DySOA: Making Service Systems Self-adaptive

Johanneke Siljee, Ivor Bosloper, Jos Nijhuis, and Dieter Hammer

Department of Computing Science, University of Groningen,
P.O. Box 800, 9700 AV Groningen, The Netherlands
{b.i.j.siljee, i.e.bosloper, j.a.g.nijhuis,
d.k.hammer}@rug.nl

Abstract. Service-centric systems exist in a very dynamic environment. This requires these systems to adapt at runtime in order to keep fulfilling their QoS. In order to create self-adaptive service systems, developers should not only design the service architecture, but also need to design the self-adaptability aspects in a structured way. A key aspect in creating these self-adaptive service systems is modeling runtime variability properties. In this paper, we propose DySOA (Dynamic Service-Oriented Architecture), an architecture that extends service-centric applications to make them self-adaptive. DySOA allows developers to explicitly model elements that deal with QoS evaluation and variable composition configurations. Having the DySOA elements explicit enables separation of concerns, making them adaptable at runtime and reusable in next versions. We demonstrate the use of DySOA with an example.

1 Introduction

Building systems from services has been emerging as a software paradigm [1], [2]. Service-centric systems consist of multiple services, possibly from different service providers, working together to perform some functionality. A service implemented by combining the functionality provided by other services is a *composite service* [3], and the way a composite service is structured and behaves is the *service composition*.

Service-centric computing provides new techniques that allow for greater runtime flexibility. Services are located, bound, and executed at runtime using standard protocols such as UDDI, WSDL, and SOAP [4]. Because services are loosely-coupled and have an explicit interface, it is relatively easy to integrate third-party services, and to substitute one service for another at runtime.

Although the techniques for runtime adapting service systems are available, it currently happens seldom. The reason is that no standards exist for *self-adaptation*, the process where the service system autonomously makes decisions on when and what to change and autonomously enacts the changes. Because technologies for self-adaptation still miss, the burden for adaptation would fall on service users or service providers. But users just want to use the service system, without being bothered with collecting and composing the right services to make up the system. And service providers might provide service systems that have thousands of users, making manual adaptation an impossible task. This results in service-centric systems that, once bound, will always call the same services.

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 255 – 268, 2005.

Having such "static" service systems would not provide any problems, if nothing changes during the period that a user makes use of the service system. Unfortunately, this is not the case. Almost every service system exists in a very dynamic environment that makes it nearly impossible to keep delivering the quality of service (QoS) that the user pays for. The QoS that the service system has to deliver is often formalized in a Service Level Agreement (SLA), and not fulfilling these QoS requirements may result in penalties, e.g. the provider has to pay a fine or will loose customers. Examples of the dynamics that service systems are confronted with are:

- *Unreliable third-party services*: third-party services are not controlled by the service system provider and can fail unexpectedly.
- *User changes*: a service composition may serve multiple users, with each a different SLA and thus different QoS requirements. These QoS requirements can change when the user's context changes, for example because the user moves or starts using the same service on a different device. An example is a changing security requirement, caused by a user leaving the office building and going out on the street. Data transfer should then be better encrypted and limited to non-secure documents.
- *Network irregularities*: available network bandwidth and throughput rates between distributed services vary over time, potentially causing services to be unreachable.

The dynamic context of service systems requires them to adapt to context changes in order to keep fulfilling the QoS requirements. Service systems should be self-adaptive, because, as explained earlier, manual adaptations by users or service providers are not a feasible solution. In order for service systems to be self-adaptive, they must be able to self-detect when and what to change and make this change autonomously. This ability requires, among other things, runtime evaluating if the current QoS fulfills the QoS requirements, and knowing the runtime variability options. In this paper, we focus on modeling the possible configurations (i.e. the variability) in self-adaptive service systems.

1.1 Design of Self-adaptive Systems

A software architecture provides a global perspective on the software system in question. Architecture-based design of large-scale software systems provides major benefits [5]. Designing the architecture for a software system shifts the focus away from implementation issues towards a more abstract level. This enables designers to get a better understanding of the big picture, to reason about and analyze behavior, and to communicate about the system with others.

