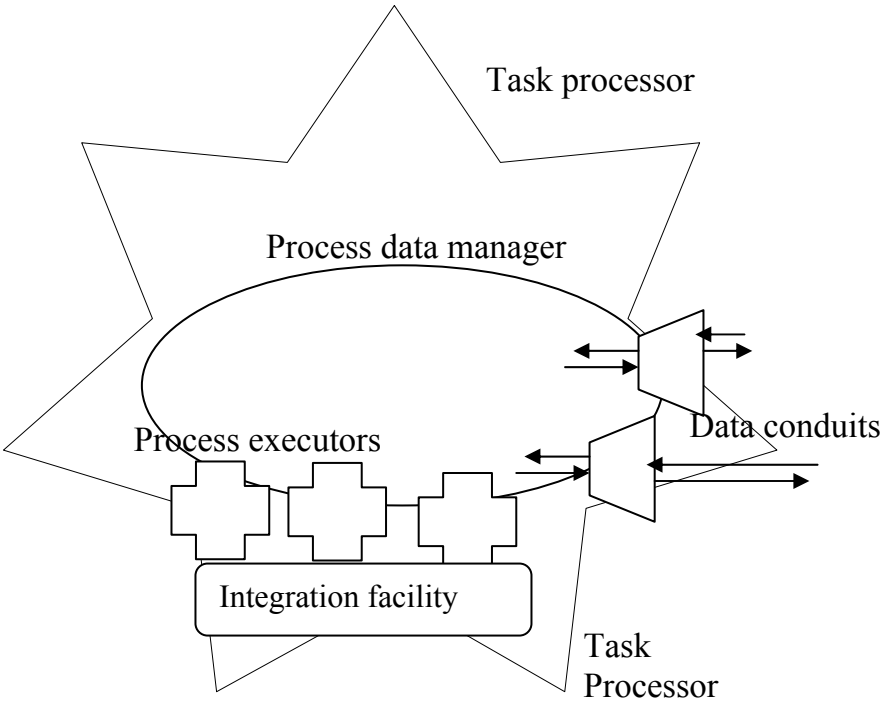

Figure 8: Abstract view of a task processor.

To introduce the discussion, Figure 8 shows how a generic task processor can be organized – independently from the specific process engine adopted, or the kind of coordination problem at hand. A task processor typically revolves around a *process data manager*, which is in charge to manipulate and maintain in a consistent fashion suitable data structures that capture all the information that is relevant to the enactment of the orchestration process. Such data includes an operational representation of the process specifications loaded within the engine, the current process state, any input or output data exchanged with external software entities, representations of or pointers to process resources and artifacts, and possibly other support or specialized information. The process data manager – in a de-centralized enactment architecture – is also in charge to interact, exchange information and

synchronize its internal state with the other remote task processors that take part to the same decentralized enactment engine.

The process data manager regulates the access to process enactment data on the part of other elements in the task processor. Those other elements are modules that either support process enactment and need to variously interact with the process data maintained by the manager, or interface the task processor with internal external components and tools that have some role in the process.

Strictly connected to process interpretation are the *process executors* in the task processor. They are intrinsic elements of the process enactment engine, which provide the machinery that ensure the correct execution of a process loaded in the data structures of the process data manager, according to the operational semantics of its specifications. Those facilities take the form of one or more computational modules, devoted to interpret the loaded process representation and to incrementally modify its state by scheduling, initiating and overseeing the enactment of process steps.

Process executors may need at times in the course of the process to employ external tools and applications to complete some work units. To that end, an *integration facility* can be used, which enables and abstracts those interactions. Integration facilities are usually dependent on the application domain and the tools they need to male available to process executors.

The task processor also needs to be equipped with *data conduits*, because a quantity of input and output data may need to be exchanged with external programs. Input data must be relayed and converted into adequate formats suited to be used within the

process engine for its own purposes, in particular for updating the information maintained in the process data manager. Also data generated as a by-product of the process execution may need to be output and reported to external entities, possibly after suitable re-formatting. The data conduits therefore need to operate in both directions; they also may either work in batch mode (i.e., reading/writing data from/to permanent or semi-permanent storage such as a database or the file system), or in streaming mode (directly communicating with other executing applications, which produce/consume immediately the exchanged data).

It is clear how in the context of an-application-to-application coordination process, such as that of dynamic adaptation, the importance of data conduits becomes very significant. For example, information captured in models of the target system, as well as any update of its state, may be conveyed via these data exchange modules. A data conduit constitutes also a valid way to implement the communication channel between the diagnostic facilities and the controller, relaying process triggers to it. Finally, data conduits can be used to transfer information to and from the part of the system that is in charge of actuation, and of interfacing with effectors.

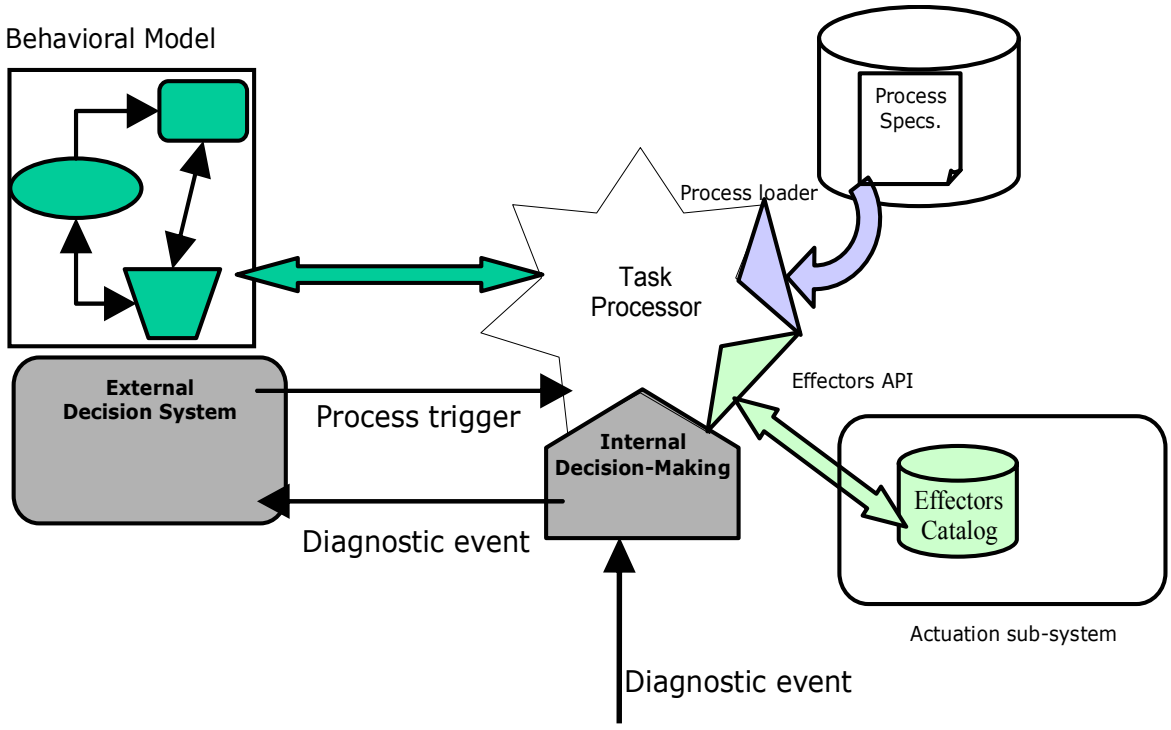

Figure 9: Design of a process-based controller.

Coming to the role of a task processor in the context of dynamic software adaptation, the diagram of Figure 9 highlights the various interfaces that enable the functional integration of a task processor embedded in a controller within the conceptual architecture previously described. To that end, three major interfaces can be recognized, dedicated to *process loading*, *decision making,* and *effectors control*.

The *process loader* is in charge to acquire some process specification and to load it into in an appropriate operational form that is executable within the task processor. Process loading can be performed both in "push" and in "pull" mode. The push mode is implemented by an entity (a user, or some software) which explicitly requests the process loader module to load some determined process specifications. The pull mode instead is implemented by the task processor itself, which is able to react to some

event (for example, a process trigger), asking the loader to search for a determined process specification, fetch it and load it.

Irrespective of the means used to load the process specifications, which are not constrained by this architecture, the logic adopted is completely incorporated within the process loader. In both cases, the loader can make use of another component, which serves as a *process repository*. In practice, it implements a database that keeps in store a collection of process specifications (describing process fragments) that potentially need to be made available to the various task processors. No assumption or limitation on the nature and the format of the stored specifications need to be adopted at this stage. Also the way in which the process repository is populated may widely vary: for example, the repository might enable dynamic addition of new process specifications, or also the update of existing specifications with new versions.

Figure 9 also depicts the juxtaposition of the decision facilities and the task processor within the controller. An internal decision-making module may be put in charge to make a first evaluation of incoming diagnostic events, which can lead to the selection – whenever necessary – of some process fragment to be enacted. The simplest way to implement such an internal decision module can be either via pattern matching or query mechanisms, which relate the format and the content of the incoming events to some process specifications already present in the task processor, or contained in the process repository and ready to be loaded via the process loader.

For more complex decision scenarios, however, it must be also possible to use an external decision-making component with substantial computational capabilities. Using an external decision facility allows to isolate the decision logic, which can be

at times quite involved, and delegate it to an external system. It is thereby possible to separate more clearly decision aspects (for instance, which process fragment – if any – is the best suited to achieve a certain adaptation under given conditions) from co-ordination aspects (i.e., how to enact the selected process according to its specifications).

Such a decision facility may be a third-party or otherwise stand-alone component. To oversee the decision, that component must be enabled to access the original diagnostic event, as well as any portion of accumulated knowledge about the current state of the target system (captured by the gauges, and based upon the behavioral model) and of the process, as maintained in the task processor. In that case, the internal decision module serves as a connector between the task processor and the separate decision system.

Directing actuation

Coming to the interface of the task processor with the actuation role, which is arguably the feature that most strongly characterize a process engine devoted to dynamic software adaptation, Figure 9 shows an Application Programming Interface (API) and a connection with an *actuation sub-system*. That relationship is displayed in greater detailed in Figure 10.

The actuation sub-system is in charge of instantiating, managing and guiding effectors destined to impact the target application, as required by the enactment of the dynamic adaptation process. It is made of a few essential components.

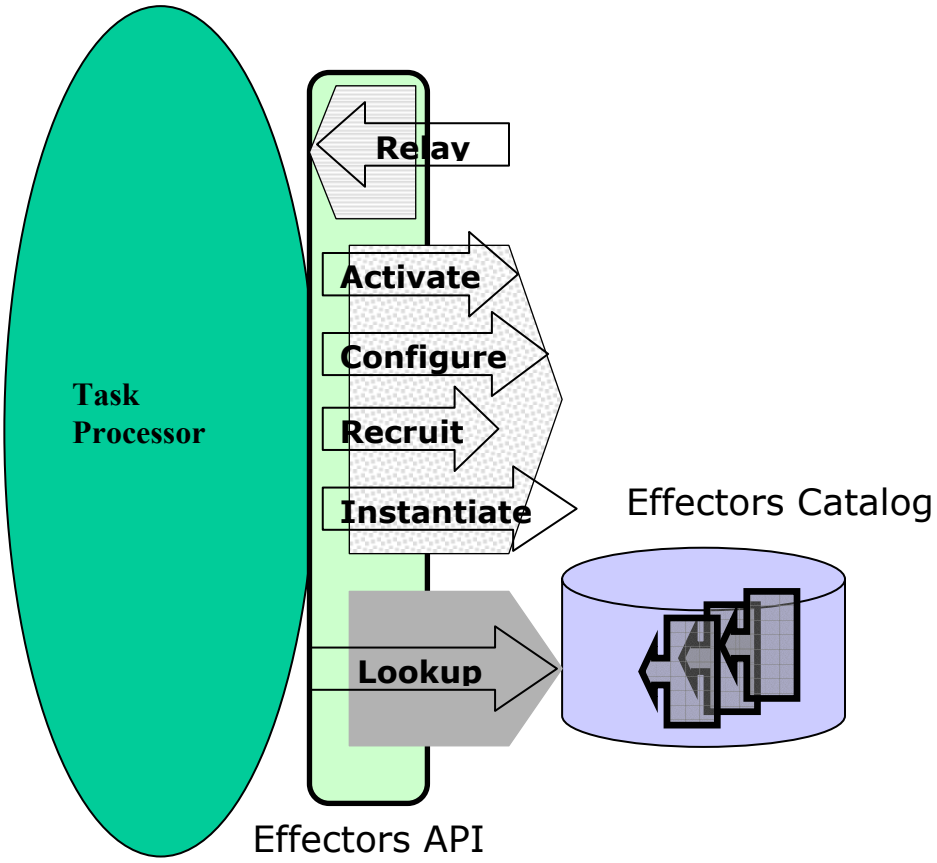

Figure 10: Interfacing the task processor and the actuation role.

The first component is an *effectors catalog*. In practice, it implements a repository that keeps in store a collection of information about code artifacts that represent the effectors that may be needed for the dynamic adaptation application at hand. The information stored in that repository might differ with respect to aspects such as the effectors' purposes, their functionality, their methods, their technological underpinnings, and more. For example, multiple effectors may have the same purpose and functionality (i.e., their execution aims at producing equivalent side effects from the point of view of dynamic adaptation), but may achieve it in different ways, and may be implemented or compatible with different technologies. In principle, the

effectors catalog must include some mechanism (such as associated meta-data) for the purpose of describing, discriminating and selecting suitable effectors for each task of the dynamic adaptation process, and for the computing environment to be effected. In some cases, also the executable code of the effectors – or some pointer useful to retrieve such code – can be included within the repository; in other cases, for instances when effectors come already embedded with the target components, only the runtime handles to those already instantiated effectors is present in the catalog.

Notwithstanding the abovementioned possible heterogeneity of effectors, a significant amount of standardization in the interactions between the process engine and the effectors it coordinates can be achieved. For that, a high-level, conceptual interface can be employed, which also helps in keeping cleanly decoupled the coordinator from the subject of coordination across technologies and application contexts. Such an interface is relatively simple at a high level of abstraction, since it provides only a few primitives, which constitute a conceptual *effectors* or *actuation API*:

- *Look Up*: the ability of querying the effectors catalog and obtain in response what code artifacts in the catalog correspond to suitable effectors.
- *Recruit*: the ability to summon and obtain control of some effectors, in order to exploit them for the purpose of the adaptation at hand.
- *Instantiate*: A specialization of the Recruit primitive, which implies the ability of creating new instances of effectors within the runtime environment of the controller or the target system, for the purpose of their execution. Recruit applies primarily to pre-existing effectors, and Instantiate implements the same semantics on new effectors' instances.

- *Configure*: customize effectors to the specific conditions of the adaptation they are about to carry out, by means of parameter-passing, variable setting and other similar means.
- *Activate*: launch effectors' execution on top of target system components that need to be impacted. That may involve the preliminary deployment of the effectors.
- *Relay*: make available the means for effectors to report back to the task processor the results of their work on top of target system components. Since the effectors' work can have long-duration and can occur asynchronously with respect to activation, it is not usually convenient to model the passing of results in a request/response fashion, like that of Remote Procedure Calls (RPCs). It may be more appropriate and general to equip the effectors in use with a data conduit, which the effectors can employ whenever they need to relay data back to the task processor.

With the exception of Look Up, all primitives tend to be strongly dependent in their implementation on the technologies employed to develop the effectors, with their idiosyncratic properties. In the Figure, the conceptual, high-level API is exposed as a whole towards the task processor, thus providing it with a single, uniform manner to interact with the effectors and command them. That API aims at effectively hiding from the task processor any idiosyncrasy linked to the possible specific characteristics of different effectors, and at interacting with said effectors in a transparent fashion.

The counterpart of that conceptual interface at the implementation level can be constructed effectively by distinguishing among three different slots, which group

together subsets of the API primitives, and are differently implemented. The three slots, which are shown in Figure 10, are the following:

- a slot for the Lookup primitive, whose implementation does not depend from the nature of effectors and is therefore always available;
- a slot that groups the Instantiate, Recruit, Configure and Activate primitives, whose implementation is technology-dependent. To accommodate multiple implementations, this slot can be filled by adopting a plugin mechanism. Multiple plugins may exist, developed according to the various available effectors technologies, and can be loaded into the slot dynamically. A plugin is selected and used every time a certain effector is looked up, to allow the task processor to interact with the effector respecting the semantics of the primitives in the slot, and at the same time in compliance with the technology of the effector;
- a slot for the Relay primitive, which is implemented also by means of technology-dependent plugins. The plugin for this slot is selected, and passed to the instantiated effector as part of its Configure stage. It provides to the effector a communication channel back to the task processor (for instance, a callback mechanism) to support the semantics of the Relay primitive in compliance with the technology of the effector.

An important result of this architectural design is that the interaction between the coordination and the actuation roles is kept simple, independently from implementation concerns, because of the limited number and the high level of abstraction of the primitives in the effectors API.

3.3 Applicability and scope

The spectrum of issues, problems and applications that can be addressed in principle with dynamic adaptation techniques is rather vast. It is therefore important to define the likely boundaries of applicability of the proposed approach, considering its most distinguishing traits, that is, the use of process technology and the externalized stance with respect to the target system. Therefore, although process-based coordination can be employed also in internalized dynamic adaptation solutions, the discussion in the remainder of this Section focuses on its use within an externalized platform, and addresses two issues: what kind of dynamic adaptation problems are well or badly suited for process-based coordination; and what kinds of target systems are feasible or unfeasible.

Applicability scenarios

Let us consider the four declared major areas of application for autonomic computing and similar initiatives, i.e., *self-configuration*, *self-optimization*, *self-healing* and *self-protection* [32]: the application of process technology to the coordination role of dynamic adaptation enables a variety of scenarios which apply, at a minimum, to the problems of automating the configuration, healing and optimization of the target system, and can extend also to partially cover its protection.

With respect to the target system configuration, a dynamic adaptation process may be used first of all to coordinate and automate the deployment of a distributed software application onto an available and suitable computing infrastructure. Deployment is an ensemble of possibly complex but mostly repetitive technical procedures, which are sufficiently self-contained and with a limited set of states and outcomes; therefore, its

automation is feasible, and likely to bring about significant advantages, especially when it is repeated on computing environments of the same kind. In fact, numerous commercial solutions exist, which cover some parts of the automated deployment spectrum, such as installation [33], distribution [34], or update [35]; they are increasingly common, and today a large part of commercial software packages, including operating systems distributions, come with their own automated installer and updater. Most of those solutions, however, operate on single software packages, either in isolation on a single host, or by volume on a number of similar host machines. The dynamic adaptation focus is instead on comprehensive, customized and orchestrated deployment of multi-component applications on the part of a general-purpose automated deployment facility, similarly to systems like the Software Dock [36]; in that context, specialized installation, distribution, updating, etc. utilities like the ones cited above can take the role of effectors, which can be employed under the coordination of the generic deployment facility.

A distinction can be made between the initial application deployment and later re-deployments: the initial deployment generally follows an explicit decision by some system administrator who specifies when, where and how to dispatch and start up all the application components – possibly with the support of appropriate configuration tools. By enabling the insertion of specific triggers from an external conduit in the control loop of a dynamic adaptation facility, it is possible to unify the initial proactive system deployment and any subsequent re-deployments (partial or complete), which may occur either automatically as a reaction to some runtime condition in the deployed target system, or again following a directive coming from

an external entity, such as an administration cosole. A process that reconciles all deployment cases can therefore be conceived, which re-uses the same overall logic, and the same knowledge to describe and predicate upon its resources (the components and packages of the software to be deployed, the computing environment at hand with its topology, its characteristics and its state, the various deployment facilities that may serve as effectors for this specific kind of adaptation, etc.).

Automated (re-)deployment addresses a number of configuration issues, by enabling the on-the-fly addition or replacement of relatively coarse-grained features, components and services in a system: an example is the automation of the various administration concerns related to a system-wide upgrade without service downtime (also known sometimes as *staging*) An example of a staging process, in the context of a Workflakes experiment, is reported in Section 5.1.

In the course of deployment, another configuration issue that can emerge is the elimination of any potential or detected conflicts with applications previously deployed on the same distribution architecture. Another form of finer-grained configuration that is naturally intrinsic to deployment, but also occurs in many other adaptation contexts, is the application of appropriate systems settings and parameters to single components, or to subsystems, or to the target system as a whole. The dynamic adjustment of such settings to reflect the initial state of the computing environment and of the target system at deployment time, as well as any subsequent variations thereof, can be resolved with a reactive process that is fed with triggers indicating that variations are needed.

Composite processes that variously combine (re-)deployment and parameters setting have the potential to cover the vast majority of the dynamic configuration needs of the target system. As a special case, it is worthy to consider the configuration - or re-configuration - of the architectural layout of a distributed application, in terms of its components and connectors. Process tasks enacting deployment strategies can take care of the delivery and the launch of the various components on certain hosts, while other tasks provide those components with the appropriate discovery, location and networking settings, which allow components to find and connect to each other, thus effectively putting in place the connectors of the architecture, which enable components to interact. Those deployment and parameter-setting tasks need to be appropriately interspersed, in order to build or modify the system architecture in an orderly way, and the process can be fragmented in such a way to take care of each recognizable architectural sub-structure autonomously. An example of architectural re-configuration supported by Workflakes is reported in Section 5.3.

With respect to target system healing, one can recognize two major categories of adaptation: fault recovery and fault avoidance. Fault recovery is fully reactive by nature; assuming that diagnostics can correctly indicate the kind of fault occurred, the main issue in fault recovery is related to the strategy chosen to fix or *survive* (i.e., overcome) that fault, while minimizing adverse effects on functionality and performance. That choice largely depends also on where the fault has occurred among the many layers of the stack that underlies the implementation of a distributed software system. Recovery from hardware, system or network failures is clearly quite different from, say, application level faults: in the former case it is more likely that

the recovery put in place with dynamic adaptation cannot actually fix the fault, but rather overcome it by changing the configuration of the system in an appropriate way. Limiting the discussion – for the sake of brevity – to the latter category, some of the possible options are the isolation of faulty components or subsystems, their shutdown and/or replacement, or more granular repairs that impact inner modules and settings of the target components. All of those remedies may require the re-configuration of target system elements and also some software re-deployment. It must be noted that the enactment of fault-recovery adaptations may incur in a number of unforeseen complications, due to the possibly unreliable state of the target system in presence of a fault, and also because the diagnostic role may have uncovered the fault without necessarily discovering its reason. Thus, alternative courses, re-planning, backtracking, and other similar devices for managing those contingencies are often prominent in processes that orchestrate fault recovery, increasing their degree of complexity.

Fault avoidance requires possibly sophisticated predictive capabilities by the diagnostic role of dynamic adaptation, which must be able to infer the probable occurrence of a fault in the future, on the basis of the current snapshot of the state of the target system and possibly its history. Since fault avoidance takes a proactive stance with respect to fault management, in an attempt to preserve target stability and improve quality factors such as availability and reliability. One difference with respect to fault recovery processes is possibly that a constrained time window for effecting the adaptation might need to be respected. Also, the activities in a process aiming at the prevention of a fault are likely to lean more towards adjusting running

components, their state and their operation parameters to skirt trouble and approach full efficiency, rather than repairing them, which often implies taking them off, since restarting a component is at times the surest way to fix it.

With respect to target system optimization, the focus is on the management of the resources employed by the distributed applications as a whole, and by each of its components. Optimization processes try to automatically and dynamically tune those resources, a goal that translates in practice into a continuous exercise in balancing trade-offs. First of all, it is necessary to strike the right balance between maximized performance for end users and minimized load for the computing and communication infrastructure, under the ever changing usage conditions typical of many distributed systems and services. On top of that, the allocation of certain resources must be balanced among the target system components that compete for them at any given time. Notice that those two different trade-offs are not orthogonal, and may interfere with each another.

Optimization processes may be the most granular with respect to the adaptation they pursue, as well as the ones that demand the fastest turnaround time between the reception of a trigger and the fulfillment of the corresponding actuation. Orchestrating some target system optimization may therefore require in certain cases specific logic and activities, but many times may also re-use concepts and practices that are typical of healing and configuration. For example, an optimization process aiming at the dynamic load balancing of user requests for some service may need to deploy new instances of the service software, re-configure the settings of the load balancing facilities to take advantage of those new instances, and divert requests in excess from

overloaded servers before they might crash. The net overall result is an optimization of incoming traffic and requests, but that effect is in fact achieved by a combination of re-configuration and fault avoidance techniques. An example of such an optimization, supported in a Workflakes experiment, can be found in Section 5.1.

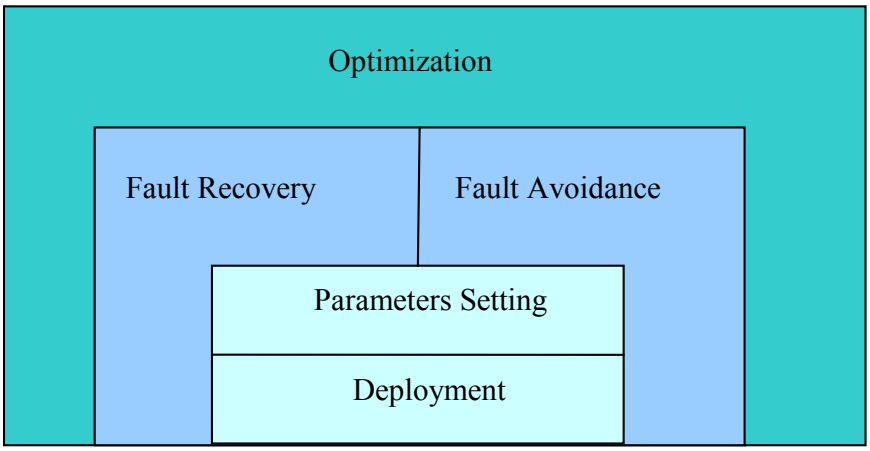

Figure 11: Dynamic adaptation scenarios.

Figure 11, shows the relationships between the various dynamic adaptation scenarios discussed above, and how solutions for those various scenarios are likely to build upon one another.

As for target system protection, in a number of cases it can be equated to a form of healing. For example, securing the target system against some attack is not conceptually different from preventing a fault. Once again, it is only a matter of semantics: in the latter case the cause of the problem is incidental, while in the former case it derives from a malicious intent. Therefore, the same logic guiding fault avoidance processes may apply for example to the rejection of a Denial-Of-Service attack. Recovering system components infected by a virus, and getting rid of the infection can instead be approached at times as a fault recovery problem, with the

mandatory extra requirement that system functionality should be preserved during and following the adaptation that eliminates the virus.

In a variety of situations in which protection problems can be handled like healing issues, dynamic adaptation techniques similar to the ones described in this work can apply: among other examples, SABER [159] and Willow [31] approach the problem of the survivability of software systems and services in those terms.

A major difference between system healing and system protection ensues when, to ensure protection, it is necessary to take measures that have an impact externally to the target system proper. For example, to protect some service offered by some system within a certain organization, it may be necessary to modify the access rights, trust policies, network topology, or other relevant elements at the level of the whole IT infrastrucuture under attack, or at least of some substructure thereof. Entities that are affected by those kinds of measures do not belong to the target computing environment, unless it is defined in an extremely broad sense; that contrasts both conceptually and practically with the way the control loop of dynamic adaptation is conceived, with its endpoints well rooted in the target system (see Figure 2). In particular, that Figure assumes that the actuation capabilities needed for dynamic adaptation are all exposed in some way by elements of the target system; that assumption in turn provides a well-defined context for both the coordination and actuation roles. In order to undertake adaptations that affect the organizational computing infrastructure at large, that assumption must be broken, in favor of a much broader and all-encompassing vision in which a number of dynamic adaptation roles must work across the whole organization and all of its information systems.

That is a different, challenging endeavor, which goes beyond the issue of providing a technically suitable software infrastructure, but also needs to embrace and resolve a number of organization-wide concerns. In other words, it entails moving from level 4 to level 5 in the autonomic capability model previously introduced in Section 2.2: level 5 requires, besides the full automation of adaptation of level 4, awareness, support and integration of organizational and business issues and policies within the autonomic facilities. That represents a path along which the research presented here can evolve in the future: one line of investigation according to which such integration could proceed is the federation of business processes with dynamic adaptation processes, which remains however out of the scope of this work.

Target system feasibility criteria

While the generic concept of dynamic adaptation applies in principle to about any distributed software system, a number of considerations can be made on the kinds of target systems that are viable for the approach proposed in this work.

A first issue that impacts the suitability of an externalized coordination facility, such as a process-based one, regards those systems and components that come with some built-in adaptation logic of their own. If some internalized dynamic adaptation provisions – with its own decision and control policies - are active in some components of the target system, those policies may conflict with a global adaptation plan pursued by an externalized platform. That makes for a case of “process-in-the-tool syndrome” [37], that is, the interference in the enactment of a coordination plan on the part of the very subjects of coordination.

