Reusing Legacy Software in a Self-adaptive Middleware Framework

Santiago Hurtado, Sagar Sen, Rubby Casallas

Reusing Legacy Software in a Self-adaptive Middleware Framework

Santiago Hurtado
Universidad de los Andes
Bogota, Colombia
s-hurtad@uniandes.edu.co

Sagar Sen
ATLANMOD, Ecole des Mines
Nantes, France
sagar.sen@mines-nantes.fr

Rubby Casallas
Universidad de los Andes
Bogota, Colombia
rcasalla@uniandes.edu.co

ABSTRACT

Software that adapts its behavior to an operational context and/or feedback from within is self-adaptive. For instance, a computer vision system to detect people may change its behavior due to change in context such as nightfall. This may entail automatic change in architecture, software components and their parameters at runtime. Legacy software components do not possess this ability. Therefore we ask, can legacy software be successfully cast into a self-adaptive middleware framework ? We present Tekio, a self-adaptive middleware platform to dynamically compose legacy software behavior. Tekio is based on *dynamic component loading* available in a Java implementation of Open Service Gateway Interface (OSGi). Tekio contains generic components to capture context/feedback, plan an adaptation strategy, and reconfigure domain-specific components. The domain-specific components encapsulate legacy behavior implemented possibly in native languages such as C/C++. We implement a self-adaptive vision system in Tekio as a case study. We perform experiments to validate that the self-adaptive layer based on OSGi has negligible effects on the performance of the legacy library namely OpenCV. We also demonstrate that the self-adaptive middleware can handle about 30 adaptations in a span of 2 seconds while producing meaningful output.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Experimentation, Performance, Measurement

Keywords

Self-adaptive software, OSGi, legacy software, software reuse, middleware, framwework

ARM 2011 , December 12, 2011 Lisbon, Portugal

1. INTRODUCTION

Self-adaptive systems dynamically modify themselves due to contextual changes and feedback from within their own components [11]. Contextual changes may emanate from monitoring events from the physical environment, running software/hardware components, and detectors of social and lingual boundaries to name a few. Moreover, these software systems continue running despite user interventions and failures in the underlying software and hardware [3]. Well known examples of such systems are sites such as Google [7]. Contrary to these modern systems legacy software systems/libraries were not built with continual execution, adaptation to context and fault tolerance in mind. Therefore, a natural question arises: Can we reuse existing legacy libraries as components in a self-adaptive framework where they can be loaded/unloaded/replaced at runtime?

Reusing legacy libraries in a self-adaptive middleware framework is the subject of this paper. Usually the legacy libraries are black-boxes with callable functions. For instance, the computer vision library OpenCV [18] contains functions to segment images and detect different types of objects. Different legacy libraries may be available in many different programming languages. In a typical native implementation, these legacy functions are sequentially called in a static program, that is first compiled and then executed with core behavior that is practically immutable at runtime. Our target self-adaptive middleware framework aims to create self-adaptive systems using legacy libraries whose behavior can change at runtime. We aim to (a) separate legacy library functionality into components (b) achieve interoperability between components in different languages (c) automatically (re)configure a set of components in a processing chain based on contextual events and feedback events from the system itself (d) monitor Quality of Service (QoS) and evaluate system performance.

In this paper, we present the self-adaptive middleware framework, Tekio. Tekio adheres to the requirements in [6] for component frameworks to implement dynamic self-adaptive systems. It has the following functionality:(1) Component management that helps define components and the interactions amongst them called a system configuration, (2) Instance management that permits the component lifecycle to be administered and (3)Self-adaptation management for context understanding and mapping context to a system configuration. Tekio is a implemented in Java and provides access to legacy libraries in different languages via Java Native Access (JNA). Self-adaptation in Tekio is

achieved using *dynamic component loading* provided by the OSGi framework specification (formerly known as the Open Services Gateway Initiative). OSGi provides an universal publish-subscribe based protocol for components with different underlying implementation languages to communicate with each other. The OSGi framework has applications ranging from mobile applications, IDE, applications servers to software in automobile industries. The OSGi has 136 official members plus several research projects. It has seven implementations such as Eclipse Equinox, Apache felix, Knopflerfish and projects such as JBoss, Glasfish Fuse EXB Eclipse platform and WebSphere. The widely used OSGi provides the basic functionality to create self-adaptive systems. This paper serves as an evaluation of OSGi to realize self-adaptive middleware and systems.