Part of a service architecture is the service composition, which can be described with languages like BPEL and UML. Many other Web Service standards are used to describe other aspects of the system. Each standard allows developers to specify a certain part of self-adaptive service systems, but no approach exists for developers to design variability options of these systems. This void results in ad-hoc solutions at the implementation level, which hinders the development, reuse and evolution of systems.

In this paper we present DySOA, a Dynamic Service-Oriented Architecture. DySOA extends service applications to make them self-adaptive in order to guarantee

the QoS, despite the dynamic context of service systems. DySOA structures the elements that deal with self-adaptation and variability, making them easier for developers to model and reason about. DySOA provides explicit components that deal with QoS evaluation and composition variability. Having all major self-adaptation elements first-class makes it easier to develop them, to runtime update them, and to reuse them for other systems.

The remainder of this paper is structured as follows. We describe the DySOA architecture in section 2. We show the use of DySOA with an example in Section 3. Section 4 covers related work and Section 5 concludes the paper.

2 DySOA

DySOA stands for Dynamic Service-Oriented Architecture, and is an architectural extension for service-based application systems. DySOA provides a framework for monitoring the application system, evaluating acquired monitoring data against the QoS requirements, and adapting the application at runtime.

The purpose of DySOA is to assist the service application system in maintaining its QoS. At design time, an application developer designs a system that is targeted to fulfill the requirements. However, some of the QoS requirements are only known at runtime (e.g. negotiated in an SLA), and service systems live in dynamic environments, of which the properties cannot always be foreseen at design time. In order to keep delivering the QoS requirements, the application system should be able to self-adapt when necessary.

Many different aspects need to be taken into consideration for the development of a self-adaptive system. It is very difficult to address all concerns in one model, and this one model would be hard to evolve. The complexity can be reduced by splitting the process from monitoring to reconfiguration into several steps. The different concerns are then addressed in different components and models within each step. Having explicit, separate models for the different aspects allows better communication between different stakeholders (e.g. service providers or service users) and independent evolution of the aspects. Furthermore, in order to evolve at runtime, the specific models have to be available at runtime. In the next sections we describe the architectural model of DySOA and the relation with service-based applications.

2.1 The DySOA Adaptation Process

Figure 1 shows the activity diagram of the DySOA runtime adaptation process. First, monitors collect data about the application context. From the collected monitoring data the QoS is determined. Some QoS attributes are directly measurable (e.g. response time), but the values of many QoS attributes cannot be directly monitored and need to be inferred from other context information. The determined QoS is compared with the QoS requirements. If the result of this evaluation indicates the QoS is good enough, then monitoring continues. If the QoS is not good enough, a new configuration is chosen that will satisfy the QoS requirements. Finally, the changes are enacted in the application. Possible changes are substituting a bound service for an alternative service or changing the structure and the flow of the service composition.