There are ways to overcome the process-in-the-tool syndrome and achieve a gracious co-existence of externalized adaptation orchestration with internalized adaptation provisions. For example, when some potentially disruptive internal adaptation logic is present, documented and active, it is many times possible to design the orchestration process to incorporate it in its own end-to-end plan. That can happen, for instance, when the external coordinator is empowered to activate and de-activate the internal adaptation mechanism at will, by issuing from some specific process step an effector that has that side effect. That way the overall process can at certain junctures accept to delegate certain forms of adaptations to the internalized features, but maintains a control on when allowing delegation. If the local, internalized adaptation could interfere with a certain more global adaptation plan , the coordinator could switch off the internalized mechanism and enforce alternative ways to resolve the same issue.

Another possible issue regards the class of target systems that must operate under precise timing constraints. Externalized dynamic adaptation may insert a level of uncertainty with respect to the timing of operations of its target systems. The control path put in place by the adaptation loop needs some time to be traversed from the production of monitoring data to the execution of the appropriate actuation – no matter how efficiently it is implemented. That can be defined as the *end-to-end response time* of the dynamic adaptation control loop, which may influence the execution time of the target system, or at least of that portion of the system that undergoes adaptation.

There are a number of components to that response time. One component is inevitably a communication delay, due to the fact that both the target system and the

dynamic adaptation platform implementing the control loop may be widely distributed. Even in case stable and reliable communication channels which minimize the delay and variance in communication time can be assumed – thus allowing to assign an upper bound to that contribution – each of the diagnostic, decision, coordination and actuation components in the dynamic adaptation platform also provides its own contribution, deriving from the work they need to perform.

Actutation is probably the least problematic aspect, since effectors are limited snippets of computational code which can many times be developed in such a away that their execution can be appropriately bounded. The other roles are likely to pose more complex problems.

In particular, a considerable slice of the dynamic adaptation response time is likely spent within the coordination role. When expressing coordination, for instance as a process, the temporal aspect is paramount in all but the most trivial cases: the various activities that are to be coordinated need to be properly sequenced, and executed accordingly, with later activities often depending on the reported outcome of earlier ones. To that intrinsic temporal aspect of coordination, it is also necessary to add the execution time of the coordination facility itself.

It may be particularly hard to characterize the component of the response time contributed by the adaptation process overall with a constant upper bound (which could be accounted for as a penalty, and added to the normal operation time of the system). Each specific kind of adaptation may take a different time to complete, and there may be considerable variance even among distinct occurrences of the same kind of adaptation. That happens because of the dynamic nature of the coordination

process and of the dependencies between steps: each step in the task decomposition hierarchy may unfold each time in very different sub-hierarchies of tasks, for instance when exceptions are raised and their corresponding handler sub-processes are fired.

Given the significant delays that can be introduced in the response time of an externalized dynamic adaptation loop by the orchestration of the whole process, and the level of variability of those delays, “hard” real-time systems, in which all operations must observe a fixed temporal upper bound, may not be very well suited for this approach: while carrying out dynamic adaptation could resolve a number of problems and criticalities in those systems, it may also induce an unpredictable extra latency to their execution.

Some externalized dynamic adaptation facilities that are applied to cases with hard real-time constraints exist; however, it can be noticed how they employ simplistic approaches to, or even bypass alotgether, the coordination role. See for example [181], which addresses survivability concerns. In it, each adaptation is a single intervention, carried out by a carefully chosen computational effector, whose execution time can be bound.

With respect to hard real-time constraints, internalized adaptation provisions may have an advantage over externalized platforms, since they can achieve a better response time, and a lesser variance thereof. In fact, following the introduction of the dynamic real-time model in the late 80’s - early 90’s [39], a quantity of internalized approaches and systems have been researched, to overcome the limitations of the “binary” (guarantee or reject), static service management scheme for real-time systems. RESAS [51] was possibly one of the earliest examples of those dynamic

adaptation solutions: it promotes a programming model for the specification, development and enactment of internally adaptable applications, aimed specifically at enforcing reliability and timeliness in real-time systems. A multitude of other works have since then tried to address in a dynamic and adaptive way common concerns in that field, related to issues like resource management [38] [52], and QoS (re-)negotiation [41]. Mostly, those efforts rely on some form of specialized middleware, which provides, in conjunction with support for real-time distributed communications and computations, a number of built-in adaptive features, mechanisms and policies that impact either the operating system level, or the application level, or both. Those solutions tend to optimize the closed set of adaptation operations they provide, to bound and minimize the induced latency, and sometimes also incorporate a model of their response time [40]: they can thus use that model for their decision-making, and opt for an adaptation compatible with the timeliness guarantees required by the normal system functionality, if such an adaptation exists.

Those techniques could be somewhat extended also to externalized dynamic adaptation solutions. In the case of KX and Workflakes, that means that the platform would need to offer a closed set of possible adaptations, which would be mapped to a number of process fragments of limited complexity and dynamism, such that their enactment could be time-bounded with confidence. The diagnostic and decision component would then need to take in account temporal aspects in their work: each time the need for some adaptation were detected, only those adaptations that could be safely completed within time bounds would be approved and fired. Such an extension

to hard real-time system, may represent another open theme of investigation for the future development of this research.

The same techniques apply of course also to “soft” real-time systems, i.e., systems for which timeliness of operation is a significant *property*, rather than a strict *constraint,* and for which the failure to operate within time bounds but does not necessarily equate to a fault. A significant part of the distributed applications of real-time computing, in particular over the Internet, fall in the soft category, including, among others, the increasingly prominent category of networked multimedia applications, such as audio/video streaming. Soft real-time systems – as opposed to hard ones - are also a more viable category of targets for externalized dynamic adaptation, given their more relaxed timeliness requirements. Even for soft real-time systems, however, the matter of the response time of the dynamic adaptation loop is very important: an adaptation that is intended to impact some target operation may be effective only if it completes within a certain amount of time (for example, in less or comparable time with respect to the temporal requirements for that operation), while it may be useless or even counter-productive in case it takes effect too late.

In this work, the dynamic adaptation of soft real-time systems has been experimentally addressed, with the intention of validating the process-based coordination approach under such demanding requirements, and testing the limits of its applicability. One of the Workflakes case studies regards a typical soft real-time distributed application, since it addresses the synchronized viewing of an audio/video stream provisioned at different compression levels to a group of remote users, who work as a team and employ multimedia clients with diverse profiles of host and

networking resources for their collaboration. Section 5.2 discusses how the process-based dynamic adaptation exerted by the Workflakes prototype attempts to optimize the settings of each client, in such a way that it can view the stream at the best possible level of compression given its available resources, while remaining in sync at all times with the other members of the group. Notice how that experiment considers the group of clients as well as the server as participating all together in the same target system, which is seen as a distributed CSCW application.

Another feasibility issue regards the number of self-standing distributed components taking part in the target system, each of which may, in certain conditions, become a subject of dynamic adaptation. While the majority of distributed applications are constituted in practice of relatively few autonomous, inter-communicating computer processes, each of which, in turn, is made of a limited number of recognizable components, there are cases in which a single application results from the interactions of hundreds or even thousands of components: Significant cases can be found in the fields of agent-based computing (see [42]) and grid computing (for example, [43] [44]). A large number of potential adaptation subjects obviously puts to test the scalability of any externalized, end-to-end dynamic adaptation platform, and of each of its roles, including of course process-based coordination. There are some works that address the dynamic adaptation of classes of distributed systems that tend to have a large number of components, such as the afore mentioned agent [45] [46], and grid applications [47]; however, the application of process technology to the orchestration of dynamic adaptation of that scale has not been yet sufficiently experimented with,

either in the present work[5] or (as far as it is known) elsewhere. One can only try to infer the feasibility of process technology to orchestrate effectively an adaptation involving hundreds or thousands coordination subjects, by looking at some results achieved by state-of-the-art distributed architectures for process enactment in other, similarly demanding, coordination contexts. For instance, in the ALP [48] and UltraLog [49] DARPA research programs, the decentralized architecture provided by Cougaar [25] has been used to run logistic planning workflows in which hundreds of coordination subjects participate. It may be important to notice that the time scale for the enactment of a logistic plan is is in the order of hours and days, therefore, likely to be orders of magnitude less demanding than that of the dynamic adaptation of running software; on the other hand, the process logic needed for such a large scale exercise is quite sophisticated, and complex to enact. Therefore, a clear conclusion on the feasibility of process technology for the coordination of systems of such a scale can not be easily reached, and this issue remains open for further investigation and experimentation.

3.4 Critical assessment of the model

The way process technology is used in this work to achieve the orchestration of dynamic adaptation, described in Section 3.1, is based on a number of ideas and assumptions; among the most important ones, there are the reactive nature of the process, its fragmentation, and the correspondence of the various fragments with triggers originating from the diagnostic or the decision role.

[5] None of the applications that were made available as case studies has those characteristics.

After having described how that model enables the definition of the architecture of a process-based coordinator, and the scope of its applicability, it becomes important at this point to discuss the assumptions at the basis of the model, or rather some of their possible implications, in particular to see whether and how they could impose limitations to the approach.

One major criticism is that the approach may remain limited to handling well known and well-proved dynamic adaptation contexts and solutions.

With respect to context, that problem can derive from completely pre-determined associations between triggers and the corresponding process fragments. One motivation for such a strict correspondence is that a trigger, on the one hand, represents an entry point into the reactive adaptation process, and, on the other hand, signifies some (critical) condition related to the target system. To be able to properly codify the trigger, in terms of the information it carries to the process and in order to enable the firing of the corresponding process fragment, it may be necessary to know the condition in great detail. That need contrasts with the greater flexibility available in principle for the diagnostic role: the diagnostic facilities may be able to carry out different kinds of inference on the basis of the flow of raw information originating from the monitoring role, and report upon a variety of conditions relative to the target system, including some whose semantics and format could have not been previously analyzed, recognized and codified. Such a level of diagnostic sophistication may remain inaccessible to the orchestration process, whenever any events corresponding to unknown or new target system conditions cannot be mapped on the fly to a corresponding reactive process fragment, but, on the contrary, each process fragment

is intended to deal with some target system condition defined *a priori*. The mediation operated by the decision role between diagnostic events and process fragments can be particularly important to avoid this kind of limitation.

Another possible limitation can originate from design decisions on the level of process fragmentation. It is relatively easy to define fragments in such a way that each of them handles fully on its own some macroscopic target system condition, and captures a complete, internally coherent plan that moves the target system as a whole from a well-known, diagnosed state to another well-known – and more desirable - state (such well-known states are sometimes called *postures* of the target system [31]). Those fragments are likely to be of considerable size and reach, and the orchestration of the dynamic adaptation for each condition is hence completely regulated by a single process fragment, in a fully planned ahead fashion.

Such a planned-ahead approach aims at producing scripted processes, with a limited number of plausible paths and a few, predictable possible outcomes in response to any given trigger. That kind of process design is attractive since helps making different process fragments as disjoint as possible, which simplifies the construction, understanding and maintenance of the overall process. Planned-ahead design also reflects the way process technology is most commonly employed in other, more traditional application domains, such as business or clerical work; in those contexts, technologies and tools for process formalization and enactment are typically introduced to automate from start to end already established and proven work practices or procedures.

Unfortunately, in the peculiar application domain of dynamic adaptation, a likely consequence of a planned-ahead approach with coarse-grained fragments is a lack of open-endedness of the overall process, since those fragments are self-contained, have limited inter-relationships and hence are hard to compose with one another to form different plans. That implies that the process can formalize and orchestrate the dynamic adaptation of the target system only for those conditions whose solutions has previously been conceived and developed (and also thoroughly tested and validated). Within a given target system, those conditions map to the subset of expected problems. The guidance provided by the process for those problems is certainly valuable, since it guarantees that the corresponding solutions are faithfully automated according to a repeatable, consistent and controllable plan. However, target system conditions that are known, but whose resolution is not yet well understood, cannot be handled in that manner. To make the target system more fully autonomic, a process-based dynamic adaptation facility that employs a fully planned-ahead approach may always need to be supplemented by different kinds of management provisions, for taking care of those "harder" cases.

This limitation is not inherent in the use of process technology for dynamic software adaptation; rather, it is a design issue. A more desirable alternative to the limitations brought about by a dynamic adaptation process that adheres strictly to a planned-ahead design exists. It requires the codification of the process as a larger number of finer-grained reactive adjustments to the target system, associated to equally fine-grained triggers. While this issue can be seen simply as a bias in a trade-off regarding the modality and the granularity according to which the overall dynamic adaptation

process is developed and structured, its implications are in fact quite vast. The use of finer-grained, elementary process fragments and triggers effectively provides the coordination facility with a “bag of tricks” that are lower-level, remain more generic and can be used in many situations. In other worlds, it can improve the composability of the dynamic adaptation process as a whole, thus enabling the construction of open-ended adaptation processes that may enjoy greater variability and flexibility than the scripted processes previously described. The concatenation of several of those low-level fragments may incrementally guide the target system to an optimal, stable state in a situational, open-ended way, rather than following a plan scripted a priori. Open-endedness enables the process engine to enact adaptation strategies that may considerably vary from time to time.

In part because the application of process technology to the orchestration of dynamic adaptation is still a rather novel field, planned-ahead design of dynamic adaptation processes currently remains dominant. Methods for the design of processes in an open-ended, fine-grained and incremental fashion are currently not well outlined, and deserve systematic investigation. Techniques that would enable to automatically construct open-ended processes as a composition of those fined-grained fragments are an equally open problem.

Also the experiments used for the evaluation of Workflakes, reported in Section 5, employ processes that are largely planned-ahead. However, they have been instrumental to reaching an understanding of the limitations of that approach and to start devising how to overcome those limitations, leading to some ideas on how to achieve a more open-ended orchestration of dynamic adaptation, which will be

presented in Section 6.3 and represent a major direction for the evolution of this research.

It seems clear that the ability of defining open-ended processes would considerably benefit not only dynamic adaptation, but also other application domains of the process-based coordination of software systems, such as those outlined in Section 2.4. The call for open-endedness in general stems from the observation that the number of possible states in which multi-component, distributed systems can find themselves in real-life operation condition is huge[6]. It is noticeable how that observation is also one of the facts that drive the promotion and adoption of autonomic solutions in complex computing environments. The flexibility of a coordination plan aiming to guide the behavior of an ensemble of distributed software entities in all situations should, in the ideal case, match the high level of variability that can be observed in the state of that distributed system during its execution, as well as in its contextual and environmental conditions.

[6] In an extreme, but telling example, IBM has estimated that these days a multi-tier software system built starting from multiple commercial-grade products and components may have as many as 10^{20} different configurations, resulting from the cross-product of all the configuration options of participating components [30].

4 Implementation

A major principle that has been followed throughout the implementation work has been that of re-using and composing selected existing technologies whenever possible. There are several reasons for this. First of all, the contribution of this research intends to be mainly in investigating and evaluating the concept of process-based orchestration and its potential for achieving externalized, end-to-end dynamic software adaptation; with respect to that goal, the development of a certain technical solution that supports that concept serves mainly for experimentation purposes. Then, one of the working hypotheses at the basis of this work is that process / workflow technology is at the present time sufficiently mature in its major traits to provide the right kind of support to the coordination role of dynamic adaptation. Furthermore, since one of the main concerns of externalized dynamic adaptation is the ability to handle not only brand new systems, but also component-based and legacy systems, it makes sense to adopt a similar stance with respect to the construction of the dynamic adaptation facilities themselves.

Therefore, implementation efforts have focused on using, adapting and integrating carefully selected existing process enactment tools, process specification paradigms and also effector technologies to be orchestrated; moreover, a lot of attention has been given to easy integration and interchangeability, in order to remain open to changes involving different implementation options of any of the constituting elements.

The experimental and development work for Workflakes has produced two successive implementations of the process-based controller design described in Section 3.2.

Besides a number of incremental improvements, derived from the insights and lessons learned from experience, the main difference between the first and the second version of the Workflakes implementation regards the way the dynamic adaptation process is represented loaded and processed within a Workflakes task processor.

In its first version, the Workflakes engine employed process descriptions that are expressed via a set of coding patterns directly codified in the Java programming language, which are then dynamically loaded and executed as Java classes and objects into the engine at run time.

In its second version, instead, Workflakes has adopted a state-of-the-art, high-level process specification language (Little-JIL [19], see Section 4.2 for details), and the Workflakes runtime incorporates appropriate class libraries and mechanisms to support the loading, interpretation and enactment of process specifications expressed in that formalism.

Clearly, as in any paradigm change that elevates the level of abstraction employed for some practice, that difference benefits primarily the amount of work that is needed to develop the orchestration process, and, consequently, its understandability and maintenance. Furthermore, by embracing an existing formalized process specification formalism, it is also possible to focus on the evaluation of how well current process technology addresses any specificities inherent to dynamic adaptation processes, which features are most useful and which others are insufficient or even missing.

4.1 Workflakes v. 1: coding the process in a programming language

Since its first implementation, Workflakes has relied on the process enactment core of the Cougaar open–source distributed platform [25] as the basis upon which to develop the runtime of the dynamic adaptation coordinator.

Cougaar is a Java-based open-source platform for the creation and management of large-scale distributed applications, whose centerpiece is a decentralized process planning and execution engine. Cougaar includes resident process representation and enactment capabilities, which owe much to the domain of logistic planning. Typical Cougaar applications regard in fact the elaboration and automation of military logistics plans, which involve a large number of coordination nodes and a multitude of resources. To that end, Cougaar includes some advanced features, such as real-time monitoring of the process execution, evaluation of deviations or alternative plans, and selective re-planning. Cougaar supports a mechanism for the substitution of the default process formalism specialized on logistics planning with others; it also supports composition of complex applications, via process federation.

The motivation for the selection of Cougaar as the basis for the Workflakes enactment engine is multifold: for instance, its support for large-scale distribution, coupled with its proven robustness and performance derived from the logistics application domain, even as a non-commercial prototype; furthermore, the availability of its code base as open source, plus the availability of know-how and support on the part of an active community of developers advancing and maintaining that open-source project; the option to integrate different process formalisms; finally,

the possibility of synergic work within the context of the DARPA DASADA program, in which members of the Cougaar development team were also involved, as representatives of BBN Technologies, which is strongly involved in maintaining the Cougaar open-source initiative and community.

Version 1 of the Workflakes implementation was built on top of Cougaar release 8.8, and adopted its typical run-time architecture, based on a number of largely autonomous decentralized task processors that are interconnected via a distributed tuple space, named the *Blackboard*. In Version 1, the main focus was to come up with an implementation that could represent a valid proof of concept and constitute an operational basis for experimenting with case studies and getting hands-on experience. For that reason, it was decided not to address at that stage the investigation of suitable process formalisms for dynamic adaptation, and how to integrate and enact them in the Cougaar runtime. Instead, the resident logistics planning was adopted, and the dynamic adaptation processes employed in the various experiments conducted with Workflakes Version 1 were expressed according to it.

In Cougaar, there is no dedicated process modeling facility, such as an editor of process specification documents. Processes are instead directly represented as executable code and programmed in Java. To that end, Cougaar provides a library of classes that capture many basic process concepts and abstractions within a set of relatively low level coding patterns [160].

The basic unit of a process in Cougaar is the task, which is represented by the homonymous Java class *Task*. A Task is seen as an action, and as such is modeled as a *Verb*, which identifies it. A Verb can have a *direct object* and multiple p*repositional*

phrases, taking *indirect objects.* Prepositions and objects (direct as well as indirect) are the cosntrcuts used to codify the bindings to a task of resources and artifacts, which are represented by objects of the *Asset* class and its subclasses. A task in a certain Cougaar process can therefore be represented by a signature in the form:

*<Verb> <Direct Object> [<Preposition> <Indirect Object>]**

for example:

Deploy Server Upon Host-Machine Using Tool

where *Server*, *Host-Machine* and *Tool* represent Asset types that need to be used for the task *Deploy*.

Other Java classes that take part in the specification of a task in Cougaar are *Preferences*, which in a typical example pertain to the temporal scheduling of the task (e.g., *within 1 hour*), but can also be predicates on other *Aspects*, i.e., quantitative or logical features attached to tasks and groups thereof. There are also *Constraints* that can be imposed on Aspects: a typical way to employ Constraints is to define precedence relationships among tasks, for example for sequential or parallel ordering. A single task is therefore by itself a collection of objects that needs to be properly instantiated.

Other Java classes in the same Cougaar class library, termed *PlanElements*, describe and regulate the grouping of process sub-structures made of inter-dependent tasks. Since Cougaar uses principally task decomposition, the *Expansion* of a non-atomic task into a full-fledged sub-process is the most important PlanElement type. An *Aggregation* is another PlanElement type, which represents a fan-in point for the process flow, in which multiple parallel tasks are merged into a single task. The

Allocation PlanElement type defines instead the runtime binding of asset instances to a task instance, against the direct and indirect objects in the signature of that task. There are other classes still, which provide mechanisms for the evaluation of *Workflows*, i.e., enacted process sub-structures with their attached aspects and preferences, and which can be programmed to oversee their planning, for example to guarantee that any imposed Constraints are actually met by the process hierarchy as it unfolds.

It is clear that the experience of the programmer with the class library sketched above is an essential tool to correctly convey the specification of a non-trivial Cougaar process in a set of Java objects, which get loaded and reside in the Blackboard of one or more task processors.

With respect to the runtime, Cougaar employs a plugin approach to compose and customize the functionality of each single task processor. Each plugin implements some particular logic or provides some specific capability; it interacts directly with the Blackboard, and, through the Blackboard, indirectly with other plugins or other task processors. For example, the process execution mechanisms within a task processor are implemented as a set of plugins that typically includes a scheduler, a state and plan evaluator, an allocator of resources and data, and possibly others. They subscribe to the Blackboard and start to evaluate, schedule, fire and expand process tasks, allocate resource and data assets to them, manipulate their state, and also execute any computational functions that may be associated to tasks. The runtime libraries of Cougaar offer base implementations for those *core plugins*, which are however devoid of any logic on how to deal with Blackboard objects representing a

certain process. It is another responsibility of the Cougaar programmer to specialize by inheritance those base implementations, in order to correctly handle the semantics of the process specification objects that he/she loads in the Blackboard.

Workflakes embraced the plugin philosophy of Cougaar, and its implementation was largely carried out on top of the afore mentioned facilities, by developing a set of specialized plugins. Those plugins implement together the architectural blueprints described in Section 3.2, and – as it is discussed below – supplement normal Cougaar functionality in two major ways: they implement facilities for the integration and handling of effectors as first-class process entities; furthermore, they offer mechanisms to dynamically load and in the engine of process specifications.

As shown in Figure 12, a typical task processor in Version 1 of Workflakes includes a number of plugins. Several plugins are what Cougaar calls *Logic Data Model (LDM) plugins*. They reify the design of data conduits discussed in Section 3.2, and are employed in the first place to import and convert KX gauge events in terms of process-relevant facts stored in the Blackboard; another LDM plugin is devoted to the interface with the effectors catalog; others can be used to convey and maintain internally the knowledge about the target system and its state maintained in the behavioral models, communicate that knowledge to other parts of the dynamic adaptation platform, or otherwise wrap generic external systems, such as decision facilities.

Moreover, the essential components overseeing process enactment are: an *Expander plugin,* which loads process definitions, spells them out as hierarchical decompositions of tasks and schedules them; an *Allocator plugin* that maps scheduled

tasks to resources (among which are effectors and target system components) as needed; and *an Executor plugin* that handles the instantiation and execution of effectors and encapsulates the actuation subsystem of the controller that regulates all interactions with the actuation role of dynamic adaptation.

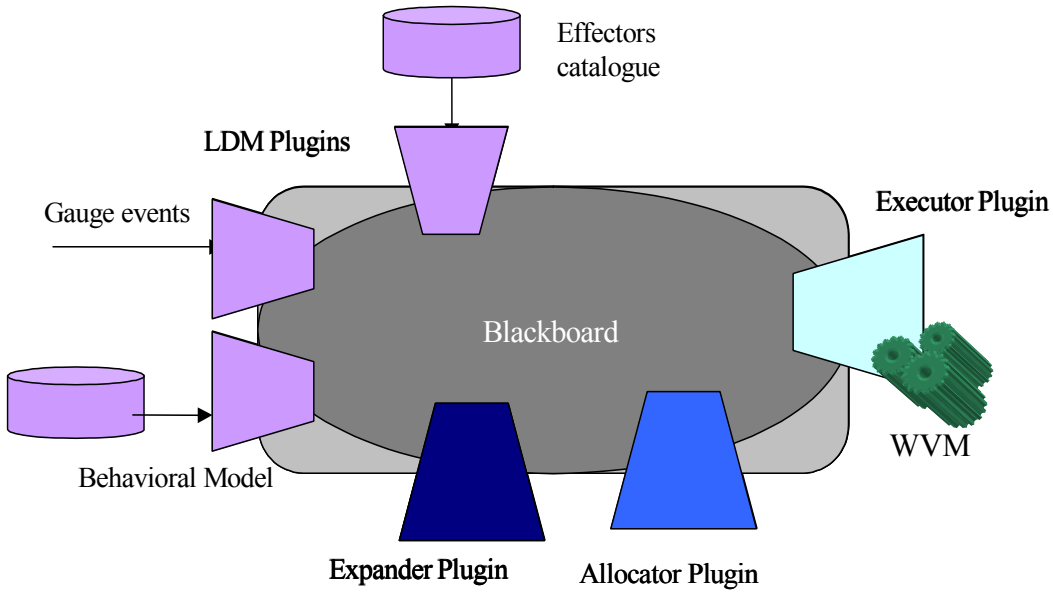

Figure 12: Representation of a task processor in Workflakes Version 1.

One aspect in which Workflakes substantially extends and specializes a generic Cougaar task processor is precisely the focus on integrating actuation facilities. In Version 1, the preferred option for implementing effectors is employing mobile code. Mobile code is intuitively a particularly apt technology for fulfilling the actuation role within an externalized dynamic adaptation infrastructure, since by its very nature it operates on the target system from the outside. That guarantees that new forms of adaptation computations can be easily developed and deployed at any time onto the target with minimal disruption to service operation, once that the target components are equipped with the facilities necessary to exchange and execute mobile agents. In

particular, the first version of the Workflakes implementation was integrated with the Worklets mobile agent platform, which the Programming Systems Lab had developed originally for unconnected reasons [70], and then adopted also for dynamic adaptation [7]. Worklets are code-carrying agents: each worklet is a container that carries Java mobile code snippets (termed *worklet junctions*), and deposits them onto one or more target components, according to a programmable trajectory. Once deposited, a junction is governed by programmable constructs that specify certain facets of its execution, such as conditional execution, repetition, timing, priority, etc. The agent transport facilities and the code execution environment are provided by Worklet Virtual Machines (WVMs) residing at all "stops" in a worklet trajectory. The availability of WVMs embedded in all the target system components that may need to be visited by incoming Worklets is therefore a pre-requisite to use this mobile code approach.

Workflakes regards effectors' code snippets as first-class resources for the dynamic adaptation process. It takes care of selecting appropriate effectors, configuring them with any data of interest coming from the process context (that is, the parameters flowing into the process task that instantiates and invokes an effector), loading them onto worklets, and dispatching those worklets onto the target system, to induce the side effects intended by the adaptation process. On their part, worklets effectors are configured to return to base in all cases, whenever they are finished with their work: that way, they report back to the process the outcome of their work, which may be critical to steer the rest of the adaptation this or that way.

All of that occurs typically as part of the execution of a leaf task in the adaptation process, and is accomplished by the Executor plugin. That plugin incorporates a WVM, which acts as the "launching pad" for worklet effectors. The Executor plugin collaborates with the effectors catalog, which for worklets takes the name and the form of a *Worklet Factory*. A Worklet Factory is composed of a repository of junctions, and a mechanism for searching, instantiating and configuring junctions. In the Worklet Factory, it is possible to associate semantic descriptors to junctions, which can be used to retrieve the appropriate effectors for the various process tasks: that is typically a three-ways match, which must take in account the semantics of the process task that requires the instantiation of an effector, the description of the junctions in the catalog, and the knowledge about the characteristics of the target components to be effected.

The level of sophistication used for implementing that match may greatly vary: from a simple lookup of the junction class name on the basis of some information attached to the process task that requires an effector, to the evaluation of architectural knowledge captured in the behavioral models available to the dynamic adaptation platform, to semantic reasoning on ontologies like those employed in Semantic Web contexts (e.g., OWL [161]).

Workflakes Version 1 was initially tightly coupled with the worklet technology, and incorporated specialized, ad hoc provisions for interacting with them. Later on, other options for integrating effectors were investigated. For example, SOAP-based messaging was used in a few experiments. Those alternative experiments led to a more clear-cut decoupling of the Executor plugin from actuation technologies, and

eventually to the design of the high-level actuation API as it is conceptualized in Section 3.2, which was implemented later on in Version 2 (see Section 4.2).