We use Tekio to build a self-adaptive vision system. This system serves as a case study to evaluate Tekio and OSGi as the middleware framework to reuse legacy open-source libraries. We reuse the OpenCV libraries [18] in software components dynamically managed by Tekio. Tekio components call native code in C/C++ using Java Native Access. We provide number of configurations of these components for adaptation. These configurations achieve tasks such as intrusion detection, face detection, and segmentation. During adaptations we measure frames per second indicating throughput. We also measure the settling time between adaptations. Settling time indicates the time required by Tekio to produce meaningful outputs after adaptation. We perform experiments to demonstrate that Tekio's throughput for a configuration is very close to an identical native implementation despite the layer of software for self-adaptation. The self-adaptive system can demonstrate very low settling times for low and medium resolution input video. For instance, it can provide about 30 adaptations in a span of 2 seconds without significant loss in throughput. However, for high resolution input videos the system fails to provide meaningful outputs when the adaptation frequency goes beyond a certain level. From these results we infer that managing self-adaptation requires rigorous empirical analysis and may entail trade-offs. Empirical analysis is possibly the most practical approach to validate and reuse complex legacy libraries where complete visibility/understanding is impossible.

The paper is organized as follows. In Section 2, we present Tekio's architecture based on OSGi. In Section 3, we validate Tekio. In Section 4, we present the comparison of Tekio with other self-adaptive middleware frameworks. We conclude in Section 5.

2. ARCHITECTURE

In this Section, we present the *Self-Adaptive Middleware* called *Tekio*. The middleware framework allows arrangement of legacy modules in a processing chain that is self-adaptive. Tekio is a component centric architecture that provides possibility to replace any of its components at runtime and maintain a clear separation of concerns that is between self-adaptation and behavioral components. Tekio is built in layers as shown in Figure 1 (a). The lowest software layer is that of OSGi including the Java Virtual Machine. The OSGi layer is primarily responsible for execution of the system using the publish-subscribe paradigm for service oriented architecture. The legacy libraries are called from within domain-specific OSGi components in the second layer. The Java Native Access library is used to access the native library. Finally, Tekio's self-adaptation components (see Section 2.1) manage these domain-specific OSGi components (see 2.2).

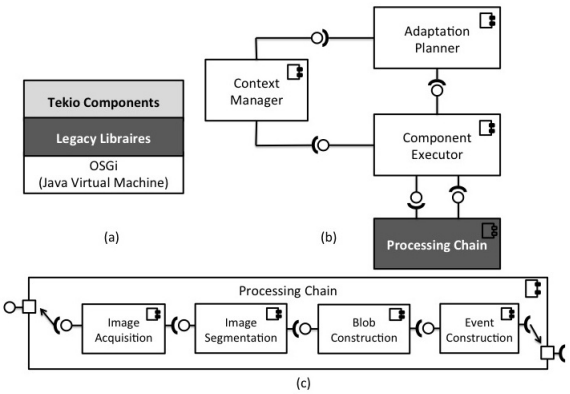

Figure 1: Tekio (a) Software Layers (b) Generic Architecture (b) Domain-specific Architecture

2.1 Tekio Self-adaptation Components

Tekio's self-adaptation is provided by three components as shown in Figure 1. These components realize behavior prescribed by the Monitor, Analyze, Plan, Execute, Knowledge (MAPE-K) loop conceived by IBM in [8]: 1) The *Context Manager* monitors and analyzes the symptoms to know when to request a change in system configuration, the symptoms are represented as Context Events that are events produced by the user, system itself (self) or the environment. The environmental events can be from the video or other hardware sensors, such as movement and weight sensors. 2) The *Adaptation Planner* computes a change plan. This component uses Acher et. al. [1] proposal to specify rules as constraints between two feature models. One feature model captures the variability in context configurations while the other captures variation in system configurations. The adaptation planner generates the system configuration and order in which components must be unloaded/loaded. 3) The *Executor Component* (re)configures the processing chain by the execute the load/unload steps presented by the adaptation planner. The new running configuration may produce new symptoms. This component administers the life cycle of the components in the processing chain. At this moment the system is capable of self-configuring and self-optimizing based on QoS feedback and context awareness.