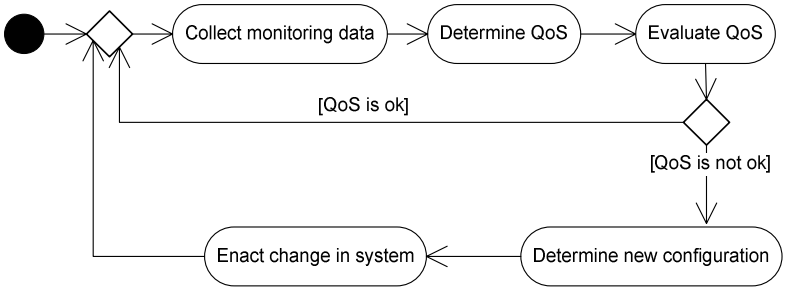

Fig. 1. Activity diagram of the DySOA monitoring and adaptation process

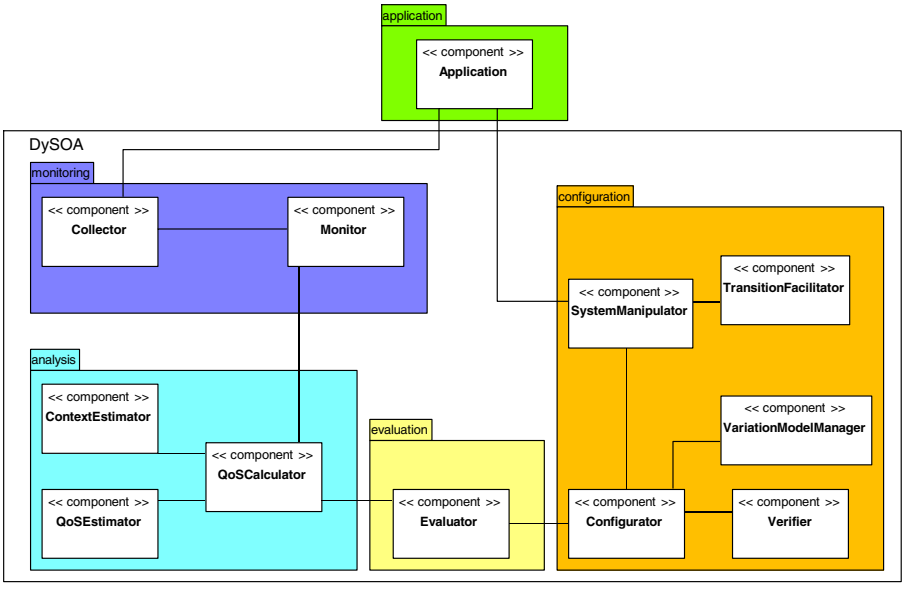

Fig. 2. Overview of the DySOA component architecture

2.2 Overview of the DySOA Architecture

Figure 2 shows an overview of the DySOA architecture. It consists of four component packages: the Monitoring component, the Analysis component, the Evaluation component, and the Configuration component. The Application component does not belong to the DySOA architecture, but refers to the service-based application system that DySOA monitors and configures. Next we describe the functionality of each component and its subcomponents.

2.2.1 Monitoring

The Monitoring component deals with acquiring information about the running application and its environment. The *Collectors* gather the data necessary to

determine the current application QoS. A Collector can, for example, intercept and inspect service messages, or monitor a system resource. The kind of data collected depends on the application domain and the QoS requirements itself, but it typically involves data about individual services in the application (e.g. response times, failure rates, exceptions), the execution environment (e.g. network bandwidth, processor load), and the context of the application users (e.g. user GPS coordinates).

Collectors are runtime created, deployed and removed by the Monitor, which does not interfere or deal with monitoring data itself, but manages the Collectors based on a list of collectors needed per QoS attribute. Upon application reconfiguration the Monitor re-evaluates the list and removes or deploys Collectors where necessary. The Collectors provide the monitoring data to the QoSCalculator.

2.2.2 Analysis

The QoSCalculator uses monitoring data to determine the current QoS of the running application. The determination may be executed in two steps; this depends on whether QoS attribute information can be monitored directly. We distinguish three cases:

1) The QoS can be directly monitored, and the QoSCalculator just sends the monitoring data on to the *Evaluator*. For instance, response time is directly measurable.
2) The monitoring data contains information on the application or user context, and has to be combined with e.g. information on the current application configuration to determine the current QoS. In this case the QoSCalculator sends the monitoring data to the *QoSEstimator* for QoS determination. The result is provided to the Evaluator.
3) Again, the monitoring data only contains information on the application or user context, but of such a low level that first a better understanding of the context is necessary before the QoSEstimator can be used. In this case the QoSCalculator transforms the monitoring data with the *ContextEstimator*. The returned context information is used by the QoSEstimator to determine the QoS sent to the Evaluator. For example, the GPS-coordinates of the user location first need to be translated to country and corresponding language.

The ContextEstimator determines the context by analyzing the monitoring data. A context model is used to associate monitoring data with context situations. A context model can be based on a table or ontology (e.g. OWL [6]), and may be designed by experiments. In the example of the GPS-coordinates, the context model associates them with a language.

The QoSEstimator determines the QoS of the application, based on the context information or the monitoring data. For example, to determine the availability of the entire application system, the down-times of the individual services making up the application are monitored. Because the overall availability depends on the workflow between several services (e.g. parallel or in series), to determine the overall availability the monitoring data is combined with a representation of the dynamic structure of the application.

The QoSEstimator may use a composition model, containing the current configuration of the application and QoS metrics, to calculate the QoS from the monitoring data. Another option is a number of formulas to calculate the QoS. Distinguishing between ContextEstimator and QoSEstimator allows both to be

adapted separately: the former when the context interpretation has to be changed and the latter when the translation to QoS has to be changed.