In Workflakes Version 1, there is another fundamental way in which the task processor interacts with and exploits worklets, which also represents the other major extension with respect to the underlying Cougaar technology. Workflakes Version 1 uses worklets also to load process definitions on the fly onto task processors, either with a pull or a push modality.

The main rationale for that is that the dynamic adaptation of target applications is likely to call for dynamically adaptable coordination and control. That contrasts with the Cougaar approach of hardcoding the process logic in a software program written within process execution plugins. In an effort to provide support for dynamic process loading and process evolution, as well as in an attempt to pull up somewhat the level of abstraction for the specification of processes, albeit in the absence of an abstract modeling language, a set of *shell plugins* were implemented, to substitute for the base Cougaar implementations of core plugins and LDM plugins. Also shell plugins in Workflakes – just like the base Cougaar plugin classes - are devoid of any hard-coded logic related to any particular process, but instead of supporting the implementation of process semantics via inheritance and specialization, they accommodate the insertion of process semantics at runtime from incoming worklets that carry appropriate code.

When a Workflakes task processor is launched with a certain configuration of plugins, they are all initially idle, and are merely indicative of the kinds of service and functionality that task processor is meant to offer within the overall distributed

Workflakes engine. Shell plugins can then be activated at any time by the injection of specialized *process definition junctions* containing the Java code that implements some process specification. Worklets dynamically deploy those junctions onto one or more task processors and their shell plugins. Only after such deployment, shell plugins acquire a definite behavior, and start taking part in the enactment of the process.

The main advantage of that scheme is that the process enactment infrastructure and the process specification remain more clearly independent. No modification of the core process enactment mechanisms is required anymore every time a new process must be defined; process programming, although still carried out in Java, remains confined to the development of a certain number of process definition junctions, and is conveniently supported by a relatively small, dedicated class library.

Furthermore, this process delivery mechanism is effective for a centralized as well as a more scalable, decentralized process enactment architecture. It may, for example, be used in the pull modality to incrementally retrieve process fragments from a process repository, only when requested to handle certain specific adaptations, or in the push modality for on-the-fly process evolution across a distributed Workflakes installation. A cohort of process definition worklets can be configured to distribute different process fragments to a number of task processors, as it is most convenient for execution: that can be used also for migration of process specifications at run time between nodes, which enables various forms of dynamic meta-adaptation of the process-based controller itself. In this scheme, since process are specified by

developing a small set of worklet junctions, the Worklet Factory which has the role of the effectors repository, doubles up as the process repository as well.

Finally, the interplay of shell plugins and process definition junctions – as implemented in Workflakes Version 1 - provides a limited set of uniform coding patterns, which guides and simplifies to some extent the work of defining processes.

4.2 Workflakes v. 2: employing a process modeling notation

Following the experience gained in implementing the first version of Workflakes, and the lessons learned in applying it to a number of case studies (see Sections 5.1, 5.3 and 5.4), a second implementation iteration was carried out. Besides the need fo a more generic and abstract actuation subsystem – as already mentioned – it was clear that the other major outstanding concern was the support of some high-level, expressive and formal process specification language, which could be accommodated by the Cougaar runtime. The rational was to significantly simplify the task of developing, testing and maintaining non-trivial dynamic adaptation processes in Workflakes. As a consequence, the majority of the effort in Version 2 was spent in designing and developing the integration of a process language in the Cougaar runtime. Little-JIL [19], developed at the University of Massachusetts at Amherst, was the language of choice for this experimental development. Some of the major characteristics of that language, which guided that choice, are listed below:

- Little-JIL has an explicit focus on agents coordination: therefore it naturally leans towards problem domains that – as discussed in Section 2.4 – carry a

number of similarities with respect to those faced by dynamic software adaptation.

- Little-JIL provides well formalized execution semantics for a rich set of process definition constructs. Among other things, it includes sophisticated support for exceptions and their handling, which are crucial for dynamic adaptation (to handle contingencies and implement alternatives, backtracking, compensations, etc.).
- It offers a high-level graphic language and editor.
- Processes are expressed according to a task decomposition hierarchy, which maps well to the chosen model for dynamic adaptation processes described in Section 3.1, as well as to the major constructs that have built-in support in the Cougaar runtime libraries.
- Process specifications are modular and support well the composition of fragments: a sub-process in the main process specification document can be represented simply by its parent task, while can be fully specified with all of its hierarchy of sub-tasks in a different document.
- Bindings for the data model can be included in and referenced from within the process diagram, which facilitates representing data that must be conveyed to effectors, as well as the effectors themselves.

All of the characteristics above indicated that providing support to processes defined with Little-JIL in the second release of Workflakes could represent a significant enhancement of the expressive capabilities of the system, and consequently of its usefulness. On the other hand, the major difficulty lied in being able to port in a

faithful way the rich set of process constructs made available in the Little-JIL language within the Cougaar runtime and its computational model. While the generic structure of workflows according to both approaches is a task decomposition hierarchy, some of the more sophisticated Little-JIL process constructs cannot be readily expressed in terms of the core workflow class libraries and API of Cougaar. Some noticeable constructs are outlined below (for a more complete overview refer to [19] and the language documentation at http://laser.cs.cumass.edu):

- Advanced sequencing modes for the workflow of sub-processes originating from a parent step. Besides classic sequential and parallel flows, the other modes supported are: *choice*, which non-deterministically selects one of the sub-steps for execution; and *try*, which tries in sequence all the sub-steps, until one is successfully executed.
- Pre-requisite sub-processes attached to any step, which are enacted before that step, and whose successful execution determines if that step can be enacted.
- Post-requisite sub-processes attached to any step, which are enacted immediately after the workflow of that step is finished, and whose successful execution determines if that step can itself be considered successful.
- Four different semantics for catching, handling and consuming exceptions: *continue,* which does not change the scheduling of the workflow of the step catching the exception in any way; *complete*, which forces the immediate completion of the step catching the exception; *rethrow*, which terminates the step catching the exception and passes the exception one level up in the task decomposition hierarchy to its parent step; and *restart*, which forces the step

catching the exception to start its workflow anew. Each of those exception handling modes can also be associated to a sub-process (*a handler step*) that is enacted just before the exception is consumed by the step that caught it.

- Cardinality of the transitions between steps, which determines how many instances of a given sub-step (default is 1) are to be enacted for the workflow of the parent step. Cardinality values can be constant values, or can be variables, linked to the size of a resource set passed from the parent step to its sub-steps.

Because of all of the above, a number of new, custom plugins had to be developed, to aid in the translation and execution of a Little-JIL process. Each of them takes care individually of certain Little-JIL constructs and specificities; all together, they cooperatively implement all the necessary capabilities for a Little-JIL enabled Cougaar task processor. The structure of a Little-JIL enabled task processor is displayed in Figure 13,. For the integration, Cougaar release 9.6 and Little-JIL 1.3 were employed.

In Workflakes Version 2, process specifications are codified directly in the Little-JIL graphic editor, thus setting the control flow and the data flow of artifacts through tasks, as well as the resource allocation requirements. Resources and artifacts, in accord to the data modeling method adopted in Little-JIL are defined as Java classes: those classes can be themselves object-oriented implementations of data and resources, or can represent wrappers for the management of data and resources (possibly legacy) that remain external to the process. In both cases, their definitions are included in the process specification document (also called a Little-JIL *diagram*).

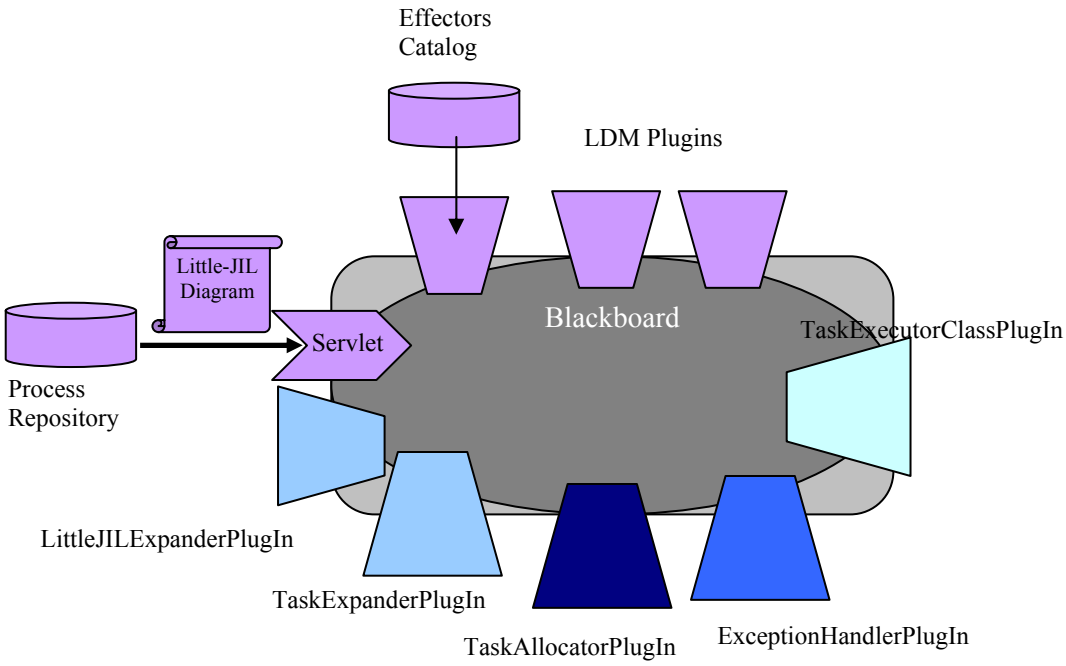

Figure 13: Representation of a task processor in Workflakes Version 2.

A process diagram can be output by the editor in two formats: as a serialized set of Java objects, or as an XML document. The serialized (binary) format is the most commonly used, since it allows to get the whole diagram in or out of the memory of a Java computer program very efficiently and reliably. Workflakes currently employs the serialized form; a diagram can be input to a task processor by using a specialized Servlet component that is available in the Cougaar framework, namely the *LittleJilLoaderServletComponent*. By using a servlet mechanism, Workflakes Version 2 necessarily leans towards the "push" modality for loading process specifications into the enactment engine. As an example, it could be the decision role of the dynamic adaptation platform that would select an appropriate diagram from the process repository and invoke the servlet to load that diagram into Workflakes.

With a high-level, formal description of the process that is developed outside and is then interpreted within of the process engine, and with the servlet mechanism mentioned above, most of the motivations for shell plugins and workflow definition junctions as conceived in Workflakes Version 1 are not cogent in Version 2. In fact, Version 2 plugins that represent process execution mechanisms in the engine are specializations of standards Cougaar plugins, not shell plugins. Shell plugins can still be used in Version 2 to enable the dynamic modification of the behavior of components in the engine devoted to other purposes: for instance, LDM plugins that are used for the exchange of data with external sources and sinks, or for the interaction with other elements in the dynamic adaptation platform, such as the decision facilities or the actuation subsystem.

Notice in fact that, to complete the information needed for running the dynamic adaptation process, data about effectors that are to be used in the course of the dynamic adaptation process must also be loaded to supplement the process diagram. That is accomplished through an LDM plugin that interfaces with the effectors repository, and can be done together with the process loading (in push mode), or later on, on a per-need basis (in pull mode). Data about effectors is captured by assets that subclass *ExecAgentAsset*. They provide explicit associations between certain leaf tasks in the dynamic adaptation process and certain effectors: those associations allow the process engine to *instantiate* or *recruit* specific effectors (for the definition of Instantiate, Recruit and other effector-handling primitives refer to the discussion on the actuation API in Section 3.2) when needed by the process as it is enacted.

A loaded process diagram is interpreted by the *LittleJILExpanderPlugIn*, which traverses its structure and creates Cougaar Task instances in the Blackboard for each Little-JIL step, and recursively for each of its sub-steps. That plugin sets constraints correctly for sequential or parallel execution, and creates other structures of Cougaar tasks, to capture appropriately certain specific constructs, such as cardinality, pre- and post-requisites, and the *try* and *choice* sequencing modes of Little-JIL.

While the LittleJILExpanderPlugIn is concerned with the translation of Little-JIL diagrams into an internal runtime representation, the *TaskExpanderPlugIn* has the main purpose of advancing the execution of the process by working on that representation. The TaskExpanderPlugIn evaluates the outstanding constraints of each non-atomic (parent) task that is put in the blackboard by the LittleJILExpanderPlugIn; in case all constraints are satisfied, it creates an expansion that effectively initiates the enactment of that task and contains some of its subtasks In accordance with the semantics of the corresponding step, the subtasks to be inserted in the Expansion change (e.g., it initially includes only the first subtask for a sequential task, or all of the subtasks for a parallel task, and so on). Thus, the TaskExpanderPlugIn incrementally creates the entire task decomposition hierarchy of the process as it progresses, from its root down to its leaf tasks.

Leaf tasks – i.e., the loci in which the coordination and computational semantics of a dynamic adaptation process come together - are managed by the *TaskAllocatorPlugIn*. Its main purpose is to bind to each posted leaf task any effectors or other computational entities (such as helper functions) that need to be executed at that point in the process; it accomplishes that by creating an appropriate

instance of a Cougaar Allocation. A completed allocation includes an ExecAgentAsset as an assigned resource, and signifies that the leaf task is ready to carry out its side effects on the target system through the actuation subsystem.

Within the process engine, the plugin in charge of managing generic effectors and their execution, and therefore of implementing the actuation subsystem according to the design of a process-based controller in Section 3.2 is the *TaskExecutorClassPlugIn*.

Effectors in Workflakes Version 2 can be worklets, as originally in Version 1, or other computational facilities that can be exploited to actuate the target system. All kinds of effectors in Version 2 are uniformly wrapped by and accessed through a Java interface that is named *ExecutableTask*. Classes implementing that interface are assumed to encapsulate the internal mechanics, functionality and logic of some effector, and provide a simple way to develop effectors of various types. The class name of the ExecutableTask specialization provided by an effector is part of the information stored in an ExecAgentAsset. In all experiments, effectors have been coded in Java; to integrate non-Java effectors, one can rely on the cross-platform interoperability facilities made available by the Java framework, such as the Java Native Interface (JNI) [162], and subclass ExecutableTask to expose those facilities.

The TaskExecutorClassPlugIn takes control of ExecutableTask instances by looking at the allocations published in the Blackboard by the TaskAllocatorPlugIn; it then decides whether to recruit an existing effector of the kind specified in the ExecAgentAsset of the allocation, or – if needed - creating a new instance of it; then, the TaskExecutorClassPlugIn activates the effector by invoking its operation through

the ExecutableTask interface. The TaskExecutorClassPlugIn also provides the methods that allow to *relay* back to the process the return data generated by the effector's execution, which must always include a success/failure flag.

Finally, a Little-JIL-enabled task processor includes the *ExceptionHandlerPlugIn*, which is in charge to implement the logic necessary to comply with the various kinds of Little-JIL exceptions. The exception mechanism can be used to describe internal as well as external contingencies that can impact the dynamic adaptation process. Typical external contingencies exceptions are thrown by the TaskExecutorClassPlugIn when a task fails, or by the TaskAllocatorPlugIn when some piece of data or some resource cannot be found and bound to a task as needed. Internal contingencies exceptions are instead typically thrown as a part of the side effect of the computational actions carried out by effectors.

Since in the Little-JIL language the handling of exceptions can be defined in a variety of ways, each with its own semantics, the language provides sophisticated support to the kinds of exception handling that are necessary for dynamic adaptation processes, and in particular to compensations. To properly support compensations, an exception handling semantics that is likely useful in many cases is the *rethrow*, coupled with a handler step. That allows to modularize compensations: for each side effect that does not apply anymore as the consequence of the exception, a process fragments in charge of undoing that side effect can be defined. Furthermore, rethrowing the exception allows to traverse the various levels of the task decomposition hierarchy back up until all unwanted side effects are compensated.

5 Experiments

Several experiments of different scale and in varied application domains have been carried out to validate the KX externalized platform for dynamic software adaptation, and the Workflakes process-based coordinator. Case studies to date have considered distributed target systems that range from Internet-wide information systems, to Business-to-Customer (B2C) marketplaces, to multi-channel instant messaging applications, to collaborative multimedia systems. The principal traits of those case studies are presented hereby, together with the most significant results and lessons that have been drawn from them.

It is noticeable that while most of the case studies have been implemented and evaluated by components of the same research team that developed KX and Workflakes, others have been carried out by external organizations, in the context of multi-party collaborative research projects[7]. One such case study is reported in this document - necessarily with less detail than the others - since it helps to highlight the usability and usefulness of this work, although in the state of a research prototype, in contexts that were not known or under the full control of the prototype developers.

Finally, notice that among the case studies presented below, the majority was implemented by using the earlier version 1 of Workflakes. Only the AI^{2}TV case study of Section 5.2 has been carried out in such a time frame that it could exploit version 2 of the prototype.

[7] Specifically, the case study discussed in Section 5.2 was developed within the EURESCOM project P-1108 (OLIVES).

5.1 Instant messaging service

Background

Figure 14 represents the architecture of a J2EE-based multi-channel instant messaging service for personal communication (IM in the remainder), which is currently offered on a 24/7/365 basis to tens of thousands of customers through a variety of channels, such as the Web, PC-based Internet chat, Short Message Service (SMS), Wireless Application Protocol, etc.

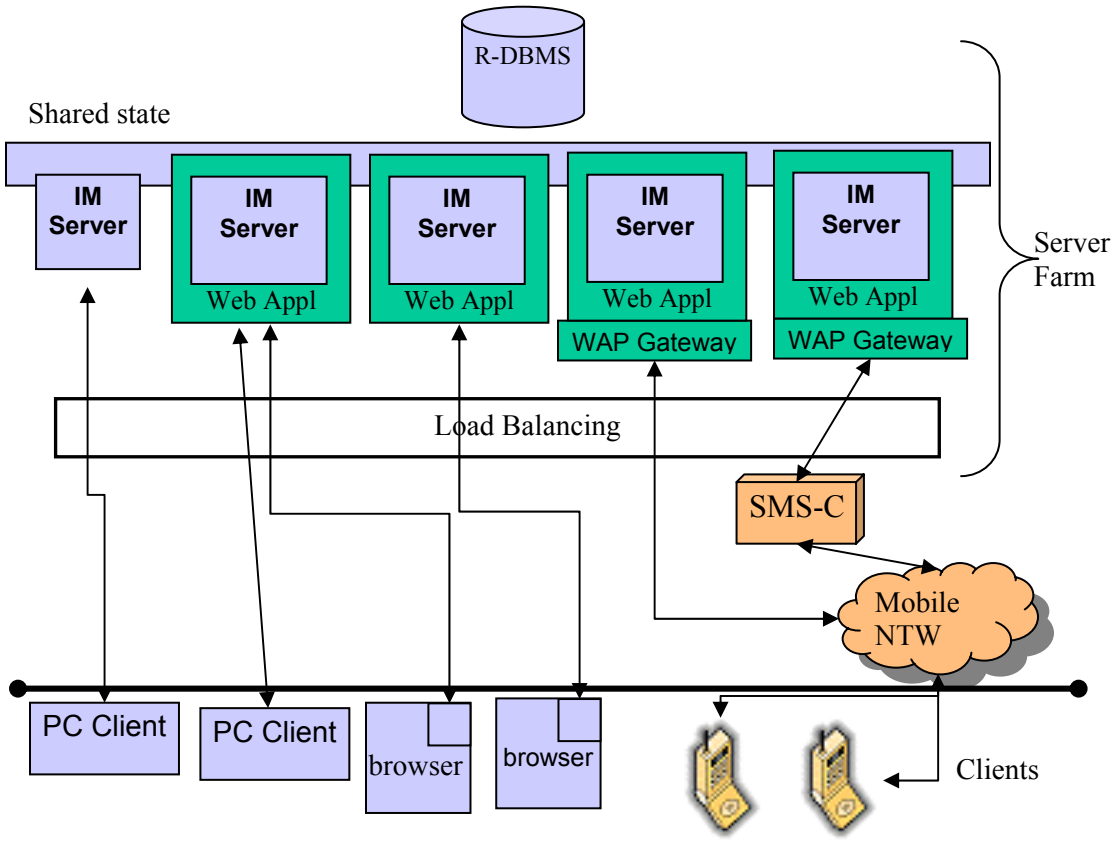

Figure 14: The IM service architecture

The service runtime environment consists of a typical three-tiered server farm: a commercial software provides a common load balancing front end to all end-users

and redirects all client traffic to several replicas of the IM components, which are installed and operate on a set of middle tier hosts. The various replicas of the IM server all share a relational database and a common runtime state repository, which make up the backend tier, and allow replicas to operate in an undifferentiated way as a collective service. Some of the IM servers are wrapped by Web applications running on commercial J2EE application servers; others may provide additional facilities, which handle access to the service through specific channels, such as SMS or WAP, and interoperate with third-party components and resources, e.g., gateways to the cell phone communication network.

Case study description

The case study addressed two main goals: enhancing the Quality of Service (QoS) perceived by end users, and facilitating service management by the staff in charge of supporting such a complex distributed application.

With respect to QoS, the requirements of the case study focused on resolving existing load and availability problems by automating service scalability, as well as reconfiguring promptly and opportunely service parameters related to serving client requests efficiently. As for service management, the requirements focused on the automation of the deployment, bootstrapping and configuration of the various service components, the continuous monitoring of those components and their interactions, and the support for “hot” service staging via automated rollout of new versions and patches without service interruption. Together, those requirements address configuration, optimization and healing aspects.

All of those requirements are captured and addressed within a dynamic adaptation process automated by Workflakes. This process requires – among the logic and data loaded at startup onto the Workflakes engine – explicit knowledge about the service architecture and the runtime environment of the server farm. That knowledge is currently codified in a proprietary way: it is expressed as data that is input into Workflakes at the beginning of its operation and is maintained as a set of assets in its Blackboard.

At startup, Workflakes is given a configuration of service components that must be instantiated. Workflakes selects some hosts in the server farm for this initial deployment and sends them Worklets to execute bootstrapping code for the IM components and configure the servers with all the necessary parameters (such as the JDBC connection handle to the DBMS, the port numbers for connections by clients and other IM servers, etc.). Notice that not only the configuration information, but also the executable code of the IM server is deployed and loaded on demand, taking advantage of a code-pulling feature of the Worklets agent platform. (This approach is also followed in Software Dock [36]).

Depending on the types of the components, the deployment sub-process may change. For example, a normal IM server can be instantiated and configured by a single Worklet in one step. Web-based IM servers are notably more complex to startup and configure, since that requires first of all the spawning of a new instance of the application server, then the instantiation and parameterization of the residing Web application with respect to the hosting application server, and finally its configuration and activation as an IM component.

When a Worklet starts up an IM server, sensors are simultaneously activated to track the server's instantiation and initialization. When the instantiation is successful, the process must dispatch other Worklets onto the load balancer of the server farm, which accepts traffic for the IM service, to instruct it to route it to the right host address and port for the new server. In the event of an unsuccessful initialization, instead, the likely cause is inferred by the gauge layer of the dynamic adaptation platform and reported back to the process (and also to a dashboard GUI implemented specifically for this case study). Depending on the cause of that contingency and the stage of the deployment process, Workflakes may react in different ways: it may decide to try to bootstrap an IM server on the same host again, or on another available host, or it could skip that portion of the configuration or even abort the whole process.

Following the initial bootstrapping phase, and after the intended service configuration is in place, Workflakes takes a fully reactive role, while the probing and gauging layers of the platform start monitoring and analyzing the dynamics of service usage. Sensors and gauges are activated for a number of purposes: to capture the logging in and out of clients onto the servers, count the number of users logged on at each server, signal the raising of exceptions, monitor service latency and the number of service requests queued by the Web applications, etc.

This case study is particularly concerned with load and responsiveness. Each IM server has an associated load threshold, which is best expressed in terms of the number of concurrently active clients, in relationship with the memory resources of the host. When that threshold is passed, Workflakes reacts by trying to scale up the service: it selects from the system model some unused machine available in the server

farm, and repeats the server bootstrapping process fragment on that machine, providing a new server replica for handling the extra load, thus achieving enhanced reliability and performance of the overall service.

For Web IM components, it was also possible to reach a finer level of adaptation, exploiting the management capabilities built into the J2EE application server used for the IM web applications, that is, BEA WebLogic server [163]. They are JMX Management Beans, some of which could be integrated smoothly within both the sensors and effectors layers of our platform. By taking advantage of those Management Beans, Workflakes can decide to intervene also in response to variations in the size of the queue of pending requests, and manipulate the details of the threading model of the Web IM application in response. That optimizes the degree of parallelism in processing client requests, and improves responsiveness.

The case study also experimented with service staging and evolution scenarios, aiming at complete automation and minimal service disruption. It turns out that a service evolution campaign can be supported by Workflakes with relatively minor changes to the service bootstrapping process described above. The staging process includes specific fragments that gradually withdraw from the load balancer outdated server instances (thus disallowing new traffic to be assigned to them), and shut them down when traffic is absent or minimal, while another process fragment coordinately starts up other server instances with the new code release, registers them on the load balancer, and thus makes them gradually available to users.

Case study results

As part of the work carried out on the IM case study, some results originating from the experiments described above have been evaluated.

A set of quantitative results were derived from running and observing the adapted IM system in lab conditions, using manual and automated traffic simulations. The automated simulations used the same tools and traffic profiles that were employed by the service developers for their stress and quality assurance testing, which simulated traffic spikes interleaved with periods of steady request levels to the IM servers.

Following from the main goal stated for the experiment, the results refer primarily to the levels of automated support provided to the maintenance and management activities carried out onto the IM service on the field. Also some measures about the development effort necessary to implement the case study were taken. The most significant quantitative results are reported below:

- Substantially reduced effort for the deployment, configuration and evolution (staging) of an IM service in the field. Current manual procedures (using Unix shell scripts and assuming DBMS and application servers pre-installed in the server farm) can take ½ to 1 person-day, with expert personnel present locally. With KX and Workflakes, that is reduced to 1-2 minutes from a remote location.
- Reduced monitoring and maintenance effort necessary to ensure the health of the running service. A system administrator was previously needed on-site 24/7/365, with a secondary support team of experts available on call. KX with Workflakes completely automates the monitoring of a set of major service parameters, as well as the counter-measures to be taken for a set of well-known critical conditions.

Additionally, since the process of scaling the system is completely automated, there is no risk of under- or over-provisioning, which represents another improvement for the service management and administration.

- Reduced reaction times and improved availability and reliability: for example, KX/Workflakes recognizes the passing of the IM load threshold in 1-2 seconds and takes approximately 40 seconds to put in place an additional server replica. Previously there was no direct overload detection: the sysadmin in charge was supposed to check the number of concurrent users from the logs and to manually start up an additional server when necessary. That is clearly error-prone and could endanger service availability, in which case resource shortage would crash overloaded servers.
- Manageable coding complexity: by exploiting the facilities provided by KX, sensors, gauges and effectors are derived from generic code instrumentation templates that are then customized with situational logic. This results in rather compact code: 15 Java code lines for sensors on average, usually less than 100 for effectors. The total code written for this specific case study on top of the generic dynamic adaptation facilities provided by the KX/Workflakes infrastructure was slightly above 2000 lines of Java and XML code.

As a conclusion that can be drawn from the quantitative measurements above, employing KX and Workflakes in this case study has shown higher levels of automation, flexibility and reliability with respect to the management of the target service and its QoS, when compared with more traditional labor-intensive management and administration practices. Those results, when put in the frame of the

autonomic capability model, contribute to raise the system management practices for the IM service from level 2 (mostly manual management, supported by some monitoring tools) to level 4 (fully automated management, taken care of by the adaptation closed control loop).

Additional interesting lessons originating from this case study include the following qualitative considerations:

- Impact on service development: The team that carried out the case study positioned themselves past the end of the development phase of the project life cycle and just prior to the deployment phase. No requirements were conveyed back to the separate team in charge of furthering the development of the target IM service. The software for the IM service was hence treated as a complete legacy, although a legacy for which all the specifications, software artifacts and accumulated project knowledge happened to be available, and could be shared between the case study team and the service development team. Notice that also a different kind of legacy applies in the case study: the application server and the load balancer are commercial software products (WebLogic Server by BEA [163] and Network Dispatcher by IBM [164], respectively), which however provide sufficient APIs for carrying out their monitoring and actuation. Even within those limitations, it was possible to satisfy all the requirements of the case study.
- Impact of the behavioral model: the amount of effort to analyze the model of the target system and its behavior for dynamic adaptation purposes constituted by itself about one third of the overall effort spent for the case study (46 out of about 140 person / days), that is, to develop, test and evaluate the dynamic adaptation

solution for the IM service according to requirements and on top of the KX platform. Furthermore, more than 10% of the software that was written was intended to capture architectural information, relationships and inferences with respect to the target system, and represent them to KX and Workflakes as proprietary models that could inform the work of the platform. That constitutes evidence of the strong dependency of dynamic adaptation on the ability to capture, describe and expose in an abstract and machine-readable way knowledge about a number of aspects relevant to the target architecture, and provides empirical support and motivation to explore integration of dynamic adaptation tools and platforms with formal models.