Tekio is able to provide four different adaptations types: (a)Parameter Adaptation of any of the running components, (b) Component Adaptation to replace a component at runtime for another that provides a similar task, (c) Context Event Adaptation, depending on events from the processing chain the system analyzes how to adapt and request a new adaptation, and (d) QoS Adaptation on which the system is able to maintain a minimum level of quality of service.

2.2 Processing Chain

A processing chain as shown in Figure 1 (c) is the current configuration of domains-specific components managed by Tekio. We implement a self-adaptive vision system using Tekio. The domain-specific/vision OSGi components in the

processing chain call algorithms implemented in the open-source computer vision library OpenCV(Open Source Computer Vision) 2.1. This library is written in C and C++, as many other computer vision libraries for performance benefits. In this paper we focus on processing chains of components but our approach is fully applicable to complex orchestrations where components call other components in both serial and parallel. However, we assume that the orchestration is a fixed entity and only the components are replaced by various alternatives.

We achieve the link between OSGi and legacy libraries using the Java library, Java Native Access (JNA), that seamlessly calls C/C++ functions with negligible performance loss. A similar implementation JavaCV [2] uses the native access functionality. Technically, we pass the memory address of an image to the OpenCV C++ functions. This allows the OSGi framework to manage the native implementation and its process without compromising its performance and original OpenCV library functionalities. The interfaces of the different algorithms are defined in Java while the implementations of the component is in C++. This improves portability and usability of the vision components with all other components written in the OSGi framework. Any other OSGi component can execute the functionality of these vision components without worrying about low-level and native implementation detail in C/C++. We did explore alternative frameworks to OSGi including Tuscany [15] and Frascati [14]. These prototypical frameworks were heavyweight and did not have the maturity of OSGi hence Equinox OSGi was our final choice. OSGi also has the advantage of being a standard allowing interoperability with OSGi components written by other authors.

The vision system's processing chain consists on four types of algorithms: 1) Image acquisition that provides images from different possible sources such as, cameras, streams or files, 2)Image Segmentation that divides images and extracts important objects, 3) Blob Construction and Object Detection that merges the separate objects into a group called a blob and then into an object such as a face 4) Event Construction that converts information at different stages of the vision system to produce either events that are fed back to Tekio or events to final users. Each type of algorithm has several possible implementations that are different manners or configurations to provide specific tasks. For instance, if we use the Smooth Segmentation and the Find Contours blob construction algorithms the system can detect motion. If the Pyramid Segmentation and the HAAR blob construction algorithms are used the system can detect faces. However, if the last configuration uses FGD Segmentation algorithm instead of the pyramid the system can perform faster FPS but lower result quality.

2.3 Example Execution in Tekio

We demonstrate dynamic adaptation in Tekio using Figure 2.3. Tekio starts in intrusion detection mode which is the initial and current configuration. An intrusion event is detected as shown in Figure 2(a) by the context manager. The adaptation planner then decides the new configuration for face detection. The component executor loads the face detection component. The face detection configuration ensues as shown in Figure 2(b).

3. VALIDATION

(a) Intrusion Detection

(b) Face Detection

Figure 2: Dynamic Reconfiguration from Intrusion to Face Detection. Figure (a) is before and Figure (b) is after

Our objective is to validate the self-adaptive middleware Tekio built using an OSGi component framework. The validation aims to encourage the reuse of legacy components in modern self-adaptive frameworks. Our case study is a self-adaptive computer vision system built using Tekio. We choose the representative computer vision domain for a number of reasons: (a) they process large amounts of input data with variations in resolution and content of images, (b) they often require high-throughput and low latency and can run on various operating platforms, (c) they solicit the use of hardware resources such as cameras, high-end servers, and actuators, and (d) computer vision applications require complex software components involving large databases, file systems, and image processing algorithms.

We are intrigued by two principal questions that we address experimentally: **Q1**:How much performance is compromised due to the use of self-adaptive middleware on a native implementation? **Q2**:How often the system can adapt while maintaining a minimal QoS?

The experimental setup to answer these questions is presented in Section 3.1. Finally, we present results of our experiments and discuss them in Section 3.2.

3.1 Experimental Setup

Dynamic adaptation is between different system configurations of video processing components handled by Tekio. A configuration is a specific set of components and its parameters. The components in the configuration are composed in a video processing chain/pipeline. In our experiments we implement a total of six configurations in Figure 3.