2.2.3 Evaluation

The *Evaluator* determines if the current QoS satisfies the application QoS requirements. For this purpose, it uses the QoS information provided by the QoSCalculator, and uses a model containing the QoS requirements. The Evaluator compares the QoS information to the QoS requirements; if the current QoS does not satisfy, a reconfiguration is needed. The Evaluator sends this evaluation, including a description of how well each QoS requirement is fulfilled and (expected) reasons of failure, to the *Configurator*.

2.2.4 Reconfiguration

The Configurator is responsible for determining new application configurations. Configuring the application is only possible if the configuration options are known. Furthermore, the system should be able to determine if a configuration is valid. Also, the Configurator should be able to enact a new configuration in the application system. Having these features, Dysoa can reconfigure the service system.

Configuration Variability

In DySOA, designers can model the runtime variability of the self-adaptive service system in a variation point view; a view that can be used as a supplement to other design views. Variation points have been recognized as elements that facilitate systematic documentation and traceability of variability, assessment, and evolution [7]. Thus, variation points are perfectly suited as central elements in managing variability, which holds for runtime variability as well. The variation model behind this view is available at runtime, and is used by the Configurator. Our variation point view is largely based on the one presented in [8]. In this paper, we have altered some aspects to tailor the variation point view to self-adaptive software.

Variation Model

A variation point is uniquely identified by its *name*, and contains a *description* of the variability it provides. This description can be informal or formal, as long as the software developers can describe and understand the rationale behind each variation point. A variation point identifies a location where variation occurs, and is therefore associated with one or more *variants.* The variants of a variation point are, for example, several services that provide the same functionality but with different QoS characteristics, or several *composition fragments*: sets of services organized in different process flows (e.g. BPEL activities).

An intrinsic variation point constraint restricts the variant selection of one variation point. An extrinsic variation point constraint restricts the selection of two or more variants from different variation points. The selection of variant *a* for variation point vp1 might, for instance, demand the selection of variant *b* of variation point vp2, or it might prohibit the selection variant *c* of variation point vp3.

Part of the specification of a variant is the *realization*, which can be described as a recipe with instructions for realizing the binding of the variant dynamically. The current bindings of a variation point are its currently bound variants.

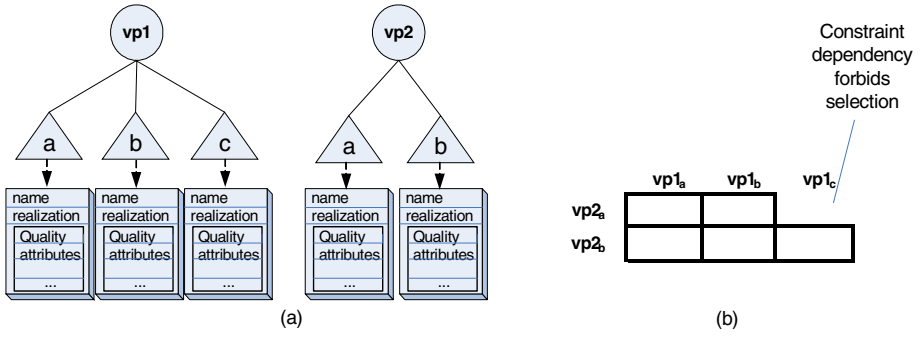

Fig. 3. (a) Two variation points. (b) The VariationModel containing the set of possible configurations of the associated variants. The VariationModel does not contain configuration (*vp2a, vp1c*) because an extrinsic constraint forbids its selection.

Furthermore, service systems often have *open variation points*: variants can be added to or removed from an existing variation point while the system is running. For service compositions this means that at runtime newly discovered services can be added to the composition.

The *VariationModel* of a set of variation points *vps* is the set of possible configurations of the variants belonging to *vps*, together with the (possibly estimated) QoS attribute values of the variants. A DySOA VariationModel only represents configurations possible at runtime. Furthermore, if an intrinsic or extrinsic constraint forbids a certain configuration, then that configuration is not part of the VariationModel. For example, Figure 3 (a) shows two variation points: vp1 and vp2. vp1 has three variants, $vp1_a$, $vp1_b$, and $vp1_c$, and vp2 has two variants, $vp2_a$ and $vp2_b$. An extrinsic constraint forbids the selection of both $vp1_c$ and $vp2_a$. Each variant has an explicit realization and quality attributes. Figure 3 (b) shows the corresponding VariationModel. Five possible configurations exist, as the selection of ($vp2_a$,$vp1_c$) is forbidden by the extrinsic constraint.