- Integrated automated management: here is where the benefit of a full-fledged process engine becomes most evident. Traditional application management is concerned with reporting warnings, alarms and other information to some knowledgeable human operator who can recognize situations as they occur, and take actions as needed. The amount of guidance and automation on the part of the management platform then may be very limited. Our approach offers instead a high level of guidance, coordination and automation to enforce what is a complex but many times largely repeatable and codifiable process.

5.2 AI²TV

Background

AI²TV stands for "Adaptive Internet Interactive Team Video". AI²TV is a project that aims at supporting multimedia-assisted distributed team work. It includes a subsystem

devoted to the provision over wide-area networks of multimedia content relevant to the work carried out by the team, such as audio/video recordings of group discussions and decisions, informational and educational events, etc. The application domain of election for AI^{2}TV is long-distance education: the focus is on enabling the remotization of attendance and review of class lectures, group study, and collaborative team assignments, such as software development projects. AI^{2}TV differs from many existing infrastructures for providing educational content in a networked context because of its explicit focus on supporting the team work aspects of on-line education.

A number of speculative, design and technological challenges underlie the main AI^{2}TV ideas. The issues that are most relevant to the work presented in this document lie mainly in between multimedia and Computer-Supported Collaborative Work (CSCW), in particular how dynamic software adaptation can aid multimedia-enabled computerized CSCW tools to meet their goal of efficiently and effectively supporting collaborative work practices.

One of the most compelling requirements for AI^{2}TV is the support for the synchronized watching of a streamed video by all members of a widely distributed team, independently of their different equipment and networking capabilities, including support for video operations like fast forward, rewind, seek, etc.

Imagine a scenario in which the team decides to review together a portion of a recorded lecture, in order to solve some difficulty in their project assignment or to clarify some notion. Given that team members can be dispersed over the Internet, and may enjoy very diverse connectivity, ranging for example from 28.8k modem, to

DSL, to cable, to T1 lines, the multimedia content they must use for their review session is to be delivered over heterogeneous Internet links to heterogeneous platforms. Moreover, in such a setup the communication and computing resources available to each user may widely and quickly vary in the course of the team work session. In contrast with those potential difficulties, the collaboration can be effective only in case the fruition of the streamed video clip remains well-synchronized, so that users, who are enabled by their clients to discuss the material among each other as they see it, have a natural group experience (like they were co-located in class, or watching a tape together sitting in a study room) and do not incur in misunderstandings, waste of time, or other serious inconveniences during their review session.

This kind of collaborative video sharing poses a twofold problem: first of all, it is mandatory to keep all users synchronized with respect to the content they are supposed to see at any moment during play time; furthermore, it is important to provide each individual user with a viewing experience that is adequate with respect to the user's available resources, which may also vary during the course of the video.

The solution proposed is based on the one hand on offering multimedia content in multiple versions, with different levels of semantic compression, achieved by employing a semantic summarization package separately developed at Columbia University [184], and on the other hand on using the process / workflow technology made available by Workflakes to dynamically adapt content provision.

Dynamic adaptation in this case is directed at modifying a combination of server and client configurations, data fetching and buffering strategies and video playing

schemes, to accommodate and harmonize varying latencies, throughputs, client processing power, and server work loads. All of that – as will become evident as the implementation of the case study is described - must be completed by Workflakes within narrow time boundaries (in the order of seconds or less), given the soft real-time nature of the application and the kind of adaptation that must be effected.

The one described above is the first possible application of process-based dynamic adaptation in the AI^{2}TV system is termed the *short-term* or *client synchronization workflow*, to distinguish it from the other possible applications, which are introduced below.

Another context in AI^{2}TV, in which process / workflow technology plays a significant part is the organization of the work of the team as well as its individual members, in accord with an agenda containing a schedule of group events (e.g., virtual study "meetings") and work deadlines. In AI^{2}TV, a workflow is used to model and guide the activities of the distributed team along that planned schedule. The typology of that workflow is in general that of a classic human-oriented process, whose stakeholders are persons and whose goal is to facilitate and guide the collaboration among those persons; furthermore, the workflow is likely to span its activities across a relatively long term, i.e., in terms of weeks, days, or hours in the most demanding cases. That workflow is therefore called the *long-term scheduling workflow*.

Given those characteristics, that kind of workflow is seemingly not concerned with any dynamic software adaptation issues. However, there are peculiarities intrinsic to the presence and use of multimedia content in the workflow that demand for the

introduction of dynamic adaptation aspects, which become intertwined with the coordination of team activities.

Multimedia content must be treated by the workflow as both a new type of artifact and an additional kind of resource, albeit an expensive one, whose use must be carefully organized and managed. One thing this workflow must do is to plan and orchestrate the distribution of such multimedia artifacts to each individual team member in a timely fashion. Ideally, all needed artifacts would be entirely transferred to all clients in advance, before a planned event begins. That would avoid the need for streaming any information on the fly, and would thus largely circumvent the problems that require the intervention of the short-term synchronization workflow, at least for planned-ahead joint work events (for virtual meetings that are set up and initiated with no or little advance notice, the client synchronization workflow remains completely relevant).

In the real world, the typical situation is likely to be somewhere in between the two extremes above. Given that, and also in the view of the high variability of the factors that may influence or hinder the ability to make available in advance the necessary artifacts, such as connectivity, servers, storage space in the clients, CPU load on clients, and more, there is a need for AI^{2}TV to adapt on the fly even in the long-term context. That way it becomes possible to overcome connectivity and capability idiosyncrasies and maximize the amount of data that can be buffered in advance and that is hence immediately available to clients at the beginning of a group session. That can be resolved with a combination of content pre-fetching and caching techniques that are orchestrated via a *pre-fetching workflow*, which kicks in as part of

the long-term scheduling workflow. The pre-fetching workflow must also graciously turn control over to the synchronization workflow whenever a group session begins, for keeping in check and adapting the synchronized delivery and presentation of the material across clients.

Besides the long- and the short- term, there is also a *medium-term* option for dynamically adapting the provision of multimedia content in AI^{2}TV. That can occur whenever, during the synchronized fruition of some multimedia stream, the group decides to pause or interrupt the viewing. That opens a window of opportunity for loading additional material in clients' buffers, taking advantage of the period in which network connections to the video server remain idle. The logic of this *opportunistic pre-fetching workflow* is akin to that of the long-term pre-fetching workflow, but it must operate within a time frame that is closer to that of the client synchronization workflow.

Case study description

The implementation of the AI^{2}TV case study that is reported here is concerned only with the short-term client synchronization aspects. The other dynamic adaptation options related to content pre-fetching in the medium and long term, and how they relate to and interact with the short-term workflow, are the subjects of future work. The experimental work described here has focused on client synchronization since, because of the soft real-time constraints inherent to adaptive multimedia provisioning, it represents a particularly interesting and demanding field of application for dynamic software adaptation in general, and for its process-based orchestration in particular, as previously discussed in Section 3.3.

Notice also that in this particular case study Workflakes is employed on its own, without the other elements of the KX platform. The design of the control loop superimposed onto the AI^{2}TV system still follows the conceptual architecture for externalized dynamic adaptation of Figure 3. However, some simplifications have been used, for expediency of design: referring to the design schematic of the AI^{2}TV system shown in Figure 15, customized sensors in the clients communicate with handcrafted gauges that are embedded together with the coordination engine. That simplification allows to minimize the communication latency between the various elements of the dynamic adaptation platform, which in turn allows to precisely determine the amount of time spent in the Workflakes controller, and its contribution to the timeliness of the adaptations (which is one of the intended results of the experiments with AI^{2}TV).

Besides the controller and the underlying event-based middleware (i.e., Siena [156]) used for distributed communications within the control loop, Figure 15 shows the video server and the AI^{2}TV clients.

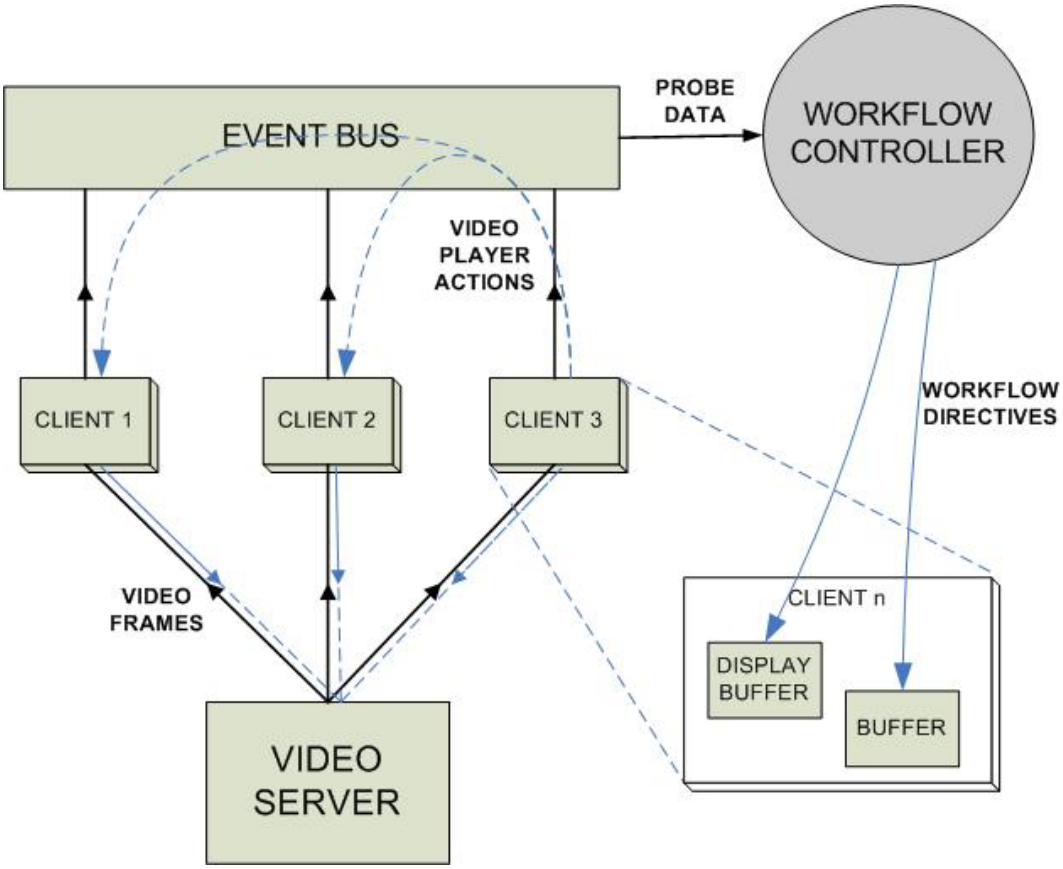

Figure 15: The AI^{2}TV System.

The video server makes available the educational video content to AI^{2}TV clients. Such content has the form of a hierarchy of video versions produced with the semantic summarization tool mentioned previously. That tool operates on MPEG format videos and outputs sequences of JPG frames. Its semantic compression algorithm profiles video frames within a sliding time window and selects key frames that have the most semantic information. By increasing the size of the window, a key frame will represent a larger time slice, which means that a larger window size will produce less key frames as compared to a smaller window size setting, effectively increasing the level of semantic compression. By running the tool multiple times with settings for different compression levels, several sets of frames are produced, which

are indexed by a frame index file. The task of the video server is to provide to clients download access to the frames and the index file over HTTP.

Notice that the various quality level produced that way are characterized by their different frame rates. Notice also that the semantic compression algorithm produces effectively a random distribution of key frames, hence the video produced by the package plays back at a variable frame rate.

Clients participating in the same group are the subjects of the short-term adaptation. A group is limited in number, since user teams are assumed to be composed of 2-5 users at least, and 10-12 at most. The task of each client is to acquire video frames, display them at the correct time, and provide a set of basic video functions. From a functional design perspective, the client is composed of three major modules: a time controller, a video buffer and manager for fetching and storing downloaded frames, and a video display.

The time controller's task is to ensure that a common video clock is maintained across clients. It relies on NTP [185] to synchronize the system's software clock therefore ensuring a common time base for the group, which each client can reference. Using this foundation, the task of each client of displaying the client's needed frame at the correct time is simplified. Since all the clients refer to the same time base, then at any time all the clients are showing a semantically equivalent frame, unless some clients do not have it available at any quality level.

The video buffer and manager constitute a downloading daemon that continuously downloads frames at a certain quality level. It keeps a hash of the available frames and a count of the current reserve frames (frames buffered) for each quality level. The

buffer manager also includes an actuation interface hook that enables the controller to adjust the current downloading quality level.

The video display renders the frames into a window and provides a user interface with controls for play, pause, goto, and stop. When any participant initiates one of those actions, all the other group members receive the same command as a time-stamped event. Referring to the common time base, all the video players can take action in a synchronized way so that results are consistent. Furthermore, the video display knows which frame to render at any time, by using the current video time and the current display quality level to retrieve into the frame index the representative frame. Before trying to render that frame, the video display asks the video buffer manager if it is available. The video display also includes an actuation interface that enables the autonomic controller, to adjust the current display quality level.

Given how AI^{2}TV clients are developed, a client at each given moment in time can be in two states with respect to synchronization,: it is either in sync, i.e., displaying the correct frame at some compression level with respect to the playing time of the video clip, or is lagging behind, i.e., missing in its buffer the correct frame that it should be displaying at that moment.

Clients are also equipped with sensors that have been developed specifically for this case study. Those sensors report data such as video display quality level, the buffer quality level, the buffer reserve frames, the currently displayed frame and the current bandwidth. A gauge samples periodically (e.g., every second) that information from each client, and stores it into *buckets*, similarly to [186]: a full bucket is a complete

sample that represents a snapshot that reports the state of the whole client group at some moment in time; that sample is then transferred to the Workflakes controller. Workflakes uses each incoming sample as the basis for decision making: the data in the sample is evaluated by a set of helper functions that compute whether the users, albeit at different levels of semantic compression, are viewing equivalent informational content, in sync with the playing time of the video clip. They also estimate whether some clients, although in sync, may risk to lag behind in the future, given the current available resources and the current quality level, or – at the contrary - whether they have abundant resources that could be better exploited for enhancing the viewing experience. On that basis, decisions are taken about which clients must be adapted and in what way, and triggers for the adaptation process are produced. Therefore, in case the multimedia clients of some users in the group are at risk of "lagging behind" with respect to others, their buffer managers are instructed by the synchronization workflow to downgrade on the fly their content fetching to a level that is more compressed, and to start displaying from a certain frame within that level; that implies that in the most critical cases certain informational content may also be skipped. This trade-off of quality for timeliness is acceptable, given that, in the context and for the purposes of the AI^{2}TV system, synchronization is arguably a more important quality factor for the user experience than content presentation, or even content integrity. Conversely, whenever for the helper functions a client results rather lightly loaded and able to keep pace without problems, the synchronization workflow may instruct it to upgrade its content fetching level and/or display level to a higher-quality, thus enhancing the user experience also with respect to content merits.

All of the above is repeated every time a full sample is produced by the gauges, therefore needs to be computed, coordinated and effected in a fraction of that sampling time. For that reason, the synchronization process cannot be over-complicated, otherwise the time spent to execute the coordination process alone could become excessive. The key of this case study from the point of view of the process enactment facilities that fulfill the coordination role is to be able to orchestrate the necessary adaptation as efficiently as possible, and to repeat the same adaptation process frequently, rather than being able to represent and enact a very involved and sophisticated coordination process. An implied challenge is to produce with a simple process the rather complex effect of synchronizing the group of dispersed AI^2TV clients, while at the same time optimizing the viewing experience for each of them.

The process of this case study is shown in Figure 16 as a Little-JIL diagram.

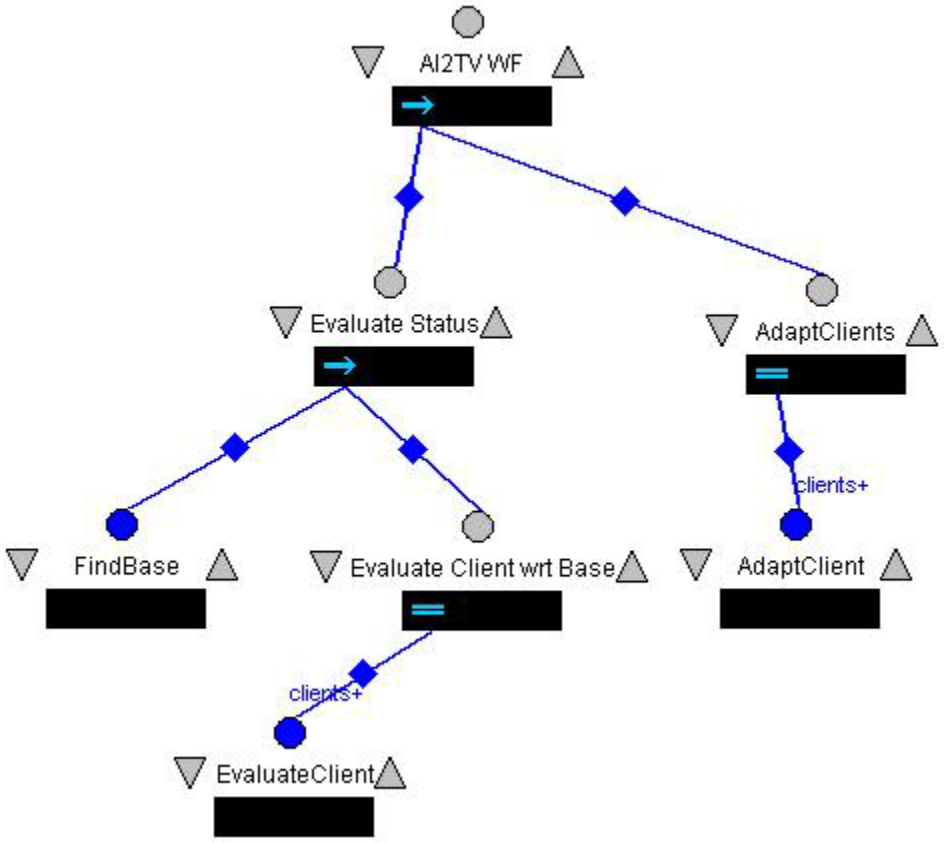

Figure 16: The AI^{2}TV process in Little-JIL.

The task decomposition hierarchy employed for that dynamic adaptation is structurally simple. One reason is that it does not need to account for particular contingencies, either internal or external; any failure in adapting some client – although may result in temporary degradation of the behavior of the group - can be recovered the next time the process is executed. Thus, the process does not need to provide specific flows for handling exceptional conditions or implementing compensations. One subtlety is how the parallel enactment of multiple instances of the *EvaluateClient* and *AdaptClient* tasks is determined dynamically, depending on the state and number of clients in the group, as signified by the *clients+* label on the transition arcs. It is noticeable how the native semantics of the Little-JIL language

supports that aspect in a very simple way, by allowing to bind the arc connecting a parallel step with some sub-step to a resource collection (called *clients* in this case, and representing the group of AI^{2}TV clients). The cardinality of that resource collection is evaluated on the fly when expanding the parallel step, and determines the number of instances of the sub-step that are enacted.

Case study results

The results of the experiments described above have been evaluated as part of the work carried out on the AI^{2}TV case study. The collection and evaluation of the results of the AI^{2}TV case study is aimed at verifying two aspects related to the client synchronization workflow:

- that process-based coordination of dynamic adaptation can be a suitable approach for target systems that have soft real-time constraints, even when those constraints are demanding as those of distributed multimedia systems;
- that when the coordination is correctly enacted within those time boundaries, an externalized dynamic adaptation platform superimposed on a soft real-time target is in fact able to enforce the desired behavior of that target system and significantly improve its quality of service.

The nature of the dynamic adaptation application in this case study enabled the collection and analysis of a wealth of quantitative data, measuring those two aspects during a number of AI^{2}TV trail runs. The trail runs involved teams having from 1 to 5, and as many as 10 participating clients, with wide variations in networking resources assigned to the link between each client and the video server.

With respect to the first aspect, i.e., making sure that the client synchronization workflow performs within the time boundaries required, that is, with a turn-around time that is significantly less than the sample time of the gauge that feeds the workflow, timestamps have been taken on the beginning and end of all the tasks in the workflow. As a result, the diagram in Figure 17 shows the average total execution time for the workflow, in trail runs involving 1, 2, 3, 4, 5, and 10 clients in the same group. The execution time displayed includes not only the time for the enactment of the process within the Workflakes engine, but also the time spent within the computational helper functions employed for decision-making, and invoked at different junctures in the orchestration process.

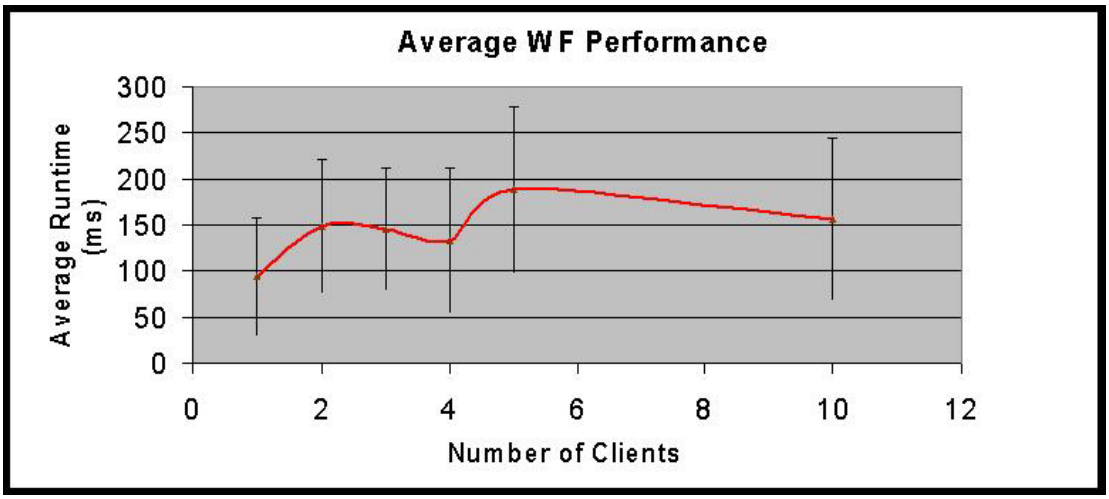

Figure 17: AI²TV - execution time of the adaptation process.

The data suggests that the execution time of the client synchronization workflow gauge occurs quickly enough to correct any clients that may be drifting out of sync in a prompt manner, and is sufficiently short to accommodate a sampling frequency of 1 second or less. That frequency seems adequate for a fine-grained control of group synchronization, at least in an educational context, in which images tend to have a

relatively low change rate. Therefore, it is possible to state that the experimental data is complaint with the timeliness requirements of the AI^{2}TV group video viewing.

Verifying whether dynamic adaptation improves significantly the quality of the target system was a more complicated affair, since the elaboration of an appropriate reference model of the QoS parameters relevant for the case study, with the corresponding metrics, was needed.

In [74], a survey and taxonomy of approaches to adapting Internet-wide video multicast is proposed. Although AI^{2}TV currently does not employ network multicasting protocols to deliver content to a team (it resorts instead to what is defined in [74] as *multiple-unicast*), the classification of approaches proposed in that survey is still helpful to characterize the kind of dynamic adaptation exerted in the AITV case study. Such classification is *layered adaptation*, since the case study couples *multi-rate* video delivery with *end-to-end* adaptation. The multiple levels of semantic compression of the content source in AI^{2}TV can be seen as a form of *cumulative layering* [75], also known as *scalable coding* [76]: those approaches to compression provide multiple versions of content, codified with different encodings of incremental quality levels (in our case with different frame rates), among which clients can choose. Moreover, the adaptation requires the monitoring and effecting of only the *end points of the communication*, without influencing in any way the intermediate nodes of the transport network and their behavior; finally, the adaptation mechanism according to which AI^{2}TV clients move up and down the hierarchy of compression levels can be characterized as *receiver-driven* since it depends from the monitoring of the clients' state.

Experimentation on adaptive video multicast is still in its infancy; moreover, layered adaptation schemes in particular have been rarely used to date (although they look promising in the near term, as research on scalable coding is becoming mature). Even more importantly, the principal quality factors to be evaluated are not consistent with the purposed of the AI^{2}TV case study: while the major concern of multicasting is achieving optimal viewing experience for the users, coupled with fairness in the data delivery to all of the various transmission end points, in AI^{2}TV the individual viewing experience must be reconciled instead with inter-client synchronization.

Because of the lack of standard evaluation procedures and data sets, and the lack of consistency in the quality factors to be measured, while it is possible to evaluate the effects of Workflakes in the AI^{2}TV case study with respect to a situation where no dynamic adaptation is applied, it would be hard to compare the benefits that can be observed in that case study with those achieved in equivalent experiments that use other approaches.

The evaluation of the AI^{2}TV case study, considers two different aspects: synchrony and Quality of Service (specifically, frame rate, because of the nature of the compression scheme adopted). That evaluation is carried out in a comparative way, with respect to a situation against which the performance of the dynamic adaptation approach can be consistently compared. To that end, a *baseline client* is used, whose quality level is set at the beginning of the video and not changed thereafter. To define the baseline client, a parameter that describes the average bandwidth per level is computed, by summing the total size in bytes of all frames produced at a certain compression level and dividing by the total video time. This value provides the

bandwidth needed on average for the buffer manager to download the next frame on time at that level. We provide the baseline client with the needed bandwidth for its chosen level by using a bandwidth throttling tool [187] that adjusts the bandwidth of the link to the video server. Notice that using the average as the baseline does not account for changes in the video frame rate and fluctuations in network bandwidth, which are situations in which adaptive control is supposed to make a difference.

When carrying out the evaluation, each controller-assisted client is assigned an initial level in the compression hierarchy and the same bandwidth as the baseline client for that hierarchy level. At the end of each experiment, we record any differences, with respect to synchrony and frame rate, between the adaptation of the clients' behavior on the part of Workflakes, and the behavior of the baseline client.

To evaluate synchrony, clients log at periodic time intervals the frame currently being displayed. This procedure effectively takes a snapshot of the system. This evaluation proceeds by checking whether the frame being displayed at a certain time corresponds to one of the valid frames at that time, on any arbitrary level according to the layered compression scheme. Arbitrary levels are allowed, because the semantic compression algorithm ensures that all frames at different levels for a certain time will contain the same semantic information if the semantic windows overlap. The system is then scored, by summing the number of clients not showing an acceptable frame and normalizing over the total number of clients in the group: a score of 0 indicates a synchronized group.

Our experiments for the evaluation of synchronization initially involved groups of clients that were set to begin playing a test video at different levels in the

compression hierarchy, and were assigned the corresponding baseline bandwidth. In those experiments, the results show a total score of 0 for all trials, with as well as without the supervision of Workflakes. Also, no frames were missed. This result demonstrates that the chosen baseline combinations of compression levels and throttled bandwidths do not push the clients beyond their bandwidth resource capacity, notwithstanding the variations in the frame rate and/or occasional fluctuations in the actual bandwidth of the clients,

We also ran a different set of experiments related to synchrony, in which the clients in the group were assigned more casually selected levels of starting bandwidths. This casual selection is representative of some real world situations, in which users must choose a desired frame rate to receive multimedia streams (typically, about the nominal bandwidth offered by their service provider) which may however differ considerably from the bandwidth actually available on that connection. We ran this set of experiments first without the aid of the controller and then with it. In the former case, clients with insufficient bandwidth were of course stuck at the compression level originally selected, and thus missed an average of 63% of the needed frames. In the latter case, the same clients only missed 35% of the needed frames, because of the intervention by Workflakes, which tried to re-assign them to more adequate compression levels for their actual bandwidth. These results provide evidence of the benefits of the adaptive scheme implemented by the Workflakes controller. Figure 18 shows the statistics of the missed frames for all the experiments: in total 26 trial runs were carried out, but only some of them reported non-0 values for the count of missed frames.