Configurations

Number	Name
C1	SMOOTH_SEGMENTATION
C2	FGD_SEGMENTATION
C3	PYRAMID_SEGMENTATION
C4	INTRUSION_DETECTION
C5	FACE_DETECTION
C6	FACE_DETECTION_FGD

Configurations Details

Component	C1	C2	C3	C4	C5	C6
OpenCV AVI Reader	1	1	1	1	1	1
Image Smoothing	2	×	×	2	×	×
FGD Background Subtraction	×	2	×	×	×	2
Pyramid Segmentation	×	×	3	×	2	×
HAAR Detection	×	×	×	3	3	3
Image Window	3	3	4	4	4	4

Figure 3: Experimental Configurations

We present the content of these configurations in Figure 3. For example, the configuration for motion detection contains three different components (a) image acquisition that reads a video from a file, (b) image segmentation that reduces or smoothens the edges of the images, (c) blob construction that finds the contours of objects and translates them into detected movement.

Another dimension of variability in our experiments is the image resolution. The input to the video processing chain is 1020 frames of video in an office space available in three different resolutions:1) High, 1024x720 pixels with a bit rate of 3,582 2) Medium, 720x400 pixels with a bit rate of 1,325 and, 3) Low, 480x272 pixels with a bit rate 681. Each experiment measures percentage of CPU usage, percentage of memory usage and FPS monitored every 5 frames.

We evaluate the configurations based on the following metrics:

Throughput We measure throughput using the rate at which the video processing chain processes frames per second (FPS).

Settling Time The time the system takes to switch from the current configuration to the next. This measurement takes into account the time needed to load the new components

CPU Usage What percentage of the processor Tekio is using at a given instant. This includes the percentage of processor used by the loaded vision components.

Memory Usage What quantity of memory Tekio is using at any given instant. This measurement also includes includes memory used by vision components in Tekio.

We design two experiments to address questions **Q1** and **Q2**:

Experiment E1 For **Q1**, we run a single configuration with three resolutions. Each system configuration is first run with legacy static C components. Second, we run the configuration using Tekio that handles the C components in its self-adaptive framework. The goal is to understand how much performance is lost by adding the self-adaptive layer, and whether this performance loss out costs the benefits of self-adaptation.

Experiment E2 For **Q2**, we run 38 pairs (without repetition) of the 6 configurations in a fixed time. First, we reconfigure 38 times in two minutes. Secondly, we decrease the time limit to 90 seconds and continue reducing to 60, 45, 30, 15, 10, 5, 4, 3, 2, 1 second(s). The purpose here is to figure how the system is affected by the stress to adapt quickly in fixed time. Can we determine when the system stops functioning properly due to a very high frequency of adaptation?

All experiments to answer our empirical questions are executed on an iMac with the Intel Core i3 Processor of 3.06GHz and 4GB 1333 MHz DDR3.

3.2 Results and Discussion

We summarize the results of executing experiment **E1** in Figure 4 and **E2** in Figure 5.

In this section, we summarize and present the results of the experimental executions. Measurements are performed at runtime during the execute phase of the MAPE-K loop.

We execute **Experiment E1** to address **Question Q1**. In Figure 4, we compare a legacy implementation of motion detection in C with motion detection within the self-adaptive framework Tekio. As expected, in Figure 4 (a) we observe that the frame rate is *slightly higher* for a native/legacy implementation of motion detection compared to Tekio for all three resolutions low, medium, and high. The CPU usage for both native and Tekio is similar as shown in Figure 4 (b). However, we observe large difference in memory usage in Tekio compared to a native implementation as seen in Figure 4 (c). The OSGi framework used to develop Tekio uses up considerable amount of memory compared to the native implementation. However, the upper limit is around 2.25% of main memory (4Gb) which is largely acceptable.