The VariationModel is not static; new services can be automatically discovered at runtime or inserted by the user or provider of the service system. Additionally, the QoS characteristics of a variant are not static and should be updatable as well. New QoS values can be determined by monitoring, or a service provider can publish a new QoS specification of its services. The *VariationModelManager* manages all the runtime variability options of the application. The VariationModelManager is responsible for keeping the available variability options up-to-date, e.g. by using service discovery techniques to update the available services (e.g. UDDI).

Configuration Verification

The *Verifier* checks the correctness of new configurations proposed by the Configurator. Examples of checks include variability constraints and deadlock detection.

Configuration Realization

When a new configuration has been verified, the *SystemManipulator* deploys the new configuration in the running system, e.g. by deploying a new orchestration in the BPEL engine of the application system or by reconfiguring a service proxy. The

SystemManipulator makes sure that application state and transactions are managed safely by using the *TransitionFacilitator*. This component can for instance make sure that no transactions are running during configuration, by postponing the start of new transactions. A different approach is to interrupt transactions, send the appropriate exceptions, and execute rollback- or compensation-actions. The state of the running business process (e.g. contained in variables) is copied to the new application state if necessary.

Configuration Selection
Now we are able to deploy new application configurations safely, the Configurator should be able to choose a new configuration, based on the results of the Evaluator. There are several strategies to deal with evaluation results. The Configurator could *optimize*, by always looking for a better configuration, handle *pro-actively*: switching the configuration when danger for QoS failure appears, or *recover*: only choose a new configuration if the QoS fails. Also, the timing for dealing with insufficient QoS is variable; instead of immediate action, it might be allowable to wait for a while to see if the QoS failure is not temporary. Additionally, choosing a new configuration can be based on a formal trade-off of quality attributes (e.g. linear programming), a random choice (in case no quality characteristics of variants are available in the VariationModel) or anything in between. For instance, if time is no issue, the Configurator can test many different configurations before making a decision.

These aspects are specified in the Strategy, a data structure that explicitly represents how to act on the Evaluation results. The Configurator bases the decision process on the currently chosen Strategy.

To summarize: the Configurator uses information from the Evaluator, the Strategy, and the VariationModel to determine a new configuration, and uses the Verifier to verify the correctness of the new configuration.

3 Example

In this section we show how to use DySOA to make a service application self-adaptive. The service application is a video-on-demand service, consisting of third-party services. In order to provide the user with the best QoS for his video stream, the service application needs to be self-adaptive.

The Streaming Video Service (SVS) offers different kinds of streaming video: movies and television series. Users contact the SVS on the internet and select a movie or series episode to watch. For the actual delivery of the video, the SVS uses services from video content suppliers. Each content supplier offers a certain set of streaming video, in several resolutions, and with specific quality characteristics. The SVS discovers the available video suppliers at runtime using a registry.

The SVS automatically binds to a video supplier service that provides the required video content. For the actual streaming, the SVS invokes a proxy service that handles the network connection between the video supplier and the user. The proxy buffers the video stream, in order to protect against short discontinuities and to provide the capability to rebind to another supplier without the user noticing. See Figure 4 for an overview of the streaming video system. Below we show how the components and data structures of the DySOA architecture are instantiated.

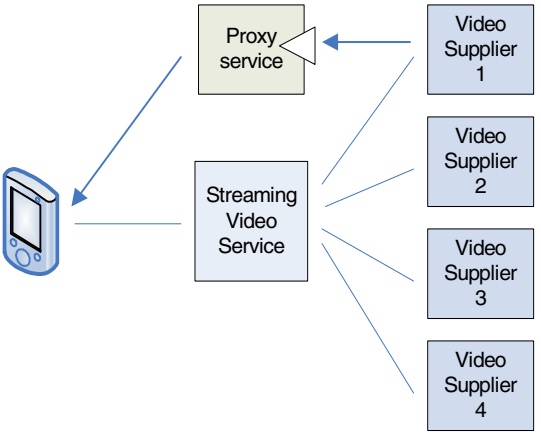

Fig. 4. Service composition of the SVS application

Qos Requirements

Because of space limitations, we do not specify DySOA for all QoS requirements that can trigger adaptations, like performance or cost. Here, we concentrate on two requirements:

- Req 1: continuous availability of the video stream. The video should not stop unless the user explicitly turns it off.
- Req 2: best possible video quality for the user. This is related to the user's display resolution, bandwidth and available streams from video suppliers.