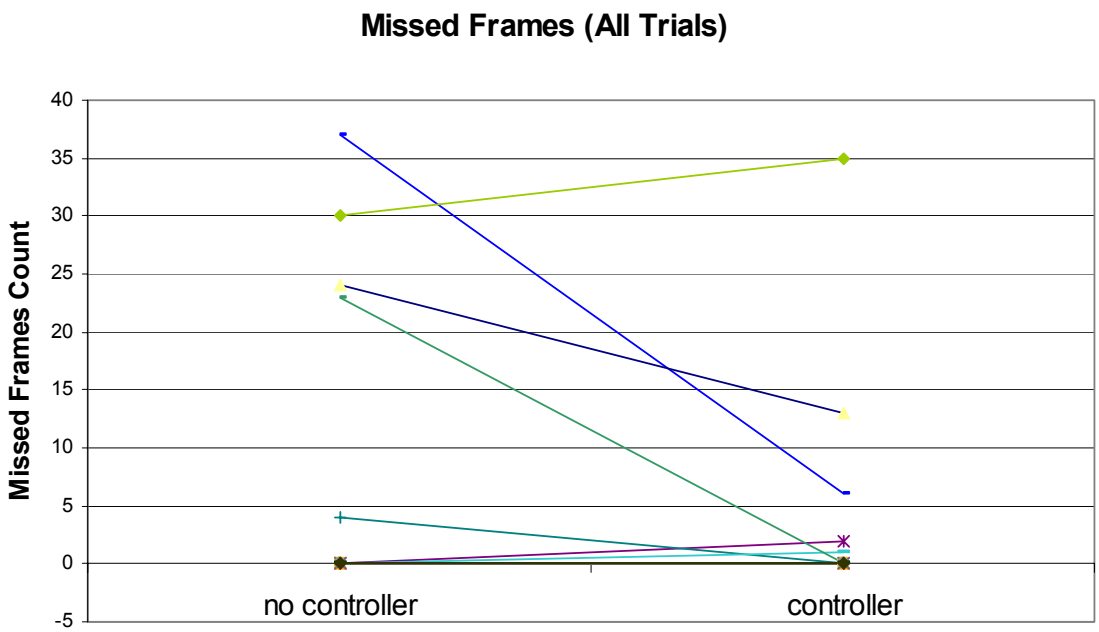

Figure 18: AI^{2}TV - missed frames count.

The other major goal of dynamic adaptation in the AI^{2}TV case study is to provide each client with an enhanced viewing experience, via adjustments to the compression level and hence the video frame rate. To attain a quantitative measure of the quality of service provided by a client assisted by Workflakes, a scoring system relative to the baseline client's quality level is used, with weighted scores for each level above or below the baseline quality level. The weighted score is calculated as the ratio of the frame rate of the two levels. For example, if a client is enabled via dynamic adapation to play at one level higher then the baseline, and the baseline plays at an average N frame per second (fps) while the higher level plays at 2*N fps, the given score for playing at the higher level is 2. Theoretically, the baseline client should receive a score of 1. The weights are thus calculated as a proportion between the average frame rates of the various quality levels, which makes the scoring system sensitive to the relative quality difference between layers in the compression scheme. Figure 19

shows the distribution of bonuses and penalties in the scoring system adopted: consider that the layered compression scheme employed in the case study has five layers.

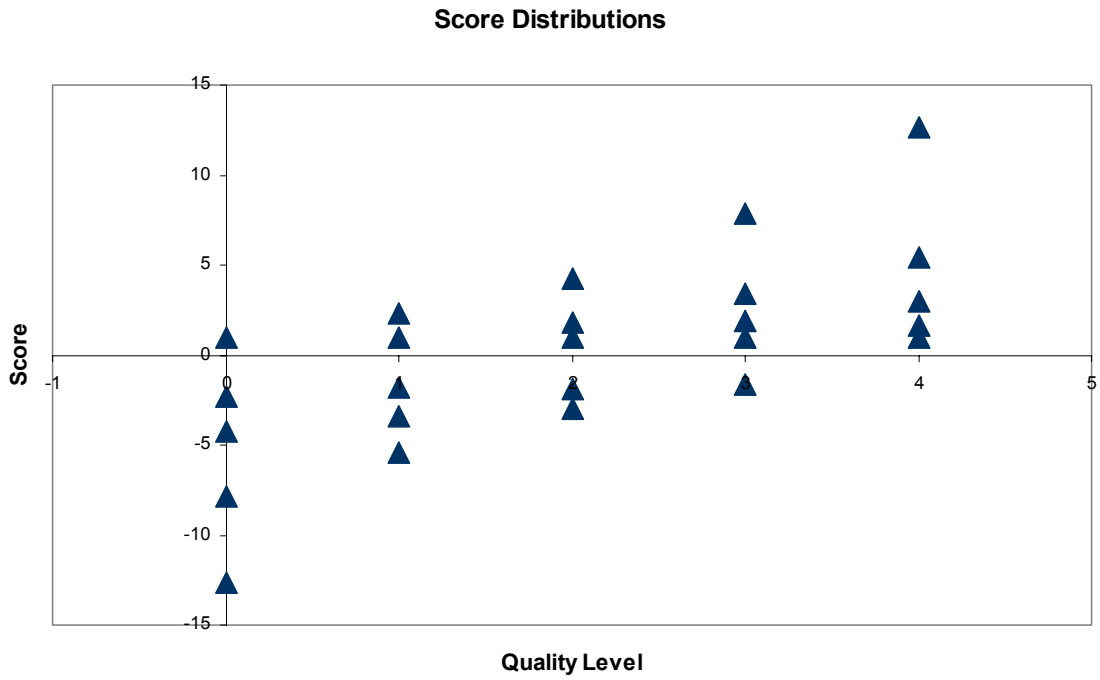

Figure 19: AI²TV - score distribution.

The results of the trial runs used for the evaluation of quality of service using the scoring system explained above show that the baseline clients scored an average group score of 1 in the various trial runs (as expected) while the clients adapted by Workflakes scored a group score of 1.25. The one-tailed t-score of this difference is 3.01 which is significant for an alpha value of .005 (N=17). That demonstrates with confidence that the dynamic adaptation orchestrated by the Workflakes controller is able to achieve a statistically significant positive difference in the quality of services. Note that the t-score does not measure the degree of the positive difference achieved by the autonomic controller. To measure the degree of benefit provided by

Workflakes, the proportion of additional frames that each adapted client is able to enjoy is measured. Overall, those clients received 20.4% more frames then the clients operating at a baseline rate (with a standard variation of 9.7). The benefits brought about by the introduction dynamic adaptation are visually evident in Figure 20, which shows the statistics of the weighted score for the baseline experiments.

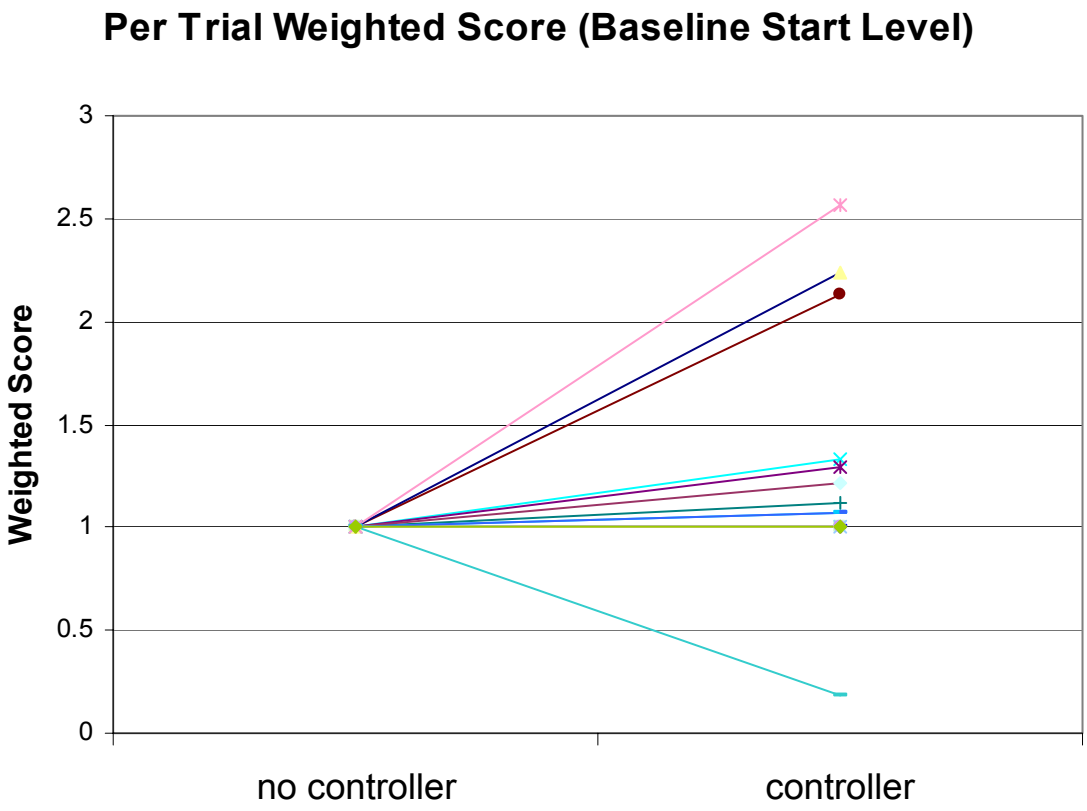

Figure 20: AI²TV - weighted score differences for baseline trial runs.

When considering all the test runs, that is, also those in which the allotted bandwidth was chosen casually, the difference of the weighted score in favor of Workflakes-adapted clients becomes even more significant, as evident from the bar graph of Figure 21. In particular, it is noticeable how the score of trial runs with no controller

in place drops significantly, while the score of controllers assisted trial runs decreases only a little, and remains well above 1.

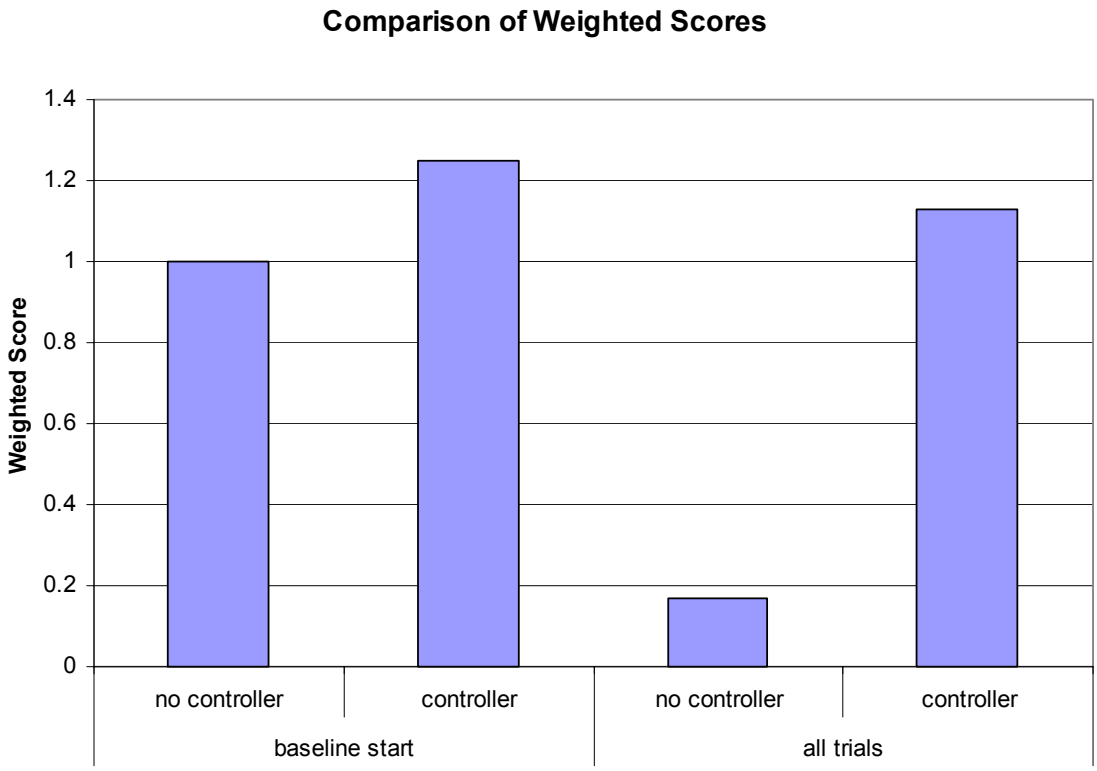

Figure 21: AI^{2}TV - comparison of average weighted scores.

For completeness, the statistics of the weighted scores for the various trial runs in which bandwidth is chosen casually are also provided in Figure 22.

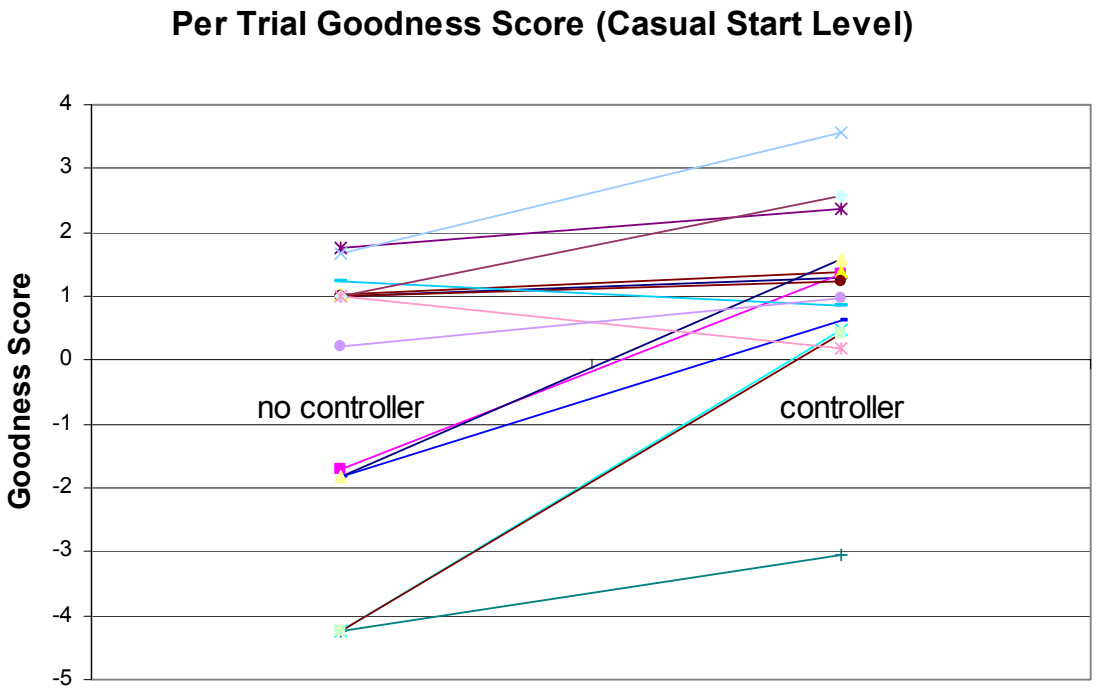

Figure 22: AI^{2}TV - weighted score differences for non-baseline trial runs.

The act of running the client at compression levels that require more bandwidth than the baseline level puts of course the client at risk of missing more frames, because the controller is trying to push the client to a better, but more resource-demanding, level. To measure whether the Workflakes-adapted clients are exposed to a higher risk of missing frames, the number of missed frames during a video session in those conditions was also counted. From that assessment of the risk of enhancing the frame rate of the clients, there was only one instance found, in which a Workflakes-adapted client missed two consecutive frames. Upon closer inspection, the time region during this event showed that the video demanded a higher frame rate while the network bandwidth assigned to that client was relatively low. The client was able to consistently maintain a high video quality level without skipping frames after that event.

The data reported from these experiments indicates how the Workflakes controller makes a significant positive difference in aiding the client to achieve a higher-quality viewing experience (all the while keeping clients in the group in sync), in two respects: less missed frames when bandwidth conditions are dire, and better video quality (i.e., frame rate) with respect to the available resources of each client. Note how the count of missed frame is kept separate from the weighted score of quality levels, to discriminate between levels of concern, though they both indicate a characteristic of quality of service.

An important qualitative consideration that supplements and in some way completes the above mentioned quantitative findings derives from an observation about the structural simplicity of the dynamic adaptation process employed for the short-term synchronization workflow. A process that (in the chosen process formalism) can be expressed in a rather straightforward and compact way is able to orchestrate effectively significantly complex effects on an ensemble of distributed and independent software components, such as those required to solve the problem of synchronized multimedia delivery to multiple recipients.

In the end, it may be interesting to compare and contrast the use and results of Workflakes in the AI^{2}TV case study with works that presents similarities.

QFabric [38] describes an internalized system for end-to-end management and adaptation of the QoS in soft real-time systems like multimedia conferencing. QFabric integrates resource managers in the operating system kernel and adapters in the application, which can therefore collaborate towards the same QoS goals. QFabric is based on the exchange of publish/subscribe events among kernel as well as

application-level entities involved in its target system; it uses the Event-Action paradigm to describe its adaptations, as reactions to specific monitoring or steering events. The work focuses mainly on the description of the abovementioned collaborative mechanisms and on how the infrastructure makes them available; no specific attention is devoted to how one could specify and automate on top of those mechanisms some policies that would guide the adaptation across the whole of the target system. In that light, QFabric and Workflakes could be seen as complementary, with Workflakes providing the means for policy specification and enactment through the use of process technology, and thus fulfilling the decision and coordination role of dynamic adaptation, while QFabric could provide the infrastructure to accommodate the monitoring, diagnostics and actuation roles.

This case study can also be compared with an earlier implementation of AI^{2}TV, which is described in [188]. In that version, a 3-D Collaborative Virtual Environment (CVE) called CHIME [189] was employed to support a variety of interactions of the study team, with the optional video display embedded in the wall of a CVE "room". The same semantic compression capability was used. Video synchronization data was piggybacked on top of the UDP peer-to-peer communication that was used at the same time for CVE updates, such as tracking avatar movements or other scene changes, in the style of multi-player 3D gaming. In that implementation, the video synchronization did not work very well, due to the heavy burden caused by the CVE on client resources; also, in that framework video quality optimization was not addressed. The new implementation of the case study presented here can run alongside the CVE in a separate window, and, thanks to the dynamic adaptation

superimposed by using Workflakes, can enhance both the group synchronization and the quality of service aspects.

5.3 GeoWorlds

GeoWorlds [168] is a strongly decentralized and componentized Internet-scale Information System, developed at the Information Science Institute (ISI) of the University of Southern California, which provides Geophysical Information integrated with Digital Library features. It is in experimental use for intelligence analysis at US Pacific Command (PACOM). GeoWorlds is built out of a distributed set of services glued together by Jini [165], which are employed to run complex information gathering jobs, expressed as GeoWorlds *scripts*.

Forms of dynamic adaptation applied to GeoWorlds have varied from service parameter modification, to component repair, to architecture-level reconfiguration, such as service migration. The latter is discussed in more detail in the remainder of this Section.

A number of different GeoWorlds execution scripts rely on computationally-intensive backend services, one of which is a noun phraser that analyzes incoming news articles and extract nouns for mapping onto geographical locations. That component is very commonly and heavily used by most GeoWorlds scripts. When the computational load due to noun extraction requests can potentially become excessive on a certain host, relocation is desirable to maximize performance or even avoid crashes.

Therefore, in this case study, dynamic adaptations that would relocate the noun phraser, and more in general any GeoWorlds components were put in place, taking also advantage of the inherent re-locability of Jini services. Sensors measuring the

overall computational load of hosts were developed, together with an architectural description of GeoWorlds, which specifies constraints for host machines and services residing upon those hosts. During the execution of the various services, if the load exceeded a predetermined threshold for a predefined period of time, gauges in the diagnostic layer of KX would detect and report it as a violation of those constraints. That would trigger a repair that entails moving the services on the overloaded host to a different Jini-enabled host that can accept the extra load.

Additional logic was also developed, to detect and avoid "oscillation" situations, in which multiple re-locations would occur in a short time span, and would cause services to move back and forth between two hosts. In such a case one of two *meta-repairs* could be taken: either the invalidation of the re-location repair strategy altogether, or the tuning of the overload threshold and/or period parameters of the gauges in charge to detect the overload condition. Both of those remedies represent cases of meta-adaptation, in which the dynamic adaptation platform (i.e., KX) itself is adapted, to better support the requirements presented by the target system.

One particularly interesting trait in this case study was that – in part building upon the experience gathered the IM case study of Section 5.1 – the GeoWorlds models were formally expressed with Architectural Description Languages (ADLs), and integrated within the dynamic adaptation loop, instead of using a proprietary format. The ABLE tool set [54] by CMU provided a formal model of GeoWorlds – including the afore mentioned constraints - to KX, in particular to its analysis and decision layers. The knowledge captured by ABLE was explicated as a set of descriptions in the Acme ADL [166] and maintained with the AcmeStudio editing tool [167]. Moreover, using

the AcmeStudio's dynamic visualization tools included in ABLE, it was possible to follow variations in the load and service state, and watch the feedback loop in action, in concert with the architectural model.

That juxtaposition of architectural representations and the corresponding implementation-level elements the GeoWorlds case study showed the potential of being able to clearly and rigorously express, reason about, validate and audit the characteristics and the effects of the modifications caused by dynamic adaptation. A difficulty that was only partially resolved in the case study was a degree of disconnection between the architectural model of the target system and its implementation counterpart in the run-time environment. Elements in the architectural model were not originally meant to be associated to and actually identify with deployed target system components. As a consequence, the need for precise bindings (such as those described for instance in [61]) between components and connectors in the architectural model and the runtime entities that reify the architecture in the field was observed. Such bindings can greatly simplify the integration of ADL-based tools at all layers of our dynamic adaptation infrastructure.

5.4 Web services marketplace

The IM case study described in Section 5.1 was developed in part within the context of a collaborative international project funded by EURESCOM (http://www.eurescom.de). The case study described in this Section was also carried out in the same project, by different project partners. Its description in full detail is available elsewhere [67], and its complete evaluation results are not available in this

document, since they represent confidential information belonging to the organizations that carried out the experiment.

The subject of this case study is a prototype of an adaptive electronic marketplace for the selection, negotiation and composition of Web Services applications. Said marketplace interfaces with a number of service components implemented and made available by multiple providers as Web Services, and offers to assemble complex services starting from scripted service chains.

The dynamic adaptation regards as its target system the core of the marketplace, which operates as a mediator and a composer, but the platform also monitors the basic functioning parameters of participating Web Services (such as availability, responsiveness, transaction completion ratio, etc.), analyzes their accumulated performance, and uses this information to adapt the behavior of the mediator in the marketplace The goal of the process-based coordinator in this case study is threefold:

- to automate the deployment of the core components of the marketplace;
- to intervene in the case of a failure of those components, and re-start them, in order to ensure the continuous availability of the marketplace;
- to modify on the fly the parameters informing the selection component of the marketplace, based on diagnostic information collected from the performance history of the external Web Service known to the marketplace.

The third aspect is of particular interest because it differs from the others, which are related to typical concerns of dynamic software adaptation, such as automated configuration and fault recovery; it borders instead on the issue of supporting dynamic software composition (see Section 2.4). By putting in place adaptive

mechanisms that can be used in selecting service providers for composing the service chains, it effectively provides the mediator component with the capability of tuning its match-making and selection of Web Services that take part in a given composed service. The final goal this kind of adaptation responds to (and the rationale guiding this case study) is to ensure compliance with requirements that may be set for the composed services, thus enhancing customers' satisfaction.

6 Evaluation

It may be useful at this stage to recall the two main working hypotheses from which this research described herein derives, as originally stated in Section 1:

H1) It is feasible and effective to employ an externalized infrastructure to retrofit pre-existing software systems and components thereof with dynamic adaptation features.

H2) Decentralized process technology provides a convenient and effective means to exert sophisticated forms of coordination and control over complex distributed software applications, such as those required by dynamic adaptation.

The first hypothesis has a lot to do with the general concept as well as the implementation of an externalized dynamic adaptation platform at large. This research, which concentrates on the coordination role of such a platform, can of course validate that hypothesis to the extent in which the coordination role is central to externalized dynamic adaptation, and inextricably tied to its other roles. On the other hand, having in place a dynamic adaptation loop, like KX, in its entirety evidently represents a prerequisite for experimenting with a process-based coordinator like Workflakes. Therefore, any positive or negative experience with the coordinator is immediately reflected upon the success or shortcomings of the entire platform to exert dynamic adaptation, and vice versa.

Some effectiveness and benefits indicators that can be applied to the evaluation of the hypothesis H1 can be broadly categorized as follows:

- Impact on the management practice related to the target system (for example in terms of effort and costs).
- Impact on the run-time behavior, performance and quality factors of the target system.
- Impact on the development of the target system (or, in the case of legacy target systems, feasibility of the approach without any impact).
- Adaptation reach and granularity that can be achieved.

The aspects listed above will be considered in Section 6.2, for framing and interpreting the results of the evaluation of Workflakes with respect to working hypothesis H1.

Any benefits related to the aspects above could in principle be measured relatively, that is, against those provided by alternatives approaches to dynamic adaptation, in particular, in this case, internalized approaches. However, there are several serious difficulties to accomplish that kind of comparison. One difficulty descends directly from the externalized stance taken by this research: externalized dynamic adaptation is concerned principally with retro-fitting legacy or other third-party software systems with adaptive capabilities. As a consequence, the legacy software that was selected for those experiments did not exhibit any intrinsic adaptive features.

There are also more fundamental difficulties that hinder relative evaluation. First of all, the lack of an agreement upon baselines for the evaluation of adaptive or autonomic capabilities in software systems: given the relative novelty of the field, there is no accepted or even proposed base experiment (or set thereof), upon which to compare different approaches and implementations. That is in turn a consequence of

the wide scope and reach of studies and results in the field, which aim at improving software systems in many different, heterogeneous quality areas.

For all of those reasons, it is more feasible to measure the benefits brought about by applying dynamic adaptation in absolute terms, that is, against a situation in which no adaptation whatsoever is exerted upon the same legacy system. That is particularly true for externalized approaches, which can be easily turned off or plugged in at will, and is the general approach taken in this work. In Section 6.3, the opportunity of overcoming the lack of a proper evaluation framework for initiatives that deal with the problem spectrum of dynamic software adaptation and, in general, autonomic computing is discussed.

The evaluation related to the second hypothesis looks at how well process technology is able in the cases at hand to describe, support and automate software coordination plans for dynamic adaptation. Also that evaluation is carried out here in absolute terms, for reasons similar to those explained for hypothesis H1: in particular, there is no sufficient accumulated experience in the dynamic adaptation field, to establish consistent benchmarks or canonical experiments against which multiple alternative approaches (in this case, coordination paradigms) can be compared.

Among the indications of the effectiveness of a process-oriented approach to coordination of dynamic adaptation there can be aspects such as:

- The effort needed to specify the coordination policies of software dynamic adaptation in terms of a process / workflow.
- The level of sophistication and complexity of those coordination policies that can be feasibly handled.

- The level of efficiency of the runtime support enacting and automating that coordination policy.
- The range of problems that can be expressed and addressed.

The aspects listed above will be considered in Section 6.2, for framing and interpreting the results of the evaluation of Workflakes with respect to work hypothesis H2.

6.1 Assessment of the experiments

To weigh the value and the potential benefits of the ideas as well as the system developed in this research, one natural way is to look at the experiments that have been carried out, and described in Section 5. From their findings and results, it may be possible to infer a certain set of contributions, and a number of claims that can be made on the basis of those contributions. Those can in turn be related to some of the major issues in externalized dynamic adaptation, and, more specifically, about its coordination role.

The table in Figure 23 summarizes the case studies described in Section 5: it classifies the kind of dynamic adaptation they provide, according to the four major concerns of autonomic computing (configuration, healing, optimization and protection); additionally, for each of those case studies, it also displays the kind of impact that dynamic adaptation is intended to have on its corresponding target system.

From the Table, it is visible that the dynamic adaptation exerted in the case studies significantly covers many different concerns that are relevant in the field, with the exception of protection / security issues. The recent SABER work [159], however, represents an effort to extend that coverage, since it proposes to employ the concepts

that have guided the development of KX and Workflakes, as well as the experience and results that derived from those works, to address security and survivability issues. Therefore, it appears that the lack of application of this research to protection may be incidental, rather than principled.

A first conclusion that can be drawn and a first claim that can be made is therefore that the approach that is pursued by this research is sufficiently general to apply to the majority of contexts and scenarios that are envisioned for dynamic software adaptation.

A similar analysis can also be made with respect to the typology of the target systems used in the various experiments, aiming at showing the suitability of certain categories of system for the proposed approach to dynamic adaptation. Various target system categorizations can be drawn, according to different dimensions. In the Tables displayed in Figure 24, Figure 25 and Figure 26, three orthogonal dimensions are used, referring to the degree of distribution, the real-time characteristics, and the distributed computing infrastructure layer upon which the target system mainly operates.

Case Study	Target Domain	Autonomic aspects				Benefits
		Configuration	Healing	Optimization	Protection	
IM	Internet, Communication	Deployment Server configuration Staging	Fault recovery Fault prevention	Scalability Resposiveness		Availability, Reliability, Mgmt. costs
AI^{2}TV	Soft real-time, Multimedia			Client Synchronization		Correctness
GeoWorlds	Internet, Information Systems	Deployment Architectural changes	Fault recovery			Availability, Reliability
Web Services Marketplace	E-business, Componentized services	Deployment	Fault recovery	Composition policy		Mgmt. Costs, Functional enhancement

Figure 23: Table of case studies contributions.