We execute **Experiment E2** to address **Question Q2**. We switch between pairs of configuration using Tekio. In all possible 38 pairs of configurations we choose to show 6 pairs where we switch to motion detection from any given configuration with varying time limits and resolutions. The results for low, medium, and high resolution input videos are are shown in Figure 5. In Figure 5 (a), we observe that frame rate starts dropping at 90 seconds time bound for high resolution while 5 seconds time bound for low and medium resolution videos. This implies that a high frequency of adaptation is not suitable for high resolution images while we may expect to adapt several times for lower resolution videos. The CPU usage drops for high resolution after its break point as seen in Figure 4 (b). However, as the frequency of adaptation increases for low and medium the CPU usage increases beyond using a single CPU (upto 170%). Finally, in Figure 4 (c), we notice that memory usage for high resolution is initially very high but gradually drops after the break point. The memory usage remains relatively static for low and medium resolution but drops after the break point.

We define the time required for the system to settle down into a state which produces meaningful results as settling time. In conclusion, we observe that most of the *settling times* are very low. However, if the system adapts very fast

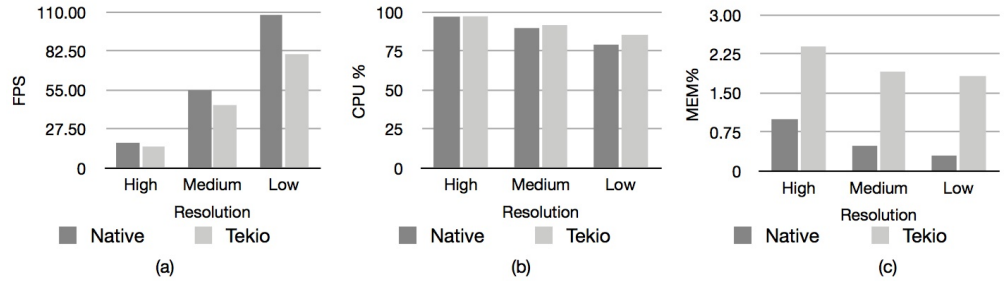

Figure 4: Performance Comparison of Legacy and Self-adaptive for Motion Detection (a) Frame Rate (b) CPU Usage (c) Memory Usage

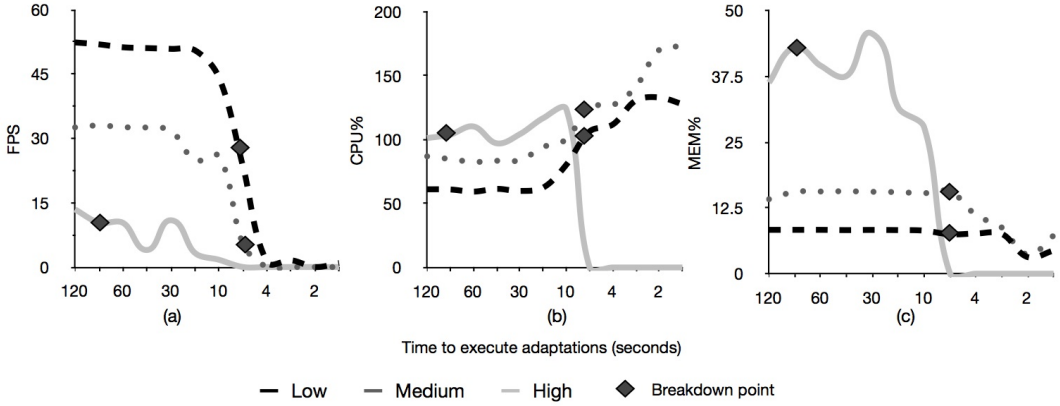

Figure 5: Stress Testing All Switches to Motion Detection (a) Frame Rate (b) CPU Usage (c) Memory Usage

(about 0.25 milliseconds per adaptation) the system can begin to fail depending on the load and video resolution. However, most importantly Tekio does not crash, it simply does not produce results and continues running, this is possible because the algorithms do not produce any exceptions. In future work, we would like to see if Tekio can recover from bursts of high loads automatically without crashing.

4. RELATED WORK

Self-adaptive middleware frameworks are an emerging area in software engineering. There are a number of contributions in the domain self-adaptive middleware. Most self-adaptive middleware are based on the MAPE-K loop proposed by IBM [8]. An ampl survey is provided in [16]. We discuss middleware frameworks that are closely related to Tekio. We place our contribution with respect to established frameworks such as WildCAT [5], Rainbow [17], DIVA [12] and MUSIC [13].