Figure 5 shows the requirements representation.

Monitoring

The quality attributes referred to in the requirements cannot be directly measured. In order to be able to evaluate whether the system fulfills these two requirements, DySOA inserts the following collectors:

```
<wsp:Policy>
 <wsp:All>
  <qos:Policy
      serviceName="VideoProxy">
   <qos:QoS name="Availability">
    <qos:Value>
     <qos:Min>0.95</qos:Min>
     <qos:Pref>1</qos:Pref>
    </qos:Value>
   </qos:QoS>
   <qos:QoS name="VideoQuality">
    <qos:Value>
     <qos:Min>0.8</qos:Min>
     <qos:Pref>1</qos:Pre>
    </qos:Value>
   </qos:QoS>
  </qos:Policy>
 </wsp:All>
</wsp:Policy>
```

Fig. 5. The QoS Requirements

1. A collector monitoring the output bit rate of the proxy video stream sent to the user.
2. A collector monitoring the number of dropped packets on the proxy-to-user connection. From time to time, the collector sends a small burst of packets to estimate the available bandwidth.
3. A collector at the proxy monitoring the user video resolution. The streaming protocol defines that if the video is resized, the collector is notified.

Analysis

Req 1 specifies availability of the video stream at the user playback device. The measured proxy bit rate does not directly define this video stream availability; we need to relate measured data to the video stream availability at the playback device. For this example, the ContextEstimator uses a context model based on the simple heuristic that the bit rate at the playback device is equal to the bit rate at the proxy output. The advantage of having this rule explicit is that it is possible to adapt this heuristic when it turns out to be incorrect.

The ContextEstimator returns the bit rate to the QoSCalculator. The latter sends this information, together with the estimated bandwidth and resolution, to the QoSEstimator, which calculates the availability and video quality.

The QoSEstimator is implemented by several functions that relate the data coming from the QoSCalculator with the QoS requirements on availability and video quality. The availability is specified in terms of the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR) of the video stream at the user, see Table 1. The MTTF is determined from the bit rate as follows:

Let B be the bit rate at the playback device. A failure F_i refers to the event that the bit rate drops to 0, where $F_i(B)$ refers to failures in B. $R_i(F_i)$ is the repair time after F_i. If n is the number of failures during time t, then:

$$MTTF = \frac{t}{\sum_{i=1}^{n} F_i(B)} \qquad MTTR = \frac{R_i(F_i)}{\sum_{i=1}^{n} F_i(B)}$$

The video quality is determined from the bit rate B, the available bandwidth A and the horizontal resolution of the offered stream ($R_{offered}$) and of the playback device R_{user}, see Table 1.

Table 1. Table with the functions for estimating the QoS attributes

QoS Attribute	Function
Availability	$\frac{MTTF}{MTTF + MTTR}$
VideoQuality	$1 - \frac{\lvert R_{offered} - R_{user} \rvert}{R_{user}} - \frac{\lvert B - A \rvert}{A}$

The data flow in the Analysis is as follows; the QoSCalculator sends the collector monitoring data to the ContextEstimator, which returns context information on the playback device's bit rate. The QoSCalculator sends the context information and monitoring data to the QoSEstimator. After the QoSEstimator has determined the current QoS for availability and video quality, the QoS values are sent back to the QoSCalculator, who provides it to the Evaluator.

Evaluation

The Evaluator compares the determined QoS values from the QoSCalculator with the QoS requirements. In our example, the Evaluator checks if the current Availability value is higher than 0.95, and if the current VideoQuality is higher than 0.8. The results of this evaluation specify how each QoS requirement performs, and this is sent to the Configurator. In this example we do not include possible causes for the failure in the message.