Case Study	Distribution			
	LAN	Corporate WAN	Extranet	Internet
IM		✓	✓	
AI^{2}TV				✓
GeoWorlds				✓
Web Services Marketplace			✓	

Figure 24: Classification of experiments (distribution dimension).

With respect to the distribution dimension shown in the Table of Figure 24, the conducted experiments demonstrate a sufficiently complete coverage across its spectrum. No target system in any experiment specifically operates on a LAN, but the issues that are typically present in a LAN are generally subsumed by those that can be found in a Corporate WAN or in an inter-organizational Extranet, which are both covered in the experiments set.

It is noticeable that in the GeoWorlds experiment, the architecture-level adaptations were in fact experimented with across the Internet at large, with the Workflakes engine sitting in Italy, and the other KX components, as well as the hosts where adapted GeoWorlds services were running, situated instead in New York (in the PSL laboratory of Columbia University), California (in the ISI facilities of the University of Southern California), and, in one occasion (for the Demonstration Days of the DARPA DASADA program) in Baltimore, Maryland,. The only noticeable effect of such a widespread configuration were – understandably - rather long delays in the

flow of communication throughout the dynamic adaptation loop, and consequently in its end-to-end response time. Given the non-real-time nature of the adaptations carried out and of the GeoWorlds system at large, those delays did not represent a critical issue.

Case Study	Timeliness		
	No real-time	Soft real-time	Hard real-time
IM	✓		
AI^{2}TV		✓	
GeoWorlds	✓		
Web Services Marketplace	✓		

Figure 25: Classification of experiments (real-time).

In fact, as shown by the Table in Figure 25, timeliness requirements are not present in any of the experiments, with the exception of AI^{2}TV, in which soft real-time aspects of dynamic software adaptation were purposely investigated (see Section 5.2). The available experimental data provides evidence that an externalized and process-based dynamic adaptation approach can be effective also for target systems that have soft real-time requirements, whereas it cannot shed light on hard real-time systems. For them, the principled objections outlined in the discussion about timeliness of Section 3.3, for instance regarding the highly variable delays that could be induced by the response time of the externalized control loop, remain valid.

Case Study	Operation Layer				
	Service / Appl.	Middleware	Data	O.S.	Networking
IM	✓×	×	×		×
AI2TV	✓×				
GeoWorlds	✓×		×		
Web Services Marketplace		✓×			

Legend:

✓ Target system operates at this layer

× Dynamic adaptation occurs at this layer

Figure 26: Classification of experiments (main operation layer).

The Table in Figure 26 lists different layers that are recognizable in a typical distributed computing infrastructure. As it is evident by simply looking at the works reported in the issue of the *IBM Systems Journal* devoted to autonomic computing [77], dynamic adaptation, in some of its many possible incarnations, appears to apply to software systems and components operating on all of those layers. Adaptation can start from the bottom with the network transmission layer [78], and move up to the topmost layer, where user applications operate and provide their services [81], through the intermediate layers represented by operating system [80], data storage and management [79] and middleware [82], including application server architectures [83].

The Table shows how the examples selected to validate Workflakes mostly concentrate upon target systems whose main area of operation is the application layer. That can be explained in two ways: firstly, the higher the distributed computing layer at which target elements operate (that is, the closer to the application layer), the easier is in general to have available and leverage open interfaces that enable the essential monitoring and actuation roles of the dynamic adaptation platform; on the contrary, the lower layers may remain partially or completely hidden, and are likely to be used in a black-box fashion by services at the higher layers. Furthermore, it is in general easier to elicit the requirements and estimate the impact of dynamic adaptation for target systems that must deliver some tangible service to end users. As a consequence, also the dynamic adaptation exerted on the selected targets impacts mainly the same layer. However, as shown in the Table, certain adaptations that are necessary to bring forth benefits onto the main operation layer of the target system are sometimes carried out also upon different layers.

The only area where no investigation was carried out in the context of this work was the operating system layer. That can be seen as a consequence of the externalized stance of the dynamic adaptation solution proposed: adding some adaptive features to an operating system from the outside would imply that a program that is executed in a non-privileged mode could acquire some deal of runtime control and influence upon the innards of the operating system, which is something that is generally not recommended. Operating systems represent a domain in which internalized mechanisms are more adequate to achieve adaptation capabilities, either statically, for instance through extensibility, like in [84] or [85], or dynamically, like in [86].

Considering all of the above, the experiments sufficiently show that the described approach can be applied to dynamically adapt most of the major recognizable elements that constitute a typical environment for distributed software applications and services. Given the nature of the experiments, however, dynamic adaptation at the middleware and data layers could be only partially explored.

Only one experiment focuses specifically on middleware issues; among the selected target systems, some of the others do not rely on a significant amount of middleware software, while for others still the presence of middleware is completely transparent and remains orthogonal to dynamic adaptation issues. From the literature, however, it is possible to derive ample evidence about the applicability of dynamic adaptation at the middleware layer. For example, investigation of reflective middleware [93] (i.e., how middleware can dynamically adapt itself, in addition to applications that run upon it) has granted – among other results - a number of International Workshops [87] [88] and a permanent space in the IEEE Journal on Distributed Systems Online [89]. Moreover, some reflective features promoted by research, like [2] for reflective Object Request Brokers (ORBs), are beginning to be accepted more widely even in commercial middleware platforms.

In fact, run-time adjustments to the middleware platform upon which an application is built are a powerful way to cause system-wide effects on that application. Reflective middleware can achieve that, in a way that is in line with the idea of internalized adaptation; externalized adaptation impacting the middleware layer, for example through one or more “autonomic services” that can be plugged onto a generic middleware core and that encapsulate dynamic adaptation roles, is a specific research

thread that deserves to be further investigated. The orchestration capabilities of such an approach could be possibly developed by building upon and extending workflow capabilities that are being already incorporated within certain middleware platforms, such as Grid computing [50] [92] and Web Services [94] [95]. Those workflow capabilities are however currently oriented principally towards the domain of automated, on-the-fly service composition, as discussed in Section 2.4, and exemplified to an extent by the Web Services marketplace case study described in Section 5.4.

Regarding dynamic adaptation at the data layer, the presented experiments treat their data sources mainly as a black box; therefore, their dynamic adaptation is either not concerned with the data layer at all, or exerted simply upon the interface of application components with data management and storage components. Dynamic adaptation regarding directly the latter is not considered.

Adaptation issues like query distribution schemes [90], adaptive caching [91] and learning query optimizers [79] constitute specialized forms of run-time database optimization, which are actively investigated within the database community and can be implemented within database systems. Were they available and exposed through an appropriate actuation interface by some data storage and management components, there would be no principled reason why those features could not be exploited by an externalized coordinator for the end-to-end adaptation of some target systems, in particularly data-intensive applications.

6.2 Assessment of the Workflakes system

All the case studies aim at validating the effectiveness of the process-based Workflakes controller, by measuring quantitative or qualitative benefits to the target applications put under its control. Those benefits, which are listed in the rightmost column of the Table in Figure 23, contribute towards the evaluation of the Workflakes system with respect to the two main working hypotheses previously remarked. Each case study and each result highlights one or more aspects pertaining to the working hypotheses and they can be classified and weighed accordingly.

The results of the various case studies are presented in detail in Section 5. Among the case studies, the two major ones (IM, see Section 5.1, and AI^{2}TV, see Section 5.2) provide quantitative measures as well as qualitative considerations. For the others, only qualitative observations are available. Those other case studies are reported principally because they reinforce with different examples some of the results observed in the two major ones, and also because they provide some degree of diversity, since (as shown in Section 6.1) they in part deal with different application domains, address different aspects of the autonomic computing field, or operate on different layers of the distributed computing environment of their target systems.

This Section intends to discuss the significance of the reported results, relative to the working hypotheses and limited to the two major case studies. As a prologue to that discussion, a comparison between the different process-based facilities used for the IM vs. the AI^{2}TV case study (Version 1 vs. Version 2 of Workflakes, respectively presented in Section 4.1 and 4.2) is hereby established, since it represents an additional result of the experimental work.

Coding vs. Modeling Dynamic Adaptation Processes

The IM case study was carried out with Workflakes Version 1; the AI^{2}TV case study, instead, represented the first application of Workflakes Version 2. There is value in comparing those two releases, to gain an understanding of the implications deriving from their different ways to develop and represent processes that orchestrate dynamic adaptation. Version 1 comported coding processes directly in a programming language, such as Java (although through the process specification paradigm and the corresponding libraries offered natively by Cougaar, plus the coding patterns that Workflakes Version 1 makes available on top of them, via shell plugins and process definition junctions). In Version 2, the process is modeled in an abstract and dedicated formalism that is substantially diverse and separated from the code in the runtime engine that interprets and executes that process.

One important preliminary observation is that in the IM experiment, the coordination process is considerably more involved than in the AI^{2}TV experiment. That is natural, considering the different natures and purposes of those two processes. The IM process essentially captures and automates system management procedures that must be enacted according to a situational logic; since the various target components impacted by the adaptation process are tightly dependent on each other for the delivery of the overall service, each phase of that process is also dependent on the outcome of other phases, and may incur internal as well as external contingencies that need to be properly accounted for. The AI^{2}TV process, instead, captures a synchronization scheme that needs simply to be executed from start to end periodically. Although the ultimate goal of the dynamic adaptation of AI^{2}TV is to

keep multiple clients synchronized, the single adaptations that may be necessary for the various clients do not have cross-dependencies that may influence each other. In case an adaptation does not work out as expected on one or more clients at some point, no particular disruption to the rest of the system occurs, besides sub-optimal group synchronization until the process is enacted again in the next round. Consequently, there are no contingencies to account for, nor alternative or exceptional process courses.

The complexity of the IM process is directly reflected in the amount of code that was written to implement it. The 6 Java classes defining the process definition junctions (i.e., the data and control flow of the process, according to Workflakes Version 1 4.1), account for almost 60% of the total lines of Java code written to customize the entire KX platform for the IM case study (and still around 50% of all the lines written, in case also XML code is counted in). It is therefore not surprising that the amount of effort necessary to develop the process specification for the IM case study with Workflakes Version 1 was in the order of several work weeks, which reflects the complexity of the problem and remains in line with the productivity that can be expected from a generic software development task.

In the AI^{2}TV case study, instead, most of the coding complexity resides not in the coordination role, but rather in the helper functions that implement the decision role and that the process invokes at each round to figure out if and how each AI^{2}TV client needs to be adapted. The code of those helper functions, although admittedly involved, is relatively lightweight, since they amount all together to about 200 lines of Java code. What is noticeable is that practically no other programming code

needed to be written to enable the client synchronization workflow of the AI^{2}TV case study, as presented in Section 5.2. The only other effort regarded the modeling of the process in Little-JIL from scratch, employing the Visual-JIL editor, which took a couple of work days.

Even by considering the different inherent complexity of the IM and AI^{2}TV coordination problems, one order of magnitude looks like a significant difference in favor of Workflakes Version 2 and its high-level process modeling capabilities. Further data to validate this observation could to be collected with the progressing of the AI^{2}TV case study, as the short-term client synchronization workflow is integrated and harmonized with the medium- and long-term workflows, in a larger, more complex and multi-faceted coordination process.

There is another factor that hints even more clearly at how hard it can be to define, maintain and evolve a dynamic adaptation process, when it is expressed in a rather low-level way, such as a conventional programming language. When, at a certain stage in the IM experiment, the process was extended to include the orchestration of graceful service staging (as described in Section 5.1), it was quite difficult to modify the code defining the process for that purpose, while maintaining it correct and backwards-compatible with the previous version, which was dealing only with the deployment and on-the-fly scalability of the IM service. In the end, it was simpler to load the junctions defining the staging process in a separate instance of the Workflakes engine, as opposed to run the process as a whole in a single engine: that comported a significant deal of unnecessary code duplication, since many of the tasks

in the staging process are the same used for orchestrating deployment and scalability, although wired together with a different and more complicated logic.

Using high-level process modeling facilities, it is easier to manage the maintenance and evolution of the process specifications and keep under control any growth in complexity like the one experienced in the IM case study. With the Little-JIL visual language, for example, it is easier to figure out at a glance the possible interdependencies of various parts of a process. It is also easier to promote re-use of fragments of the process specifications, since the hierarchical structure of Little-JIL naturally supports modularization; furthermore, Visual-JIL also enables to exploit that modularization by using *references*: a reference can appear as a legitimate sub-step in the expansion of any process step, and constitutes a pointer to some other sub-tree in the process hierarchy.

Interpretation of case study results

The IM case study was among the first experiments that were carried out with KX in its entirety (together with the GeoWorlds experiment – see Section 5.3), and the first which demanded a significant level of complexity for the coordination role. As such, it was instrumental in verifying the general feasibility as well as the effectiveness of an externalized approach to dynamic software adaptation. Therefore, measures and observations deriving from the IM case study contribute principally to evaluate the issues related to working hypothesis H1. At the same time, the IM case study contributes to evaluate also the suitability of a process-based approach to the description and enactment of coordination in dynamic software adaptation, which relates to working hypothesis H2.

It is the AI^{2}TV case study that contributes mainly to the validation of hypothesis H2, with respect not simply to the suitability but also the effectiveness of the process-based approach embraced by Workflakes, as well as of its implementation in the Workflakes system itself.

Considering all of the above, it is possible to draw a synthesis that evaluates the presented results against each working hypothesis. As far as working hypothesis H1 is concerned, such a synthesis is presented in Figure 27: that Table reports findings coming from the IM case study only.

The Table in Figure 27 conveys a twofold message. First of all, it shows how the accomplishments of externalized, process-orchestrated dynamic adaptation applied to the IM case study are in line with the kinds of benefits that are typically expected of autonomic computing and analogous initiatives. One of the major motivations for autonomic computing is to alleviate the exploding complexity of the management and administration of today's software systems; another major claim is that running autonomic software can bring about and maintain higher QoS levels. The accomplishments summarized by, respectively, the "Mgmt. savings" and "Runtime improvements" columns of the Table address precisely those two issues. Therefore, they serve as a confirmation of how the approach proposed by this work is a suitable means to reach some of the primary goals of autonomic computing.

Benefit	Example	H1 – Feasibility and effectiveness f externalized adaptation			
		Mgmt. savings	Runtime improvements	Development impact	Granularity level
Deployment and configuration	½ day to 2 min.	×		None	Architecture Component Parameter (setting)
System administration	Automated / continuous	×	×		
Improved reliability	40 s. reaction time		×		
Improved availability	On-the-fly scalability	×	×		

Figure 27: Table summarizing contributions towards hypothesis H1.

H2 – suitability and effectiveness of process-based coordination								
Case Study	Goal	Complexity factors					Effort	Runtime Support
		Dynamic planning	Internal contingencies	External contingencies	Compensations	Time constraints		
IM (automated mgmt.)	Reduce costs Enhance QoS	×	×	×			Weeks	**Time**: n.a. **Complexity**: process split in multiple engines
AI^{2}TV (distributed sync.)	Enforce correct behavior Enhance QoS	×				×	Days	**Time**: ~200 ms (soft real-time)

Figure 28: Table summarizing contributions towards hypothesis H2.

Furthermore, the results reported in the other two columns, “Development impact” and “Granularity level”, underline together how embracing an externalized approach to dynamic adaptation can be (at least) as effective as internalized approaches. One of the potentially critical limitations of externalized dynamic adaptation is that it has only limited access to the innards of a legacy target system – especially whenever the source code is not available – which could prove insufficient whenever very involved and fine-grained adaptations are needed. That is instead by definition not an issue for internalized approaches. In the IM case study, it was possible to effect adaptations at various – even quite fine – granularity levels without impacting in any way the development of the target system. Such a result testifies that, by paying the right amount of attention to the design of the platform and the use of appropriate technologies for the contact points with the target system, externalized dynamic adaptation may be able to overcome that hurdle, and deliver its signature advantage: the ability to orchestrate end-to-end dynamic adaptation across the various distributed and heterogeneous components of a legacy target system.

The Table in Figure 28 presents a synthesis of results that pertain to hypothesis H2. In the first place, the “Complexity factors” column groups together a selection of issues that (without claiming to be exhaustive) represents aspects that are particularly relevant and potentially complex for the coordination role of dynamic adaptation. The Table shows how the processes defined for the two case studies address together the majority of them, with the exception of compensations (hiven the nature of the coordination problems, provisions for compensations branches in the processes were not necessary in both major case studies). Furthermore, the Table reports, side by side

with the complexity assessment just discussed, data about the effort needed to define the processes, which highlights the level of improvement and effort savings made possible by embracing an abstract modeling formalism. It also reports indications of level of run-time support and performance provided by the Workflakes engine in its various versions in enacting the processes.

For a characterization of the range of dynamic adaptation problems that were addressed it is instead possible to refer to Figure 23, which shows how the two major case studies together cover a variety of issues pertaining to configuration, healing and optimization. That, taken together with the diverse goals of the IM and AI^{2}TV case studies, which are reported in the Table of Figure 28, can be considered as evidence of the applicability of the process-based approach to a wide spectrum of problems.

The message conveyed by the Table is threefold. It shows that process-based coordination represents a suitable choice for a number of diverse applications of dynamic adaptation, even with demanding levels of complexity. It also shows that process technology has the potential to provide efficient run-time support even in domains that impose significant performance and timing constraints to the dynamic adaptation facilities. Finally, it confirms the necessity of a high-level enactable formalism to define, handle and manage any non-trivial coordination problem in the form of a process.

6.3 Limitations and open issues

The previous Section tries to provide an organic view of the accomplishments reached by the experimental work towards the working hypotheses inspiring this research. For a complete evaluation, it is equally important to assess the inadequacies

or shortcomings that have been found, and any issues that still need to be resolved, which can motivate future research efforts on this same theme.

In the first place, it is worthy to recall that there are some limitations of externalized dynamic adaptation that restrict its suitability, with respect to certain application domains, such as hard real-time, or certain categories of adaptation targets, such as operating systems. Those limitations have been already previously discussed within this document, for example as part of the principled critique of the approach in Section 3.4, or the assessment of the experiments and their coverage in Section 6.1. Those limitations seem inherent to the externalized nature of the approach; therefore, they can be viewed as known boundaries for its usability. It is possible that further investigation and experimentation will help identifying more clearly other such boundaries, which may have not been yet highlighted.

One of the most important outstanding problems at the current stage of understanding of processes as orchestration means for dynamic software adaptation, is that each single process (or process fragment) is developed *ad hoc* from start to end for each target system condition that triggers a certain adaptation. That is obviously difficult, costly and time-consuming. The foundations and techniques for moving from such ad hoc crafting to a more systematic engineering of coordination processes for dynamic adaptation are not yet well understood at this stage. Such systematization is likely to require the ability to capture a great wealth of knowledge about the target system at run time, and incorporate it seamlessly into the decision and coordination roles of the dynamic adaptation facility.

The availability of that knowledge could enable the direct generation of coordination patterns and processes in a largely automated way. For such a breakthrough, however, a very comprehensive target system model must be available. The development of such a model requires a two-pronged approach: a mix of advanced analysis skills and tools, coupled with powerful modeling abstractions and formalisms.

The modeling part of that problem is being addressed, for instance, by advancements in Architecture Description Languages, which attempt to extend architectural models with features that deal with dynamic aspects. Besides capturing with increasing sophistication the behavioral aspects of a system in addition to the structural aspects, they start to address other issues that are important to achieve run time support, such as maintaining the model consistent with respect to the running system, continuously evaluating the system configuration against the model for the diagnosis of anomalous behaviors or other conditions, and selecting architecture-level adaptations that maintain or bring back the system as a whole to legal configurations, as per the architecture definition. Such architecture-based adaptation [53] [62] attempts to root the otherwise mainly empirical work of developing dynamic adaptation software in the realm of formal design.

Experimentation with architecture-based adaptation was a focus of the GeoWorlds case study described in Section 5.3. One of the lessons learned in that experiment as it stands regards the rather large gap that still exists between the amount and kind of system knowledge captured and made available by state-of-the-art architectural models and the nuances present in the system implementation, or, even more so, in a "live" instantiation of the implementation that runs within a certain computing

environment. That gap may be due in part due to the fact that, most of the time, architectural models are artifacts produced during the design phase of the software development process, and as such tend to provide an *a priori*, top-down perspective on the system, which necessarily remains somewhat abstract with respect to implementation details, independently from the expressiveness of the description formalism and the richness of the model. That means that architectural models by themselves might remain incomplete with respect to the amount of knowledge and detail necessary to capture and reason about a running system.

Run-time analysis tools could be employed to bridge the remaining gap. The synthesis or enrichment of architectural models *a posteriori,* from the observation and analysis of the running system, such as for instance in Software Surveyor [66] could complement architectural modeling by adding a bottom-up perspective on the architecture.

An alternative may be to produce a system model that directly captures the essence of the implementation as is, rather than abstracting it up at the architectural level. [81] proposes to attach formal behavioral descriptors to code blocks (i.e., modules): it argues that the bottom-up perspective provided by a model derived from the implementation is intrinsically more faithful, detailed, granular, and hence more suitable for the purposes of dynamic adaptation, than a model conceived mainly at design time. That method requires that developers pick up the practice of including the descriptors in their code, which can be seen as an extension of their duties, although somewhat akin to documentation.

However construed, a sufficiently complete model would offer a significant degree of support to the engineering of processes for the coordination of dynamic software adaptation. By being able to understand and reason in the abstract about the various possible ways in which a system needs to be adapted on the basis of a model, it may be possible to come up at design time with the right set of process fragments, which can contribute to the solution of a variety of dynamic adaptation situations for a given target system; then, at run time, it may be possible to compose them as needed by the situation at hand. Thus, the final layout of the process orchestrating a certain complex dynamic adaptation could be decided dynamically.

A possible extension in that direction of the use of architectural models is described below. It assumes capabilities and features for the architectural tool set similar to those provided by CMU's ABLE, which has been already experimented with in the context of the GeoWorlds case study.

First of all, to enable that scenario, some gauges are devoted to diagnosing and reporting architecturally significant events to the architectural tool set. Those gauges serve the purpose of checking and maintaining consistency between the system running on the field and its model, and can be constructed starting from certain aspects captured by the model. For example, that includes the compliance with constraints placed on the model, which state what are the acceptable working conditions for the system. The architectural tool set evaluates gauge events such as the violations of one or more of those constraints with respect to the model, and takes dynamic adaptation decisions, according to a logic that is completely encapsulated by its knowledge and understanding of the model, and can remain opaque to the rest of

the platform. Decisions are expressed as transformations, which again predicate on the model, and impact the layout and the attributes of the architecture. Those architectural transformations, which aim at placing the system in a new configuration that respects all constraints, can require one or more operations: each operation is a directive which represents a modification of some part of the model. Since that modification must also be effected on the implementation of that model, it also represents a trigger for a ration process fragment that impacts accordingly the target system running on the field.

As a simple example, a single directive that could be part of an architectural transformation and that could be pushed from the architectural to the implementation level (that is, from the model-based decision role to the coordination role) could be something like: “Move Service X from Host A to Host B”. While that can be seen as an atomic architectural transformation operation, it needs to be translated at the implementation level in a process fragment made of multiple, fine-grained adaptation steps. That adaptation process fragment may involve for example the following steps:

- deployment and instantiation of a new instance of the component providing Service X on Host B;
- detachment of any communication links between the “old” instance of X on Host A and other running components of the target system;
- re-establishment of corresponding communications with the new instance of X;
- and, finally, shutdown of the instance of X still running on Host A, but not needed anymore.

The sequence described above is hypothetical and possibly simplistic; furthermore, it does not take in account any internal contingencies that can occur at some stage of the adaptation, which should be handled explicitly by appropriate secondary branches of the process.

Recalling that each directive is a single operation in a transformation of the architecture, it is evident that – with this approach - the architectural tool set drives the instantiation and enactment of as many concatenated process fragments as the various operations needed to complete the architectural transformation that is being enforced on the architecture.

Besides facilitating the engineering of adequate processes for dynamic adaptation, an approach of that kind may be the key to resolve another shortcoming associated to the ad hoc approach to the development of dynamic adaptation processes, that is, pre-determined (as opposed to open-ended) processes.

Currently, each trigger that signals a target system condition of interest must be well known in advance and unambiguously associated to the adaptation resolving it. Similarly the adaptation must be fully planned ahead, including the insertion of explicit provisions for the handling of all external and internal contingencies. That leads to processes that provide “canned” solutions only to a necessarily limited and pre-determined set of target system conditions, rather than to open-ended processes that potentially cover also unusual or unexpected criticalities as they occur.

The definition of open-ended processes is typically more bottom-up, compositional and fine-grained than that of a fully planned-ahead process; it would require a rich catalog of “elementary” process fragments that enact low-level adaptations, which

can be incrementally composed. That, however, comports at least two major difficulties.

One problem is that an open-ended process obtained via the situational composition of fine-grained fragments may remain completely implicit and "hidden" to an analysis carried out *a priori* on the process specifications. It could possibly emerge as a whole only after it has unfolded from start to end, i.e., from the arrival of some initial trigger that fires an initial process fragment and that signifies a certain macroscopic target system condition to be dealt with, to the achievement of the desired target system state that resolves the above-mentioned condition. In the end, the dynamic adaptation process would have evolved through a series of intermediate modifications to the state of the target systems, achieved via a series of incremental, low-level adaptations. But also an analysis *a posteriori* may fail to elicit the full-fledged process as it emerges from the open-ended composition of elementary fragments, because of the potentially large number of interacting process fragments and the variability of the resulting processes.

Even if the problem of understanding (and then being able to maintain) a dynamic adaptation process designed in a very open-ended fashion is resolved, for example, by exploiting means for analysis of the process activity logs such as those proposed in [29], another problem remains. It is quite difficult to decide upon the set and mix of elementary process fragments that must make it into the process specifications. The problem is to come up with the right catalog of fine-grained process fragments, which would be largely specific to each and every dynamic adaptation application, which should cover a large and possibly indefinite spectrum of situations even within the

same application, and whose interactions should ensure the correct response to those situations, in terms of dynamic adaptation orchestration.

Both of the above problems, furthermore, affect not only the design, but also the testing and validation of any dynamic adaptation facilities employing open-ended process-based coordination.

It seems that comprehensive modeling and analysis capabilities like the one discussed earlier as a support to the engineering of dynamic adaptation processes would also help in the solution of both of the above problems: a model could be used to drive and validate the selection of process fragments to be placed in the catalog; the complementary ability to record and analyze the dynamics of the target system against the blueprints provided by the architectural model would ease the task to elicit and reason about the open-ended processes implicitly put in place.

Another open issue in relation with this work is the difficulty to come up with an evaluation framework for comparing dynamic adaptation approaches relatively to each other. The elaboration of appropriate metrics that can be used to frame such a relative evaluation is a theme that has not been investigated much to date, in part because of the novelty of the field. But a more fundamental problem comes from the quite broad goals and scope of dynamic software adaptation and other analogous initiatives. If one looks at the two major case studies reported in this work, the differences are already macroscopic. In the IM case study, the major intended improvements occurred in the area of service management efforts and costs, and in the area of improved service availability; the intended effect of dynamic adaptation in the AI^{2}TV case study, instead, is all about enforcing the correct behavior of the

system while enhancing its quality. Those goals are expressed in entirely different terms.

Since many disparate areas of interest and investigation like the ones mentioned above co-exist in the dynamic adaptation problem space, it is particularly hard to come up with a limited and coherent set of application-independent metrics that can capture and summarize the validity of general-purpose dynamic adaptation facilities. For example, a common claim in the context of autonomic computing is how it can greatly reduce system management costs, and thus of the Total Cost of Ownership (TCO). Since it can be legitimately argued that system management costs largely reflect the effort necessary to handle and keep under control the complexity of the managed system, and since the taming of such complexity is the original motivation and goal of autonomic computing, TCO reduction might be seen as a valid candidate for a generic evaluation metric. However, as exemplified by our AI^{2}TV case study, it is easy to find applications of dynamic adaptation whose benefits to the target system cannot be measured at all in terms of reduced TCO.

A possible alternative to trying to constrain within a fixed set of dimensions the evaluation of dynamic software adaptation, is to abandon the idea of accomplishing relative evaluation independently from the application, which might prove more realistic, given the nature and heterogeneity of its problem space.

A possibility could be the creation of a composite benchmark, including a variety of baseline experiments that cover the various areas within the problem space of dynamic software adaptation. It might be possible to define and exploit to that end a set of target systems in different application domains, choosing from well-known

systems, perhaps best-breed open source projects. Different approaches could be then compared relatively to each other in terms of how they score with respect to the various experiments, and the scores would capture the benefits brought about in each experiment in absolute terms, with respect to a set of dimensions and metrics that are recognizably relevant for that experiment.

Another way that could be explored is to tie the evaluation directly to the original requirements of the target system, and to the degree of fulfillment of those requirements that the adaptive solution is able to guarantee. Such an approach would help establishing a strong inter-dependence between the engineering of requirements specifications and the engineering of dynamic adaptation facilities (once again, the availability of comprehensive modeling capabilities could help, in bridging the two areas and in setting, maintaining and reasoning about that correlation).