WildCAT [5] is an easy to use Java based framework to build context-aware adaptive systems. Reusing legacy libraries may be achieved in WildCAT by extending classes provided in WildCAT. It does not directly support a standard component framework such as OSGi. In Tekio, we support the OSGi standard for components allowing a large number of components in possibly different implementation languages to interoperate. WildCAT lacks empirical evaluations of its QoS especially with the concern of reusing legacy libraries.

The Rainbow [17] framework provides a reusable architecture to build self-adaptive software systems. The components in rainbow facilities for remote procedure call to access legacy libraries. The authors evaluate the system using a video conferencing case study to show how self-adaptation can help keep latency below a threshold. In [4], the authors evaluate the effectiveness of Rainbow to a realistic rise in demand in a website Znn.com. This rise in demand necessitates quick adaptations which Rainbow seems to handle well. However, the authors do not address the impact of these adaptations on resource usage such as memory and CPU which are often limited. From an interoperability point of view all components in Rainbow are specific to the framework and must be rewritten as Rainbow components much like WildCAT.

DIVA (Dynamic Variability in complex, Adaptive systems) is a European project that provide tools and methodologies to manage dynamic variability in adaptive systems. The approach proposes the use of model-driven and aspect-oriented techniques [12]. DIVA addresses the problem of explosion of possible system configurations and the migration from the current configuration to a valid target configuration [10] in an adaptive system. The framework has been develop using the OSGi specification on the Helios Eclipse Equinox implementation. However, DIVA is a heavy-weight framework and performs adaptations in the order of seconds and not milliseconds such as in Tekio. DIVA also does not address more intricate issues such as its functionality due high frequency of adaptation. DIVA has been applied to a home automation system called Entimid [9]. Entimid

contains a number of OSGi components for sensors, behavior and actuators situated in an apartment for handicapped and elderly people. Entimid tends to behave very erratically when contextual events that trigger adaptations arrive in unpredictable ways. In our opinion, this is primarily due to lack of empirical analysis of the middleware framework.

MUSIC (Self-Adapting Applications for Mobile Users in Ubiquitous Computing Environments) [13] is an European project that provides an open source framework for development and execution of self-adaptive systems using OSGi components. MUSIC provides self-adaptive mobile applications for different devices and operating systems. It also provides developer tools that simplify its use. The platform bases its decisions on monitoring and sensing QoS characteristics of the components that compose the running system. With the help of utility or weight functions the system measures how the system can adapt to a specific context. The adaptation framework has a model of the system structured with several components that can be modified at run time. It also uses the event-driven architecture to manage context events; this allows the system to provide loosely coupled software components. The project goal is to maintain self-adaptation separated from the business logic. Tekio has been inspired by the architecture of MUSIC. However, MUSIC does not validate its infrastructure using empirical studies as we have done for Tekio.

5. CONCLUSION

In this paper, we demonstrate that legacy software libraries can be cast into a self-adaptive middleware framework: Tekio. Tekio is based on the lightweight OSGi standard for dynamic component loading that facilitates self-adaptation of functions in legacy libraries. Using the computer vision library OpenCV we demonstrate that Tekio can be used to build a self-adaptive vision system. We evaluate Tekio for a number of configurations of a vision system. First, we demonstrate that Tekio's performance is negligibly slower for a fixed configuration compared to an identical implementation in native C. The OSGi layer and Tekio's middleware do not incur a large performance overhead. Second, we demonstrate that Tekio can handle about 30 adaptations in a span of 2 seconds. This result however, is dependent on the input video resolution. Only low/medium resolution videos can be dealt with despite high rates of adaptation. When high resolution videos are treated the 30 adaptations may occur in the span of at least 90 seconds. If the adaptation rate is too high Tekio simply stops producing output. It does not crash which open doors to techniques for self-healing.

There are a number of lessons learned in our experiment of reusing legacy libraries in a self-adaptive middleware framework. We enlist some of the important lessons:

1. Self-adaptive frameworks can be superimposed on legacy libraries if the language used to build the framework provides optimized native access to the libraries. In our case, Tekio, Java Native Access used within an OSGi component is an optimized framework with minimum performance overhead.

2. Self-adaptive frameworks have limited control over the resources used by legacy libraries. For instance, internal memory management in the legacy library must be handled well by routines in the library. This is true unless the legacy library provides functions to manage memory or do garbage collection.