Configuration

The VariationModel contains two variation points; a *sup* variation point for choosing between movie suppliers, and a *res* variation point for choosing the video resolution (see Figure 6). The VariationModelManager initially creates the list of variants for *sup* by discovering available services that fulfill the functional requirements (i.e. provide the selected movie). Each variant has a realization that specifies how to invoke the variant. A *supplier* variant is realized by binding to the video supplier, and a *resolution* variant is realized by passing the right parameters during binding.

These variation points cannot be configured independently, as not every supplier provides all resolutions. Choosing a supplier can therefore rule out the choice for a certain resolution. The VariationModel also models these dependencies.

When a QoS requirement is violated, a new configuration is chosen. In this case the Strategy is a recovery strategy that acts immediately if the required QoS is not met. The Configurator asks the VariationModelManager to look up alternative variants, and to update the VariationModel with the observed QoS properties of the

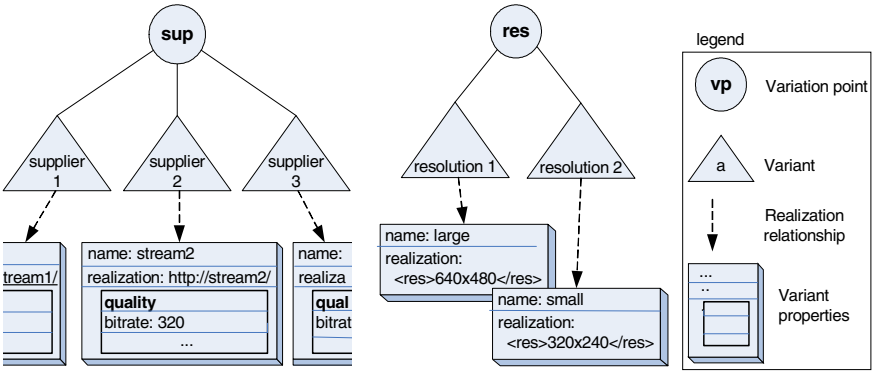

Fig. 6. The SVS VariationModel; the *sup* variation point has discovered video suppliers as variants. The *resolution* variation point has two variants.

failing variant. The Strategy is configured to select the variants that best match the QoS requirements. The selected variants are bound as described by their realizations, and the SystemManipulator is implemented by calling a management method on the proxy to switch the variant.

In this example we have shown how DySOA is instantiated for a simple example. Because all data structures and components that deal with the self-adaptation of DySOA are explicit in the architecture, it is relatively easy to runtime adapt these elements, and to reuse the design for new service applications.

4 Related Work

Most methods for developing runtime self-adaptive systems concentrate on a specific application domain or only on the implementation mechanisms for runtime change. This related work discussion is limited to the more general development approaches at the architectural level.

Some research focuses on a specific part of a dynamic architecture. Yang [9] for example proposes a modeling method for a dynamically extendable adaptation kernel that monitors whether changes should be made. The "adaptation rules" are composed of a condition, which determines when to change, and an action, which specifies how and what to change. The Lasagne framework [10] models runtime variability with "extension identifiers" and provides composition policies attached to a component to change its (messaging) behavior. Tsai [11] presents a framework and tool to specify constraints and audit these constraints at runtime. Service providers register their services at the framework, which tests the service to verify the constraints. Services that pass the tests are available to incorporate in service compositions. Tsai's approach enables quality assurance beforehand, but limits the amount of services that can be used by requiring testing every service before it is published, as not all services and all quality constraints can be tested.

Architecture description languages (ADLs) are used to formally describe a software architecture [12], and several ADLs support dynamism with specific first-class language elements. Dynamic Wright [13] allows defining a variation model by having explicit definitions of variable components. A "configuror" enacts changes and contains a rule block that specifies when to exchange certain components for other components. Weaves [14] provides explicit elements called "instruments" to collect context information. "Observers" are modeled to evaluate this information. "Actors" support enacting change by translating high-level to low-level changes.

Software construction methodologies go beyond modeling and additionally define how to implement dynamic software. Bapty [15] presents an overall design approach called Model Integrated Computing (MIC) for the development of a domain-specific dynamic system. The models of a dynamic system are defined in "multi-aspect domain-specific modeling environments". To create a resulting implementation, the MIC defines a development approach for "system synthesis tools" to turn the created models into executable artifacts, and describes how to create the "runtime execution environment".