Such an approach would also shift the issue of coming up with homogeneous means for the evaluation of dynamic adaptation from the solution domain to the problem domain. Within the definition of the problem domain of each application, it is feasible to denote the importance and the weight that each requirement has for that application (for example by exploiting requirements prioritization, traceability relationships, or other methods and tools used for analysis and evaluation in the field of requirements engineering [169]). Having established that, it may then be possible to assign a "value" to the ability by a dynamic adaptation solution to enforce the compliance of the application at run time with a given requirement.

Finally, another significant issue that remains in part open at this stage is that of meta-adaptation. A dynamic adaptation platform is in itself a complex, distributed software

system. While it strives to provide important features to its target system, such as self-configuration, self-optimization, self-healing and self-protection, it can itself suffer from problems and failures that may impact its ability to perform efficiently or event correctly. There is a clear need for the dynamic adaptation platform to be able to assess and tune itself while it runs, ideally without interrupting its supervision of the target system. Dynamic adaptations may apply to various roles and elements in the platform; some examples are the instantiation, withdrawal or tuning of sensors and gauges, or even the modification and update of decision policies and coordination plans.

To achieve meta-adaptation capabilities, the availability and semantic richness of behavioral models – applied this time to the platform itself and its possible operation – appears to be once more a crucial issue.

6.4 Comparison with the state of the art

The discussion below intends to highlight the differences and the original contribution of the research presented in this document with respect to a variety of other techniques and works that address the problem space of dynamic software adaptation in ways that are closely related to Workflakes. Therefore, this discussion concentrates primarily on approaches that show an explicit coordination focus.

Process-based software coordination

Once again, it is important to notice that it is its externalized stance that most strongly characterizes Workflakes. Often, in fact, automated solutions to software coordination and control present structural dependencies with respect to the subjects of their

coordination.

Some of those solutions can be seen as an evolution of built-in fault tolerance code. For example, [71] proposes a rule-based inference engine for decision support in application-level QoS assurance, which incorporates a coordination entity guiding a set of computational actuators. However, the coordinator and actuators must both be embedded with each target component. Solutions like that make it more difficult to define system-wide adaptations and limit the adaptations that can be carried out without rebuilding the target.

Another classic approach is that of an environment or middleware with native dynamic adaptation capabilities. Generic (i.e., not necessarily process-based) examples of dynamic adaptation middleware include Conic [3], Polylith [1], 2K / dynamicTao [2] and many others.

Also many works that employ process technology for software control and coordination adopt in fact a middleware-like approach, by exerting the coordination "from the inside", that is, on the target's own computations. For example, [72] introduces Containment Units, as modular process-based lexical constructs for defining how distributed applications may handle self-repair and self-reconfiguration. Containment Units define a hierarchy of processes that predicate on constraints and faults, and take action to handle faults within the defined constraints. The enactment of Containment Units is under the responsibility of a process engine that is integral to the system being adapted, and proceeds by directing changes on the target components, which by definition are process-aware.

PIE [10] is another example of a process-based middleware, which supports federations of components. PIE adds a control layer on top of a range of inter-component communication facilities. The control layer implements process guidance via handlers that react to and manipulate the communications exchanged by the components in a federation. Dynamic adaptation is thus limited to the reconfiguration of the service architectural connectors and is carried out by plugging in appropriate handlers, as directed by the process, which intrude in the normal course of computation of the target.

TCCS [73] has considerable similarities with Workflakes, since it employs its process engine to direct the work of analogous effector agents, to carry out the dynamic adaptation tasks. However, TCCS is the epitome of the middleware approach, since it is in charge of all interactions between the system components, even normal operations; that is, the target application simply does not exist independently from its process and agent-based framework.

In each dynamic adaptation middleware mentioned above, all service components need to be assembled from the start according to the middleware and its primitives. This not only poses a considerable barrier with respect to legacy software, but also introduces a very strong dependency between actors and subjects of dynamic adaptation. Furthermore, the spectrum and granularity of possible adaptations is effectively restricted by the set of primitives made available by the chosen middleware. A similar observation applies also to those works that exploit the characteristics of established middleware frameworks to facilitate certain aspects of

dynamic adaptation, such as BARK [4], which is limited to the EJB component model.

A particularly noteworthy effort is that presented in [180], since it regards the dynamic adaptation of distributed applications directly developed with the Cougaar infrastructure. That work considers large-scale logistics application running on hundreds of collaborating Cougaar instantiations, which are regarded as a community of distributed agents. The built-in workflow facilities of Cougaar are leveraged not only to puruse the goals of those logistics applications, but also – in combination with the resident monitoring facilities of the Cougaar infrastructure - to provide adaptive control of the operation of the various Cougaar agents. The goal is to optimize the overall performance of the community, trading-off some precision in the evaluation and production of the logistics plans for increased throughput, when necessary because of heavy computational load and environment conditions. That works thus presents the peculiar case of an internalized, workflow-based dynamic adaptation facility employed for the control of other workflow-based applications developed on the same platform.

In contrast to all of the internalized approaches above, Workflakes remains independent from any underlying computing framework and general with respect to the reach, granularity and kinds of dynamic adaptation that it can exert, since the target is fully disjoint from the dynamic adaptation engine.

The most similar process-based approach to the orchestration of dynamic adaptation (that we know of) may be Willow [31]. Willow proposes an architecture for the survivability of distributed applications, analogous to our vision of a superimposed

feedback loop. In particular, Willow can implement reactive as well as proactive dynamic adaptation policies, which are driven by codified architectural knowledge, and enacted via a process-based mechanism built upon the previous Software Dock (re-)deployment engine [36]. It appears, however, that Willow restricts itself to coarse-grained reconfigurations, such as replacing, adding and removing entire components, perhaps even composite substructures, from the target application, while presuming conventional embedded approaches for more local and refined adaptations. CHAMPS [183] is another system that employs process technology. It is noticeable because it attempts to automatically generate an adaptation workflow on the basis of a Request For Change (RFC) coming from an administration entity, which is typically operated by humans. To that end, CHAMPS includes a Task Graph Builder, which puts together single tasks and small fragments from a catalog, producing a sequence of tasks with precedence constraints. The generated concatenation is then consumed by the Planner & Scheduler of CHAMPS, which generates a slightly more complex workflow, and tries to maximize the degree of parallelism among tasks, taking in account precedence relationships, but also other aspects, like costs and Service Level Agreements (SLAs), imposed on the target system. The resulting workflow is translated into BPEL4WS for its enactment in a BPEL-compliant engine. The extension of the generative approach of CHAMPS to cover more sophisticated flow constructs is under investigation.

Alternatives for the coordination of dynamic adaptation

Among the various existing software coordination paradigms (see Section 2.3), the most common alternative approaches to fulfill the coordination role in dynamic adaptation seem to come from the fields of agent-based and rule-based systems.

Agents have been already discussed in Section 2.3 as a coordination paradigm in general, and in Section 2.4 with the purpose of highlighting their inter-relationships with process technology.

There are several examples of agent-based systems that are related to the theme of dynamic software adaptation. Some are concerned with the development of agent-based applications that are adaptive or autonomic in themselves, thus falling into the category of internalized dynamic adaptation. For example, [175] focuses on embedding within agent-based applications fault recovery features by design. DarX [176] focuses on the dynamic replication of those agents in a given community, whose capabilities are or become critical during the life span and for the work of the community. Anthill [170] implements adaptive behaviors within a large-scale community of autonomous agents; Messor [171] is an example of an Anthill application that provides load balancing for a Grid of computing elements implemented as Anthill agents.

Other works regard the usage of agent-based techniques to carry out dynamic adaptation on external, generic software applications and systems. For example, AUTONOMIA [118], employs a coordination model derived from tuple spaces for orchestrating mobile agents, which superintend to the self-healing and self-optimization of a distributed software system. Target system components must be

developed in accord with the AUTONOMIA middleware platform, which makes them internally and natively autonomic, and exposes handles for monitoring and actuation. However, the mobile agents exerting the adaptation, as well as their coordination facilities, remain external to the system, even if they operate on top of the same middleware. This configures a hybrid approach to the development of a full dynamic adaptation loop, which remains unsuitable for legacy target system, but promotes to a degree the separation of some autonomic concerns, such as decision policies and coordination plans. ABLE [119] by IBM is a component-oriented agent platform, in which each agent is composed of multiple AbleBeans (derived from standard JavaBeans), and is itself an AbleBean. Some of the AbleBeans may implement sensors and effectors to match and exploit any monitoring and actuation functionality exposed by a target system component the agent is deployed onto; others may provide analysis and control logic on top of the monitoring and actuation AbleBeans. A catalog of AbleBean components is provided to that end, which encapsulate a rich mix of techniques (such as, neural networks, rule bases, etc.) and a range of algorithms for decision-making, collection of monitoring / diagnostic data, and execution of effectors. In one typical ABLE architectural layout, agents overlay the target system, with one (or more) agent(s) co-located with each single target component, and implementing a mini-control loop that takes care of that component in isolation. As the complexity of the dynamic adaptation problems grows, other layers of agents can be added to provide an increasingly more sophisticated and global perspective on analysis, decision and coordination. With respect to coordination, ABLE seems to lean towards the implicit model of run-time negotiation

among agents, supported by a subsumption architecture inspired by the original work of Brooks [172].

Another agent-based autonomic platform is proposed in [173], which has two distinguishing traits: it is organized by multiple layers of agents with different responsibilities, like ABLE, and it has an explicit focus on architecture-level adaptations. To that end, it incorporates a full-fledged, dynamic architectural model that is exposed to the agents in the higher layer, which are devoted to decision-making. Intermediate-layer agents are directed from the higher-layer and manage the work of lower-layer agents, which implement the points of contact with the target system for monitoring and actuation. Interventions by the lower-layer agents are limited to the modification of the architectural layout, that is, adding, removing or replacing components, or modifying connectors; therefore they remain somewhat coarse-grained. Coordination-wise, plans are generated by the decision-making agents at the higher layer, and orchestrated by intermediate-layer agents by sending stimuli to lower-layer agents, which are completely reactive. It is however not clear what kind of coordination paradigm is employed to express and enact those plans.

A layered approach is also employed by Lira [182], which employs a hierarchical community of agents. The agent hierarchy maps to a structural breakdown of the target system in applications, hosts, and components within each host. Each agent has a local decision-maker, built with Petri Nets [204]. Agents at the lowest layer can decide only on local adaptations; agents at higher layers can also direct lower-layer agents to effect some adaptations. Adaptations are seen as atomic interventions chosen from a limited set, since Lira does not currently support the concept of multi-

step adaptations; however, the authors envision using a coordination paradigm – also based on Petri nets - in future developments.

There are also several works that use rule-based techniques (previously discussed in Section 2.3) for dynamic software adaptation. Rules of various kinds, such as ECA rules, have been commonly used to express management policies and support their automation to a degree (see for example [177]), even in traditional, human-intensive management systems; a rule-based representation of those policies is recommended also by the IETF [178].

Extending from there, rules can be conveniently used to specify and implement autonomic behavior within single components, such as in [71]. They can further be used to orchestrate multi-component adaptations. Autopilot [113], for instance, uses fuzzy logic rules within a close-control loop facility embedded in a computing Grid, for the optimization of the performance of parallel applications running on that Grid. The Autopilot system seems to coalesce in the rule base the decision and the coordination roles of a dynamic adaptation framework, although it is not clear what degree of coordination complexity can be achieved in that way. Similarly, also DIOS++ [114] works in the context of computing Grid optimization. The DIOS++ distributed rule base defines both the condition to be monitored by the dynamic adaptation loop, and the actions to be taken in response to those conditions. Multiple rule executors, co-located with the autonomic elements of the Grid to be adapted, work in parallel and can influence each other when firing rules. Therefore, DIOS++ has the potential to define a full-fledged coordination plan that spans the Grid. Both Autopilot and DIOS++ are evaluated with respect to the (reasonably limited)

overhead they impose onto the computing infrastructure of their targets, but not with respect to the management and/or performance benefits they bring about onto their target Grid computing environments or applications.

RUDDER (see [117]) aims at the construction of a de-centralized rule engine for the orchestration of dynamic adaptation policies on generic distributed computing applications. Rule processors (dubbed *rule agents*) work in a peer-to-peer fashion, and are distributed according to a layered architecture: the architecture includes some master rule agents at the overall application level that control the work of other rule agents, which are co-located with and, apparently, embedded into target system components. Therefore, RUDDER provides a hybrid solution to the orchestration of dynamic adaptation, which is partially externalized and partially internalized. RUDDER seems to currently be at an early stage, and various aspects are still not well specified, including, noticeably, the exact coordination semantics among peer rule agents.

Eos [179] employs a rule base for deciding upon and carrying out adaptations. The rule base contains ECA rules, augmented with additional knowledge that represents the *behavior implications* of those rules. A behavior implication defines in a declarative way the impact of firing the corresponding rule, in terms of observable characteristics of the target system, for example response time, security, throughput, availability, etc. In Eos, the decision on what rules must be fired to respond to a certain condition that requires dynamic adaptation is taken following a multi-dimensional evaluation of the likely impact of the rules' execution on those characteristics. Eos thus focuses primarily on sophisticated decision-making, while

coordinated, multi-step adaptations are not explicitly considered. However, the decision component can choose to concatenate multiple rules, because they provide a path that achieves the desired impact on the target system, while minimizing any undesired behavior implications.

7 Conclusions and future work

This work has investigated the use of process / workflow technology for the development of coordination facilities that can be used to orchestrate the dynamic adaptation of distributed software systems, in particular large-scale systems of (legacy) systems. This theme can be framed in the larger context of autonomic computing.

Those coordination facilities are intended to provide a core service and fulfill a critical role in an externalized platform, such as Kinesthetics eXtreme - KX, which aims at superimposing dynamic adaptation on pre-existing software systems, from the outside and without modifying those systems. Such a coordination role is instrumental for the transformation of decisions on what adaptations must be pursued in sequences of computational actions that actually effect the needed modifications on one or more elements taking part in the target system.

This work proposes a model for processes that are suited for the orchestration of dynamic adaptation: processes must reactively respond to triggers; they are fragmented and structured as task hierarchies; coordination constructs are maintained in inner nodes of the hierarchy, while leaf nodes map to actual units of work to be effected on the target system; processes need to incorporate suitable concepts to handle exceptions, in order to take care of internal contingencies, and must support compensations, in case of internal as well as external contingencies.

The characteristics listed above have guided the design and development of Workflakes, a workflow engine specialized for the fully-automated orchestration of

dynamic software adaptation; this work reports on the two iterations of Workflakes development completed to this date in accord to that design.

Workflakes represents one of the first process enactment engines applied to the orchestration of dynamic adaptation, whose use and effectiveness has been validated in a variety of applications. Workflakes has been experimented with in a number of case studies (as part of KX and on its own), including and industrial-grade application. The case studies reported here address a range of different application domains, with diverse requirements and characteristics. It has been applied to many of the major concerns of autonomic computing, including the self-configuration, self-optimization and self-healing of software. The target systems subject to the Workflakes controller can widely vary in a number of respects, such as their distribution, the computing layers where adaptation takes place, and their timeliness requirements. The range of adaptations supported varies in granularity from architecture-level reconfigurations to the tuning of functioning parameters within individual software modules.

Taken together, the presented case studies contribute to validate the underlying concepts as well as the design and implementation of Workflakes and KX. Quantitative and qualitative results collected in the case studies show significant benefits in various areas of importance to autonomic computing, such as management and administration savings, improvement of runtime quality aspects and the enforcement of expected system behavior. Furthermore, all of those benefits can be achieved with minimal or no impact on the development of the target system, as well

as with little effort devoted to the development of dynamic adaptation features, that must be customized for each application.

As a conclusion, this research demonstrates the suitability and effectiveness of a process-based approach to the orchestration of dynamic adaptation, and – in a larger context - the feasibility of exerting externalized dynamic adaptation with platforms that choose process technology for their coordination role.

This work can be pursued further in a number of directions. As for research on the process-based coordination of dynamic software adaptation in general, the outstanding problems discussed in Section 6.3 may represent a valid agenda, which can take advantage of some of the results of this work as a starting point. Among those problems, there are methods and techniques for the engineering of open-ended dynamic adaptation processes, and the dynamic generation of processes on the basis of the situational knowledge of the run state of the system, in order to be able to respond also to fully unexpected conditions. Those two problems are intertwined, and both seem also strongly related to other two open issues: the ability to capture, express and reason about formal knowledge that predicates not only upon the structure, but also the dynamics of the target system, including the adaptations that are or are not valid under certain conditions; and the selection of the right granularity level for the fragmentation of dynamic adaptation processes, to make possible the construction of a repertoire of process fragments that can be composed on the fly in an open-ended way, and that may cover a wide spectrum of dynamic adaptation needs for the target system at hand.

Another open issue regards criteria and techniques for the comparative evaluation of dynamic adaptation solutions, in general, as well as with respect to their various main roles, including coordination. As the work on autonomic computing approaches and systems progresses, and given their wide field of applicability, an agreed-upon set of evaluation guidelines and practices, if not a common framework will become increasingly necessary.

Finally, it is necessary to investigate the issue of meta-adaptation, that is, how dynamic adaptation facilities can keep themselves in check - while at the same time controlling an external target system - in order to preserve as well as optimize their functionality and performance.

With respect to the specific advancement of Workflakes and in the context of the KX platform, there are multiple aspects that can be the subjects of future work. One is concerned with meta-adaptation: Workflakes controllers can be used to modify on the fly the monitoring and diagnostic layers of KX, as well as tuning single sensors or gauges. Following up on the ideas sketched in Section 6.3, another future development regards a fuller integration and more extensive experimentation of the platform with ADL-based behavioral models and the corresponding tools, which has the potential to bring about architecture-driven generation of the dynamic adaptation processes. Workflakes will also be experimented with in other application domains, in order to better understand the usability limits and further evaluate the extent of the benefits of the approach and the tool.

8 Bibliography

[1] C.R. Hofmeister, and J.M. Purtilo, *Dynamic Reconfiguration in Distributed Systems: Adapting Software Modules for Replacement*, in 13th International Conference on Distributed Computing Systems, Pittsburgh, Pa., USA, May 25-28, 1993.

[2] F. Kon, R. Campbell, M.D. Mickunas, K. Nahrstedt, and F.J. Ballesteros. *2K, A Distributed Operating System for Dynamic Heterogeneous Environments*, in 9th IEEE International Symposium on High Performance Distributed Computing, Pittsburgh, Pa., USA, August 1-4, 2000.

[3] J. Magee, J. Kramer, and M. Sloman. *Constructing Distributed Systems in Conic,* IEEE Transactions on Software Engineering, 15(6):663-675, June 1989.

[4] M.J. Rutherford, K. Anderson, A. Carzaniga, D. Heimbigner, and A.L. Wolf, *Reconfiguration in the Enterprise JavaBean Component Model*, in IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, June 2002.

[5] P.N. Gross, S. Gupta, G. E. Kaiser, G.S. Kc, and J.J. Parekh, *An Active Events Model for Systems Monitoring*, in Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 12-14, 2001, available at: http://www.psl.cs.columbia.edu/ftp/psl/CUCS-011-01.pdf.

[6] G. Kaiser, P. Gross, G. Kc, J. Parekh, and G. Valetto, *An Approach to Autonomizing Legacy Systems*, in Workshop on Self-Healing, Adaptive and Self-MANaged Systems (SHAMAN 2002), New York, NY, USA, June 23, 2002.

[7] G. Valetto, G. Kaiser, and G.S. Kc, *A Mobile Agent Approach to Process-based Dynamic Adaptation of Complex Software Systems*, in 8th European Workshop on Software Process Technology, Witten, Germany, June 19-21, 2001. http://www.psl.cs.columbia.edu/ftp/psl/CUCS-001-01.pdf.

[8] G. Valetto, and G. Kaiser, *Using Process Technology to Control and Coordinate Software Adaptation*, in 25th International Conference on Software engineering (ICSE 2003), Portland, Or., USA, May 3-10, 2003.

[9] G. Valetto, *Process-Orchestrated Software: Towards a Workflow Approach to the Coordination of Distributed Systems*, Ph.D. Thesis Proposal, Columbia University, Department of Computer Science, Technical Report # CUCS-016-00, May 2000.

[10] G. Cugola., P.Y. Cunin, S. Dami, J. Estublier, A. Fuggetta, F. Pacull, M. Riviere, and H. Verjus, *Support for Software Federations: The Pie Platform* in 7th

European Workshop on Software Process Technology, Kaprun, Austria, February 22-25, 2000.

[11] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, *Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems*, in 5th Annual International Workshop on Active Middleware Services (AMS 2003), Seattle, Wa. USA, June 25, 2003.

[12] The World Wide Web Consortium, *Web Services Activity*, available at: http://www.w3.org/2002/ws/

[13] L. Lee, H. S. Nwana, N. R. Jennings, *Co-ordination in Multi-Agent Systems*, in H. S. Nwana and N. Azarmi (eds.), Software Agents and Soft Computing, Lecture Notes on Artificial Intelligence no. 1198, pp. 42-58, 1997.

[14] M. N. Huhns and M. P. Singh (eds.), *Internet-based Agents: Applications and infrastructure*, special issue of IEEE Internet Computing, 1(4), July/August 1997.

[15] K. J. Werkman, *Knowledge-based Model of Negotiation Using Shareable Perspectives*, in 10th International Workshop on Distributed Artificial Intelligence, Bandera, Tx. USA, October 23-27, 1990.

[16] S. Bussmann and J. Muller, *A negotiation Framework for Co-Operating Agents*, in CKBS-SIG, pp. 1-17, University of Keele, 1992.

[17] H. S. Nwana and M. Wooldridge, *Software Agent Technologies*, in H. S. Nwana and N. Azarmi eds. Software Agents and Soft Computing, in Lecture Notes on Artificial Intelligence, no. 1198, pages 59-77, 1997.

[18] T. Finin, R. Fritzon, D. McKay, R. McEntire, *KQML as an Agent Communication Language*, in 3rd International Conference on Information and Knowledge Management, Gaithersburg, Ma., USA, November 29-December 4, 1994.

[19] A. G. Cass, B. Staudt Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr., A. Wise, *Little-JIL/Juliette: A Process Definition Language and Interpreter*, in 22nd International Conference on Software Engineeering (ICSE 2000), Limerick, Ireland, June 4-11, 2000.

[20] J. Estublier, M. Amiour and S. Dami, *Building a Federation of Process Support Systems*, in Work Activity Coordination and Cooperation conference (WACC'98); San Francisco, Ca., USA, February 22-26, 1998.

[21] I. Ben-Shaul and G. E. Kaiser, *A Paradigm for Decentralized Process Modeling,* Kluwer Academic Publishers, Boston MA, 1995.

[22] J. Grundy, M. Apperley, J. Hosking, 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.

[23] IBM alphaWorks, *BPWS4j*, http://www.alphaworks.ibm.com/tech/bpws4j

[24] TIBCO, *TIBCO Business Process Management*, http://www.tibco.com/resources/solutions/products/bpm_datasheet.pdf

[25] Cougaar.org, *Cougaar Open Source Agent Architecture*, http://www.cougaar.org/

[26] David Jensen, Yulin Dong, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton, Jr., and Alexander Wise, *Coordinating Agent Activities in Knowledge Discovery Processes*, in Work Activities Coordination and Collaboration Conference (WACC 99), pages 137-146, San Francisco, Ca., USA, February 22 1999.

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

[28] 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 Technology of Object-Oriented Languages and Systems USA Conference (TOOLS-USA 99), Santa Barbara, CA, USA, July 30- August 3, 1999.

[29] J. E. Cook and A. L. Wolf. *Software process validation: Quantitatively measuring the correspondence of a process to a model using event-based data*, ACM Transactions on Software Engineering and Methodology, 8(2): 147-176, Apr 1999.

[30] P. Horn, *Autonomic Computing: IBM's Perspective on The State of Information Technology*, at Agenda 2001 conference, Scottsdale, Az., USA, October 15, 2001, http://www.research.ibm.com/autonomic/manifesto/agenda2001_p1.html

[31] J. Knight, D. Heimbigner, A. Wolf, A. Carzaniga, J. Hill, P. Devanbu, M. Gertz, *The Willow Architecture: Comprehensive Survivability for Large-Scale Distributed Applications*, in Intrusion Tolerance Workshop, the International Conference on Dependable Systems and Networks (DSN-2002), Washington, DC, USA, June 2002.

[32] A.G. Ganek, T.A. Corbi, *The Dawning of the Autonomic Computing Era,* IBM Systems Journal, 42(1): 5-18, January-March 2003.

[33] InstallShield Corp, *InstallShield – Software Installation and Migration for System Adminstrators and Developers*, available at: http://www.installshield.com

[34] V. Guoer, A. Bahuguna, O. Egungbohun, P. Field, C. Horwedel, and R. Kannaujia, *Implementing Automated Inventory Scanning and Software Distribution After Auto Discovery*, IBM Red Book, May 9, 2003.

[35] Marimba Inc., *Marimba Embedded Management - Creating Self-Updating Appliances and Devices,* Marimba White Paper, Mountain View, Ca., USA, 2001, http://www.marimba.com/products/datasheets/Embedded-wp-april-2001.pdf

[36] R.S. Hall, D. Heimbigner, and A.L. Wolf*, A Cooperative Approach to Support Software Deployment Using the Software Dock*, in International Conference on Software Engineering (ICSE'99)*,* Los Angeles, Ca., USA, May 1999.

[37] C. Montangero, *The Process in the Tool Syndrome: Is It Becoming Worse?,* in 9th International Software Process Workshop, Arlie, Va., USA, October 1994.

[38] C. Poellabauer, H. Abbasi, and K. Schwan, *Cooperative Run-time Management of Adaptive Applications and Distributed Resources*, in ACM Multimedia, Juan-les-Pins, France, December 1-6, 2002.

[39] J. A. Stankovic and K. Ramamritham, *The Spring Kernel: A new paradigm for real-time systems*, IEEE Software, 8(3): 62–72, May 1991.

[40] D. Roşu, K. Schwan, S. Yalamanchili, and R. Jha, *On Adaptive Resource Allocation for Complex Real-Time Applications*, in IEEE Real-Time Systems Symposium, San Francisco, Ca., USA, December 3-5, 1997.

[41] T. F. Abdelzaher, E. M. Atkins, and K. G. Shin, *QoS negotiation in real-time systems and its application to automated flight control*, in IEEE Real-Time Technology and Applications Symposium, Montreal, Canada, June 1997.

[42] A.F. Garcia, C.J. Pereira de Lucena, F. Zambonelli, A. Omicini, and J. Castro (Eds.): *Software Engineering for Large-Scale Multi-Agent Systems, Research Issues and Practical Applications*, Lecture Notes in Computer Science 2603, Springer 2003

[43] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, *SETI@home: An Experiment in Public-Resource Computing*, Communications of the ACM, 45(11):56-61, November 2002.

[44] A. Reinefeld, and F. Schintke, *Concepts and Technologies for a Worldwide Grid Infrastructure*, in 8th International Euro-Par Conference, Paderborn, Germany, August 27-30, 2002.

[45] J. Kiessel, J. Beard ,and P. Nielsen, *Failure Recovery: A Software Engineering Methodology for Robust Agents,* in 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2002), Orlando, Fl., USA, May 19, 2002.

[46] Z. Guessoum, J.-P. Briot, and S. Charpentier, *Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems*, in Proceedings of the 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2002), Orlando, Fl., USA, May 19, 2002.

[47] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, J.R.D. Dyson, L. Zhao, and G.R. Nudd, *Performance-based Middleware Services for Grid Computing*, in 5th International Workshop on Active Middleware Services (AMS 2003), Seattle, Wa. USA, June 2003.

[48] BBN Technologies, *Advanced Logistics Project*, http://www.bbn.com/abs/alp.html

[49] Defense Advanced Research Project Agency (DARPA), *UltraLog Program*, http://www.ultralog.net/

[50] J. Cao, S.E. Jarvis, G.R. Nudd, and S. Saini, *GridFlow: Workflow Management for Grid Computing*, in 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), Tokyo, Japan, May 12-14, 2003.

[51] T.E. Bihari, K. Schwan*, Dynamic Adaptation of Real-Time Software*, ACM Transactions on Computer Systems, 9(2):143-174, May 1991.

[52] D. Rosu, K. Schwan, S. Yalamanchili, FARA - *A Framework for Adaptive Resource Allocation in Complex Real-Time Systems*, in 4th IEEE Real-Time Technology and Applications Symposium, Denver, Co., USA, June 1998.

[53] P. Oriezy, M.M. Gorlick, R.N. Taylor, D. Heimbinger, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, and A.L. Wolf, *An Architecture-Based Approach to Self-Adaptive Software*, IEEE Intelligent Systems 14(3): 54-62, May/June 1999.

[54] B. Schmerl, and D. Garlan, *Exploiting Architectural Design Knowledge to Support Self-repairing Systems*, in 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, July 2002.