3. We have reasonable idea about the range of application domains where our approach would be applicable. For instance, legacy libraries that collect real-time sensor data can be placed in a self-adaptive framework. However, the final output of the application must not be sensitive to loss of some real-time data that may be incurred due to change in configurations. Video, in most cases, is an example of sensor data that remains usable despite loss of some frames. In other cases where application output is sensitive to sensor data such as in safety critical systems the adaptation must be performed only when the system is sure that no critical data is received by the sensors.

4. In our experiments, we also demonstrate that computation intensive behavior such as face detection can be reused in a self-adaptive framework. Tekio quickly loads/unloads heavy components without inflicting a domino effect that can bring the system down. In case of very high-frequency of adaptation Tekio stops producing results. As the frequency reduces, Tekio, resume normal functionality without system crash.

As future work we would like Tekio to provide self-healing and self-protection to the running system. For example, Tekio must be provide the possibility of cleaning up unused memory. Resource management including memory management and debugging of legacy components from Tekio is an important goal of our future work. We would also like Tekio to perform selection from two or more candidates configurations for an adaptation. Tekio also needs to maintain a model@runtime to simplify the interface to reconfiguration in Tekio. In an experiment, it is intriguing to see how Tekio works as a function of component granularity. Should I self-adaptive with many small and lightweight components or with few heavy components?

6. ACKNOWLEDGMENTS

We would like to thank Team PULSAR at INRIA Sophia-Antipolis that provided us with the necessary financial support and resources to apply new ideas in self-adaptive software architectures to the domain of computer vision. In particular, we are grateful to Prof. Jean-Paul Rigault and Dr. Sabine Moisan.

7. REFERENCES

[1] M. Acher, P. Lahire, S. Moisan, and J.-P. Rigault. Tackling High Variability in Video Surveillance Systems through a Model Transformation Approach. In *MiSE '09: Proceedings of the 2009 international workshop on Modeling in software engineering at ICSE 2009 (MiSE'09)*. IEEE Computer Society, May 2009.

[2] S. Audet. Java interface to opencv, Aug. 2011.

[3] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.

[4] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and Others. Software engineering for self-adaptive systems: A research roadmap. *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.

[5] P. David and T. Ledoux. WildCAT: a generic framework for context-aware applications. In *Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–7. ACM, 2005.

[6] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, S. ICT, and N. Trondheim. Using product line techniques to build adaptive systems. In *Software Product Line Conference, 2006 10th International*, page 10. Ieee, 2006.

[7] Y. S. S. K. Hao-hua Chu, Hoi Lee Candy Wong. Dynamic adaptation of gui presentations to heterogeneous device platforms, 2007.

[8] IBM. An architectural blueprint for autonomic computing. Technical Report June, IBM, 2005.

[9] B. Morin, O. Barais, G. Nain, and J.-M. Jézéquel. Taming Dynamically Adaptive Systems with Models and Aspects. In *31st International Conference on Software Engineering (ICSE'09)*, Vancouver, Canada, May 2009.

[10] B. Morin, T. Ledoux, M. B. Hassine, F. Chauvel, O. Barais, and J.-M. Jezequel. Unifying Runtime Adaptation and Design Evolution. *2009 Ninth IEEE International Conference on Computer and Information Technology*, pages 104–109, Oct. 2009.

[11] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, A. L. Wolf, S. Artlcle, and R. Modifications. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*.

[12] I. Romero, F. Juan, V. Chicote, B. Morin, and O. Barais. Using Models@ Runtime for Designing Adaptive Robotics Software: an Experience Report. *Context*, 2010.

[13] R. Rouvoy, M. Beauvois, L. Lozano, J. Lorenzo, and F. Eliassen. Music: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. *Proceedings of the 1st workshop on Mobile middleware embracing the personal communication device - MobMid '08*, page 1, 2008.

[14] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing*, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[15] R. F. H. M. S. N. Simon Laws, Mark Combellack. *Tuscany in Action*. Manning, 2011.

[16] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, SEAMS '11, pages 80–89, New York, NY, USA, 2011. ACM.

[17] S. wen Cheng, A. cheng Huang, D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37:46–54, 2004.

[18] A. Zelinsky. Learning OpenCV—Computer Vision with the OpenCV Library (Bradski, G.R. et al.; 2008)[On the Shelf]. *IEEE Robotics & Automation Magazine*, 16(3):100–100, Sept. 2009.