The architecture of a dynamic system can systematically be evaluated. Brusilovsky [16] presents a layered evaluation framework for dynamic systems, designed to

determine what parts of the architecture should be adapted if the dynamic behavior does not resemble the required dynamic behavior. The framework separates the responsibilities in the architecture of a dynamic system in two layers. The “adaptation decisions” layer focuses on the architecture for reconfiguration, and the “interaction assessment” layer describes the part of the architecture that monitors environment data and transforms it into information.

5 Conclusion

Designing self-adaptive service systems is a major undertaking and requires software engineering modeling methods and tools. The dynamic context of service systems requires them to adapt to context changes in order to keep fulfilling the QoS requirements. Current standards and techniques for service system engineering typically provide an implementation-level solution for a single aspect of the dynamic behavior. DySOA combines, at the architecture level, the necessary components and data structures for the entire process. This allows separation of concerns and enables developers to manage the complexity of the self-adaptive behavior.

The DySOA architecture can be used to develop service systems that autonomously and dynamically adapt to a changing context and changing user requirements. We demonstrated how the runtime variability is modeled in the architecture for a self-adaptive service application example. Currently we are working on the implementation of DySOA.

Acknowledgements

This research has been sponsored by SeCSE (Service-Centric System Engineering) under contract no. IST-511680.

References

1. Microsoft, Service-Oriented Architecture: Implementation Challenges, http://msdn.microsoft.com/library/en-us/dnmaj/html/aj2soaimpc.asp (2004)
2. Schmelzer R., "Service-Oriented Process Foundation Report", ZTR-WS108, ZapThink (2003)
3. Alonso G., Casati F., Kuno H., Machiraju V., Web Services - Concepts, Architectures and Applications, Springer Verlag (2004)
4. Tsai W. T., Song W., Paul R., Cao Z., Huang H., "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing", *COMPSAC 2004*, Hong Kong (2004) 554-559
5. Shaw M., Garlan D., Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, Upper Saddle River, New Jersey (1996)
6. W3C Recommendation, OWL Web Ontology Language Overview, Recommendation, http://www.w3.org/TR/REC-owl-ref-20040210 (2004)
7. Bosch J., Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach, Addison-Wesley, Boston (2000)

8. Sinnema M., Deelstra S., Nijhuis J., Bosch J., "COVAMOF: A Framework for Modeling Variability in Software Product Families", *The Third Software Product Line Conference (SPLC 2004)*, Boston, USA (2004)
9. Yang Z., Cheng B., Stirewalt K., Sadjadis M., Sowell J., Mckinley P., "An Aspect-Oriented Approach to Dynamic Adaptation", *Proceedings of the Workshop on Self-Healing Systems (WOSS'02)*, ACM SIGSOFT, Charleston, SC (2002)
10. Truyen E., Vanhaute B., Nørregaard Jørgensen B., Joosen W., Verbaeten P., "Dynamic and selective combination of extensions in component-based applications", IEEE, Toronto, Ontario, Canada (2001) 223-242
11. Tsai W. T., Song W., Paul R., Cao Z., Huang H., "Services-Oriented Dynamic Reconfiguration Framework for Dependable Distributed Computing", Hong Kong (2004) 554-559
12. Allen R., Douence D., Garlan D., "Specifying and analyzing Dynamic Software Architecture", Springer-Verlag (1998) 21-37
13. Magee J., Kramer J., "Dynamic Structure in Software Architectures", *Fourth Symposium on the Foundation of Software Engineering (FSE 4)*, ACM SIGSOFT (1996) 24-27
14. Gorlick M. M., Razouk R. R., "Using Weaves for Software Construction and Analysis", *13th International Conference on Software Engineering (ICSE 13)* (1991) 23-34
15. Bapty T., Scoot J., Neema S., Sjtipanovits S., "Uniform Execution Environment for Dynamic Reconfiguration", *IEEE Conference and Workshop on Computer-Based Systems*, Nashville, Tenessee (1999)
16. Brusilovsky P., Karagiannidis C., Sampson D., "The benefits of layered evaluation of adaptive applications and services", *Workshop on Empirical Evaluation of Adaptive Systems*, Sonthofen, Germany (2001)