[55] W. Gu, G. Eisenhauer, and K. Schwan, *Falcon: On-line Monitoring and Steering of Parallel Programs*, in Concurrency: Practice and Experience, 10(9): 699-736, August 1988.

[56] J. Salasin, *Dynamic Assembly for System Adaptability, Dependability, and Assurance (DASADA)*, http://www.darpa.mil/ipto/programs/dasada/index.htm

[57] G. Kaiser, *Autonomizing Legacy Systems*, invited talk at the Almaden Institute Symposium on Autonomic Computing, April 10-12 2002, http://www.almaden.ibm.com/institute/pdf/KaiserGail.pdf

[58] B. Balzer, *Probe Technology Adaptor Design*, February 2001, available at: http://schafercorp-ballston.com/dasada/2001WinterPI/ProbeTechnologyAdaptorDesign.ppt

[59] D. Garlan, B. Schmerl, and J. Chang, *Using Gauges for Architecture-Based Monitoring and Adaptation*, in Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 12-14, 2001.

[60] E. Kasten, P. K. McKinley, S. Sadjadi, and R. Stirewalt, *Separating introspection and intercession in metamorphic distributed systems*, in IEEE Workshop on Aspect-Oriented Programming for Distributed Computing, Vienna, Austria, July 2-5, 2002.

[61] E.M. Dashofy, A. van der Hoek, and R.N. Taylor, *An Infrastructure for the Rapid Development of XML-based Architecture Description Languages*, in 24th International Conference on Software Engineering, Orlando, Fl. USA, May 2002.

[62] D. Garlan, S. Cheng, and B. Schmerl, *Increasing System Dependability through Architecture-based Self-repair*, in 2nd ICSE Workshop on Software Architectures for Dependable Systems (WADS 2003), Portland, Or., USA, May 2003.

[63] G. Heineman, *Active Interface Development Environment (AIDE)*. http://www.cs.wpi.edu/~heineman/dasada/

[64] P. Pazandak, and D. Wells, *ProbeMeister – Distributed Runtime Software Instrumentation*, in 1st workshop on Unanticipated Software Evolution (USE 2002), Malaga, Spain, June 10-14, 2002, http://www.objs.com/ProbeMeister/paper/020523-probemeister.pdf

[65] R.M. Balzer, and N.M. Goldman, *Mediating Connectors: A Non-ByPassable Process Wrapping Technology,* DARPA Information Survivability Conference & Exposition, Vol. 2, January 2000.

[66] D.L. Wells, and P.Pazandak, *Taming Cyber Incognito – Tools for Surveying Dynamic/Reconfigurable Software Landscapes*, in Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia, December 12-14, 2001.

[67] A. Rocha, G. Valetto, E. Paschetta, and S. Heikkinen, *Continuous On-Line Validation of Web Services*, in International Conference on Electronic Publishing (ELPUB 2002), Karlovy Vary, Czech Republic, November 6-8, 2002.

[68] P. Deussen, G. Valetto, G. Din, T. Kivimaki, S. Heikkinen, and A. Rocha, *Continuous On-Line Validation for Optimized Service Management*, in EURESCOM Summit 2002, Hiedelberg, Germany, October 21-24, 2002.

[69] *Tree and Tabular Combined Notation, version 3*, ITU-T Recommendation Z.140, 2001, available at: http://www.itu.int/ITU-T/studygroups/com10/languages/Z.140_0701_pre.pdf

[70] G. Kaiser, A. Stone, and S. Dossick, *A Mobile Agent Approach to Lightweight Process Workflow*, in International Process Technology Workshop (IPTW'99), Grenoble, France, September, 1-3 1999, available at: http://www.psl.cs.columbia.edu/ftp/psl/CUCS-021-99.pdf.

[71] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. Bauer, *Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework*, in 3rd IEEE International Workshop on Policies for Distributed Systems and Networks, Bristol, U.K., January 29-31,2001.

[72] J.M. Cobleigh, L.J. Osterweil, A. Wise, and B. Staudt Lerner, *Containment Units: A Hierarchically Composable Architecture for Adaptive Systems*, in 10th International Symposium on the Foundations of Software Engineering (FSE 10), Charleston, Sc. USA, November 20-22, 2002.

[73] S.K Shirvastava, L. Bellissard, D. Feliot, M. Herrmann, N. De Palma, and S.M. Wheater, *A Workflow and Agent based Platform for Service Provisioning*, in 4th IEEE/OMG International Enterprise Distributed Object Computing Conference, Makuhari, Japan, September 25-28, 2000.

[74] J. Liu, B. Li, and Y. Zang, *Adaptive Video Multicast over the Internet*, IEEE Multimedia, 10(1):22-33, January-March, 2003.

[75] S. McCanne, V. Jacobson, and M. Vetterli, *Receiver-Driven Layered Multicast*, in ACM SIGCOMM'96, Stanford, Ca. USA, August 26-30, 1996.

[76] W. Li, *Overview of the Fine Granularity Scalability in MPEG-4 Video Standard*, IEEE Transactions on Circuits and Systems for video Technology, 11(3):301-317, March 2001.

[77] L. Herger, K. Iwano, P. Pattnaik, J.J. Ritsko, and A.G. Davis (eds.), *Autonomic Computing*, IBM Systems Journal, 42(1), January-March 2003.

[78] R. Haas, P. Droz, and B. Stiller, *Autonomic Service Deployment in Networks*, IBM Systems Journal 42(1):150–164, January-March 2003.

[79] V. Markl, G. M. Lohman, and V. Raman, *LEO: An Autonomic Query Optimizer for DB2*, IBM Systems Journal 42(1): 98–106, January-March 2003.

[80] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, P. McKenney, M. Ostrowski, B. Rosenburg, M. Stumm, and J. Xenidis, *Enabling Autonomic Behavior in Systems Software with Hot Swapping*, IBM Systems Journal 42(1):60–76, January-March 2003.

[81] K. Whisnant, Z. T. Kalbarczyk, and R. K. Iyer, *A System Model for Dynamically Reconfigurable Software*, IBM Systems Journal 42(1):45–59, January-March 2003.

[82] D. M. Yellin, *Competitive Algorithms for the Dynamic Selection of Component Implementations*, IBM Systems Journal, 42(1):85–97, January-March 2003.

[83] A. Abbondanzio, Y. Aridor, O. Biran, L. L. Fong, G. S. Gold-szmidt, R. E. Harper, S. M. Krishnakumar, G. Pruett, and B.-A. Yassur, *Management of Application Complexes in Multitier Clustered Systems*, IBM Systems Journal 42(1):189–195, January-March 2003.

[84] B.N. Bershad, S. Savage, P. Pardyn, E.G. Sirer, M.E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, *Extensibility, Safety and Performance in the SPIN Operating System*, in ACM Symposium on Operating System Principles, Copper Mountain resort, Co., USA, December 1995.

[85] M. Seltzer, and C. Small, *Self-monitoring and Self-adapting Operating Systems*, in 6th Workshop on Hot Topics in Operating Systems, Cape Cod, Ma., USA, May 5-6, 1997.

[86] IBM Corporation - Research Division, *The K42 Project*, http://www.research.ibm.com/K42.

[87] *RM 2000 – Workshop on Reflective Middleware*, IBM Palisades Executive Conference Center, NY, USA, April 7-8, 2000, available at: http://www.comp.lancs.ac.uk/computing/RM2000/

[88] *RM2003 - the 2nd Workshop on reflective and Adaptive Middleware*, Rio de Janeiro, Brazil, June 17, 2003, available at: http://www.cs.wustl.edu/~corsaro/RM2003/index.html

[89] G. Coulson, and N. Parlavantzas (eds.), *Reflective Middleware*, IEEE Journal on Distributed Systems Online, http://dsonline.computer.org/middleware/RM.htm

[90] G.I Zachary, A.Y. Levy, D.S. Weld, D. Florescu, M. Friedman, *Adaptive Query Processing for Internet Applications*, IEEE Data Engineering Bulletin, 23(2), June 2000,

[91] R.A. Hankins, J.M. Patel, *Data Morphing: An Adaptive, Cache-Conscious Storage Technique*, in 29th Conference on Very Large Databases (VLDB 2003), Berlin, Germany, September 9-12, 2003.

[92] S. Parastatidis, and P. Watson, *NEReSC Core Grid Middleware*, in UK e-Science All-hands Meeting 2003, Nottingham, UK, September 2-4, 2003.

[93] F. Kon, F. Costa, R. Campbell, and G. Blair, *The Case for Reflective Middleware*, Communications of the ACM, 45(6): 33-38, June 2002

[94] The World Wide Web Consortium, *Web Services Choreography Working Group,* http://www.w3.org/2002/ws/chor/

[95] S.Thatte (ed.), *Business Process Execution Language for Web Services Version 1.1*, May 2003, ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf

[96] N. Carriero and D. Gelernter, *Coordination Languages and their Significance*, Communications of the ACM,35(2):97-107, February 1992.

[97] D. Gelernter, *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems, 7(1): 80-112, Jan. 1985.

[98] N. Carriero and D. Gelernter, *Linda in Context*, Communications of the ACM, 32(4), April 1989.

[99] H.P. Nii, *Blackboard Systems Part One,* AI Magazine, 7(2): 38-53, 1986.

[100] H.P. Nii, *Blackboard Systems Part Two,* AI Magazine, 7(3): 82-106, 1986.

[101] Sun Microsystems, Inc., *JavaSpaces Specification*, Technical Report, July 1998.

[102] G. A. Papadopolous and F. Arbab, *Coordination Models and Languages*, in M.V. Zelkowits (ed.), Advances in Computers: the Engineering of Large Systems, Vol. 46, Academic-Press, The Netherlands, 1998.

[103] N Medvidovic and R. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, 26(1), January 2000.

[104] M. Shaw, *Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status*, in ICSE Workshop on Studies of Software Design, Baltimore, Ma., USA, May 17-18, 1993.

[105] L. Brownston, R. Farrell, E. Kant, and N. Martin *Programming Expert Systems in OPS5: An Introduction to Rule-based Programming*, Addison-Wesley, Reading, Ma., USA, 1985.

[106] G. Riley, and J.C. Giarratano, *CLIPS Reference Manual*, PWS Publishing Company, 1998.

[107] J.M. Andreoli, S. Freeman, and R. Pareschi, *The Coordination Language Facility: Coordination of Distributed Objects*, Theory and Practice of Object Systems, 2(2): 77-94, 1996.

[108] N.S. Barghouti, *Supporting Cooperation in the Marvel Process-Centered SDE*, in 5th ACM SIGSOFT Symposium on Software Development Environments, Tyson's Corner VA., USA, December 1992.

[109] E.P. Katz, *A Multiple Rule Engine-Based Agent Control Architecture*, in the 6th International Conference of Intelligent Engineering Systems (INES 2002), Opatija, Croatia, May 26-28, 2002.

[110] FIPA, *Foundation for Intelligent Physical Agents*, http://www.fipa.org.

[111] B.N. Grosof, D.W. Levine, H.Y. Chan, C.J. Parris, and J.S. Auerbach, *Reusable Architecture for Embedding Rule-based Intelligence in Information Agents*, in Workshop on Intelligent Information Agents, ACM Conference on Information and Knowledge Management (CIKM-95), Baltimore, Ma. USA, December 1995.

[112] J.P. Bigus, *The Agent Building and Learning Environment*, in 4th International Conference on Autonomous Agents, Barcelona, Catalonia, Spain, June 3-7, 2000.

[113] R. Ribbler, J. Vetter, and H. Simitci, *Autopilot: Adaptive Control of Distributed Applications*, in 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, Il., USA, July 1998.

[114] H. Liu, and M. Parashar, *DIOS++: A Framework for Rule-based Autonomic Management of Distributed Scientific Applications*, in 9th International Euro-Par Conference, Klagenfurt, Austria, August 2003.

[115] J. L. Austin, *How to Do Things with Words*, Cambridge, Ma., USA, Harvard University Press, 1962.

[116] J. R. Searle, *Speech Acts*, Cambridge, Ma., USA, Cambridge University Press, 1969.

[117] M. Agarwal, V. Bhat, Z. Li, H. Liu, B. Khargharia, V. Matossian, V.Putty, C. Schmidt, G. Zhang, S. Hariri and M. Parashar *AutoMate: Enabling Autonomic Applications on the Grid*, in 5th International Active Middleware Services Workshop (AMS2003), Seattle, Wa., USA, June 2003.

[118] S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, S. Rao, *AUTONOMIA: An Autonomic Computing Environment,* in 22nd International Performance Computing and Communications Conference, Phoenix, Az., USA, April 9-11, 2003.

[119] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao, *ABLE: A toolkit for building multiagent autonomic systems*, IBM Systems Journal, 41(3), July-September 2002.

[120] P. Ciancarini and A. L. Wolf, *Foreword to the Proceedings of the 3rd International Conference on Coordination Languages and Models (COORDINATION 99)*, Amsterdam, The Netherlands, April 26-28, 1999.

[121] A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. *Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies,* UC Berkeley Computer Science Technical Report UCB//CSD-02-1175, March 15, 2002.

[122] G. Kaiser, J. Parekh, P. Gross, and G. Valetto, *Retrofitting Autonomic Capabilities onto Legacy Systems*, Technical Report CUCS-026-03, Columbia University, Department of Computer Science, October 2003, available at: http://www.cs.columbia.edu/~library/TR-repository/reports/reports-2003/cucs-026-03.pdf.

[123] R. Khalaf, N. Mukhi, and S. Weerawarana, *Service-Oriented Composition in BPEL4WS*, in WWW 2003 conference, Budapest, Hungary, May 2003, available at: http://www2003.org/cdrom/papers/alternate/P768/choreo_html/p768-khalaf.htm.

[124] A. Shah and G. Kaiser, D*ecentralized Information Spaces for Composition and Unification of Services*, in Object-Oriented Web Services Workshop (OOWS), OOPSLA Conference, Vancouver, Canada, November 2002.

[125] A. Lazovik, M. Aiello, and M. Papazoglu, *Planning and monitoring the execution of web service requests,* in 1st International Conference on Service Oriented Computing (ICSOC), Trento, Italy, November 2003.

[126] E. Tzilla, R.E. Filman, A. Bader, *Aspect-oriented programming: Introduction,* in Communication of the ACM, 44(10):29-32, October 2001.

[127] D.C. Schmidt, R. Bector, D. Levine, S. Mungee, and G. Parulkar, *An ORB end system architecture for statically scheduled real-time applications*, in IEEE Workshop on Middleware for Real-Time Systems and Services, San Francisco Ca., USA, December 2-5, 1997.

[128] D. Harrington, R. Presuhn, and B. Wijnen, *An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks*, Internet RFC 3411, December 2002, available at: http://www.ietf.org/rfc/rfc3411.txt

[129] Sun Microsystems, Inc., *JSR-000003 Java™ Management Extensions (JMX™) specifications, Maintenance Release 2*, available at: http://jcp.org/aboutJava/communityprocess/final/jsr003/index3.html

[130] Distributed Management Task Force, Inc, *Web-Based Enterprise Management (WBEM) Initiative*, http://www.dmtf.org/standards/wbem

[131] Microsoft Corp., *Windows Management Interface (WMI) Reference*, available at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wmisdk/wmi/wmi_reference.asp

[132] S. Robertson, E.V. Siegel, M. Miller, and S.J. Stolfo, *Surveillance Detection in High Bandwidth Environments*, in DARPA DISCEX III Conference, Washington DC. USA, April 22-24, 2003

[133] SANS, *What is Host-Based Intrusion Detection?,* Intrusion Detection FAQ, http://www.sans.org/resources/idfaq/host_based.php.

[134] A. Cichocki, A. Helal, M. Rusinkiewicz, and D. Woelk. *Workflow and Process Automation—Concepts and Technology*, Kluwer Academic Publishers, 1998.

[135] D. Heimbigner, *The ProcessWall: A Process Server Approach to Process Programming*, in 5th ACM/SIGSOFT Conference on Software Development Environments, Washington, D.C., USA, December 9-11, 1992.

[136] 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 Automated Software Engineering Conference, Grenoble, France, September 11-15, 2000.

[137] S. Ahuja, N. Carriero and D. Gelernter, *Linda and Friends*, IEEE Computer 19(8): 26-34, 1986.

[138] R. Prieto-Diaz and J.M. Neighbors, *Module Interconnection Languages*. Journal of Systems and Software, 6(4): 307-334, November 1986.

[139] F. DeRemer and H.H. Kron, *Programming-in-the-large versus Programming-in-the-small*, IEEE Transactions on Software Engineering, 2(2):80-86, June 1976.

[140] . Wiederhold, P. Wegner, and S. Ceri, *Toward Megaprogramming: a paradigm for Component-Based Programming*, Communications of the ACM, 35(11): 89-99, November 1992.

[141] D.E. Perry and A.L. Wolf, *Foundations for the Study of Software Architectures*, ACM SIGSOFT Software Engineering Notes, 17(4):40-52, October 1992.

[142] D. Garlan and M. Shaw, *An Introduction to Software Architecture*, Advances in Software Engineering and Knowledge Engineering, vol. 2, December 1993.

[143] H. S. Nwana and D. T. Ndumu, *An Introduction to Agent Technology*, in H. S. Nwana and N. Azarmi eds. Software Agents and Soft Computing, Lecture Notes in Artificial Iintellingence no. 1198, pages 3-26, 1997.

[144] J. Widom, and S. Ceri. *Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann Publishers Inc., 1996.

[145] K.R. Dittrich, S. Gatziu, and A. Geppert. *The Active Database Management System Manifesto: a Rulebase of ADBMS Features,* in 2nd International Workshop on Rules in Database Systems, RIDS'95, Glyfada, Athens, Greece, September 1995.

[146] *COORDINATION – International Conference on Coordination Models and Languages*, http://music.dsi.unifi.it/coordination/default.htm

[147] R. Sterritt*, Discovering Rules for Fault Management*, in IEEE International Conference on the Engineering of Computer Based Systems (ECBS), Washington DC., USA, April 17-20, 2001.

[148] BEA Systems, Inc. *Introducing WebLogic Integration*, available at: http://edocs.bea.com/wli/docs81/overview/index.html

[149] TIBCO Software, Inc., *TIBCO BusinessWorks*, http://www.tibco.com/software/business_integration/businessworks.jsp

[150] I. Z. Ben-Shaul and G. E. Kaiser, *Federating Process-Centered Environments: the Oz Experience*, Journal of Automated Software Engineering, 5(1):97-132, January 1998.

[151] F. Leymann, *Web Services Flow Language (WSFL 1.0)*, May 2001, available at: http://www-306.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

[152] S. Thatte, *XLANG: Web Services for Business Process Design*, June 2001, available at: http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm

[153] ebXML Business Process Project Team, *ebXML Business Process Specification Schema*, available at: http://www.ebxml.org/specs/ebBPSS.pdf

[154] W.M.P. van der Aalst, A.H.M. Ter Hofstede, B. Kiepuszewski, and A.P. Barros, *Workflow Patterns*, Journal of Distributed. Parallel Databases, 14(1):5-51, July 2003.

[155] P.Wohed, W.M.P. van der Aalst, M.Dumas, and A.M. ter Hofstede, *Pattern Based Analysis of BPEL4WS*, QUT Technical Report FIT-TR-2002-04, Queensland University of Technology, Brisbane, Australia, 2002.

[156] A. Carzaniga, D.S. Rosenblum, and A.L. Wolf, *Design and Evaluation of a Wide-Area Event Notification Service*, ACM Transactions on Computer Systems, 19(3):332-383, August 2001.

[157] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps, *Content-based routing with Elvin4*. in Australian UNIX and Open Systems User Group Winter Conference (AUUG2K), Canberra, Australia, June 28-30, 2000.

[158] G. Piccinelli, and L. Mokrushin, Dynamic *e-Service Composition in DySCo*, in 21st International Conference on Distributed Computing Systems Workshops (ICDCSW '01), Mesa, Az., USA, April 16 - 19, 2001.

[159] A. Keromytis, J. Parekh, P.N. Gross, G. Kaiser, V. Misra, J. Nieh, D. Rubenstein, and S. Stolfo, *A Holistic Approach to Service Survivability*, in 1st ACM Workshop on Survivable and Self-Regenerative Systems, Fairfax, Va., USA, October 31, 2003.

[160] BBN Technologies, *Cougaar Developers' Guide*, available at: http://cougaar.org/docman/view.php/17/57/CDG_11_0.pdf

[161] D.L. McGuinness, and F. van Harmelen (eds.), *OWL Web Ontology Language Overview*, a W3C Recommendation, February 2004, available at http://www.w3.org/TR/owl-features/

[162] Sun Microsystems, Inc., *Java Native Interface Specifications*, available at: http://java.sun.com/j2se/1.4.2/docs/guide/jni/spec/jniTOC.html

[163] BEA Systems, Inc. *Introduction to WebLogic Server and WebLogic Express*, available at: http://edocs.bea.com/wls/docs81/intro/index.html

[164] C. Gage, *Writing Custom Advisors for IBM Network Dispatcher – A simple way to enhance load blancing*, February 2001, available at: http://www-106.ibm.com/developerworks/library/ibm-cust/?dwzone=ibm

[165] Sun Microsystems, Inc, *Jini Network Technology*, http://www.sun.com/software/jini/

[166] D. Garlan, R.T. Monroe, and D. Wile, *Acme: Architectural Description of Component-Based Systems*, in G.T. Leavens, and M. Sitaraman, (eds), Foundations of Component-Based Systems, Cambridge University Press, 2000, pp. 47-68.

[167] Carnegie Mellon University - ABLE Group, *AcmeStudio – Supporting architectural design, analysis and interchange*, available at: http://www-2.cs.cmu.edu/~acme/AcmeStudio/AcmeStudio.html

[168] ISI, *GeoWorlds GIS System*. http://www.isi.edu/geoworlds/

[169] B. Nuseibeh, and S. Easterbrook, Requir*ements Engineering: A Roadmap*, in 23rd International Conference on Software Engineering (ICSE 2001), Toronto, Canada, May 12-19, 2001.

[170] O. Babaoglu, H. Meling, and A. Montresor, *A Framework for the Development of Agent-based Peer-to-Peer Systems*, in 22nd International Conference on Distributed Computing Systems (ICDCS'02), Vienna, Austria, July 2002.

[171] A. Montesor, H, Meling, and O. Babaoglu, *Load-balancing through a Swarm of Autonomous Agents*, in 1st International Workshop on Agents and Peer-to-Peer Computing, Bologna, Italy, July 2002.

[172] R.A. Brooks, *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, 2(1):14-23, April 1986.

[173] S. Benarif, A. Ramdane-Cherif, and N. Levy, *A Multi-agent Platform for Reconfiguration, Adaptation and Evolution of a System at Architectural Level*, in 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2003), Portland Or, USA, May 2003.

[174] T.Murata and N.H. Minsky, *On Monitoring and Steering in Large-Scale Multi-Agent Systems*, in 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2003), Portland Or, USA, May 2003.

[175] J. Kiessel, J. Beard, P. Nielsen, *Failure Recovery: A Software Engineering Methodology for Robust Agents*, in 1st International Workshop on Software

Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2002), Orlando Fl. USA, May 2002.

[176] Z. Guessoum, J.P. Briot, S. Charpentier, *Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems*, in 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2002), Orlando Fl. USA, May 2002.

[177] M. Sloman and E. Lupu. *Security and Management Policy Specification*, IEEE Network, 16(2):10-19, March-April 2002.

[178] The IETF Policy Framework Working Group, *Policy Framework*, available at: http://www.ietf.org/html.charters/policy-charter.html

[179] S. Uttamcandani, C. Talcott, and D. Pease, *Eos: An Approach of Using Behavior Implications for Policy-based Self-Management*, in 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Heidelberg, Germany, October 20-22, 2003.

[180] K. Kleinmann, R. Lazarus, and Ray Tomlinson, *An Infrastructure for Adaptive Control of Multi-Agent Systems*, in IEEE International Conference on the Integration of Knowledge Intensive Multi-Agent Systems (KIMAS'03), Boston, Ma. USA, September 30-October 3, 2003.

[181] O.P. Kreidl, and T.M. Frazier, *Feedback Control Applied to Survivability: A Host-Based Autonomic Defense System*, IEEE Transactions on Reliability, 53(1):148-166, March 2004.

[182] S. Porcarelli, M. Castaldi, F. Di Giandomenico, A. Bondavalli, and P. Inverardi, *An Approach to Manage Reconfiguration in Fault-Tolerant Distributed Systems*, in 2nd ICSE Workshop on Software Architectures for Dependable Systems (WADS 2003), Portland, OR., USA, May 2003.

[183] A. Keller, J.L. Hellerstein, J.L. Wolf, K.-L. Wu, V. Krishnan, *The CHAMPS System: Change Management with Planning and Scheduling*, in 9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), Seoul, Korea, April 2004, to appear.

[184] T. Liu and J.R. Kender, *Time-Constrained Dynamic Semantic Compression for Video Indexing and Interactive Searching*, in IEEE Conference on Computer Vision and Pattern Recognition, Kauai Marriot, Hawaii, USA, December 9-14, 2001.

[185] D.L. Mills, *Network Time Protocol*, Internet RFC 958, December 2002, available at: http://www.ietf.org/rfc/rfc958.txt

[186] L. Gautier and C. Diot, *Design and Evaluation of MiMaze, a Multi-Player Game on the Internet*, in International Conference on Multimedia Computing and Systems, Austin, Tx, USA, June 28-July 1, 1998.

[187] L. Santi, *A User-mode Traffic Shaper for TCP/IP Networks*, available at: http://freshmeat.net/projects/shaperd/

[188] Suhit Gupta and Gail Kaiser *A Virtual Environment for Collaborative Distance Learning With Video Synchronization*, in 7th IASTED International Conference on Computers and Advanced Technology in Education, Aug. 2004, to appear.

[189] S. E. Dossick and G. E. Kaiser. *Chime: A metadata-based distributed software development environment*, in Joint Seventh European Software Engineering Conference and Seventh ACM SIGSOFT International Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999.

[190] S. X. Liang, J. Puett, Luqi, *Perspective-Based Architectural Approach for Dependable Systems*, in 2nd ICSE Workshop on Software Architectures for Dependable Systems (WADS 2003), Portland, Or., USA, May 2003.

[191] Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky (eds.), *Proceedings of the 1st ICSE Workshop on Software Architectures for Dependable Systems (WADS 2002)*, Orlando, Fl. USA, May 25, 2002, available at: http://www.cs.kent.ac.uk/events/conf/2002/wads/proceedingsW12.pdf

[192] Rogério de Lemos, Cristina Gacek, and Alexander Romanovsky (eds.), *Proceedings of the 2nd ICSE Workshop on Software Architectures for Dependable Systems (WADS 2003)*, Portland, Or., USA, May 2003, available at: http://www.cs.kent.ac.uk/events/conf/2003/wads/Proceedings/wads2003.pdf

[193] N. Sample, D. Beringer, L. Melloul, G. Wiederhold, *CLAM: Composition Language for Autonomous Megamodules*, in 3rd International Conference on Coordination Models and Languages (COORD'99), Amsterdam, The Netherlands, April 26-28, 1999.

[194] H. Herbst, *Business Rule-Oriented Conceptual Modeling*, Physica-Verlag Heidelberg, Germay, June 1997.

[195] Microsoft Corp., *Microsoft BizTalk Server: Home*, available at: http://www.microsoft.com/biztalk/

[196] IBM Corp., *WebSphere MQ Workflow*, available at: http://www-306.ibm.com/software/integration/wmqwf/

[197] Integration Definition for Function Modeling, *IDEF0*, Federal Information Processing Standards Publications, December 1993.

[198] Workflow Management Coalition, *Workflow Process Definition Interface – XML Process Definition Language (XPDL)*, October 2002, available at: http://www.wfmc.org/standards/docs/TC-1025_10_xpdl_102502.pdf

[199] F. Ren, The *Marketplace of Enterprise Application Integration*, February 2002, available at: http://www.geocities.com/ffren/eai.html

[200] B. Peuschel and W. Schäfer, *Concepts and Implementation of a Rule-based Process Engine*, in 14th International Conference on Software Engineering (ICSE 1992), Melbourne, Australia, May 1992.

[201] F. Casati, *A Discussion on Approaches to Handling Exceptions in Workflows*, in CSCW98 - Towards Adaptive Workflow Workshop, Seattle, Wa. November 14, 1998.

[202] Microsoft Corp., *Microsoft Dynamic Systems Initiative Overview*, March 2004, available at: of Colorado, 1998.

[203] P. Oreizy, N. Medvidovic, and R. Taylor, *Architecture-Based Runtime Software Evolution*, in International Conference on Software Engineering 1998 (ICSE'98), Kyoto, Japan, April 19-25, 1998.

[204] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Mandrini, *DEEM: a tool for the dependability modeling and evaluation of multiple phased systems*, in Dependable Systems and Networks, New York, USA, 2000.