Towards Lightweight Dynamic Adaptation
A Framework and its Evaluation

Leonardo M. Rocha, Sagar Sen, Sabine Moisan, and Jean-Paul Rigault

INRIA, Sophia-Antipolis, 2004 Route des Lucioles, BP-93 Sophia-Antipolis, France
Firstname.Lastname@inria.fr

Abstract. Many modern systems must run in continually changing contexts. For example, a computer vision system to detect vandalism in train stations must function during the day and at night. The software components for image acquisition and people detection used during daytime may not be the same as those used at night. The system must adapt by replacing running components such as image acquisition from color to infra-red. This adaptation involves context detection, decision on change in components, followed by seamless execution of a new configuration of components. All this must occur at runtime while minimizing the impact of dynamic change on continuity and loss in performance. We present Girgit, a lightweight Python-based framework for building dynamic adaptive software systems. We evaluate it by building a dynamically adaptive vision system followed by performing rigorous experiments to determine its continuity and performance.

Keywords: Adaptive Systems, Dynamic Adaptive Systems, Software Architecture, Framework, Vision System

1 Introduction

Software development is stressed to adapt to needs budding from varying operating environments. Operating environments can vary in terms of hardware platforms such as desktop, servers or mobile devices, software operating systems, and external elements such as the weather, light conditions, or any information captured by a network of sensors. While there is change in environment or *context* we see a need to maintain continuity in *software execution*. Maintaining this continuity involves dynamic adaptation of software to new configurations given occurrences of different contexts. For example, computer vision systems present the need for such dynamicity, where, for instance, change in light conditions solicit the need for different types of cameras and specific vision algorithms. What is the software framework to build such dynamically adaptive systems (DAS) and how can we expect it to behave? This is the question that intrigues us.

Currently, there are already existing frameworks that help create dynamically adaptive systems. For instance, frameworks such as DiVA ([1]) provide a high-level approach using aspect-orientation and model-driven engineering to build DAS. Among its major drawbacks is its relatively new approach involving models and aspects that leads

to a slow learning curve. It is also missing a study on the impact of the dynamic adaptation to continuity and performance. With lessons learned from existing frameworks (see Section 5 for other examples) and prior experience in design and implementation of a large vision system (Scene Understanding Platform, developed at PULSAR team at INRIA Sophia-Antipolis) requiring dynamic adaptation we present our framework, Girgit.

Girgit is a Python-based lightweight framework suitable to build open and closed dynamic system [9]. It leverages the dynamic language abilities of Python such as dynamic module loading and introspection. It provides the basic functions of a DAS such as loading/caching components, mapping context events to a configuration, and changing configurations at runtime. Girgit is easy to learn and use as it provides a small API and its based on the well-known Python programming language. In this paper, we tailor Girgit to perform dynamic scene understanding in video using components for vision algorithms. We perform rigorous empirical studies to validate Girgit for performance in terms of adaptation time and frame rate. We observe that mean adaptation time between configurations is 8 ms without component caching and less than 2 μs with caching of components. This negligible adaptation time has very little effect on the resulting frame rate hence preserving continuity. We also perform a comparative study between pure C++ implementations of configurations and configurations in the Girgit framework for the same set of components. We observe that for processor intensive components the Girgit framework behaviour is similar than the pure C++ implementation while pure C++ implementations performs better for non-processor intensive components.
We may summarize the contributions in the paper as follows:
Contribution 1: Using Girgit we demonstrate that it is possible to easily build closed and open dynamically adaptive systems.
Contribution 2: We also demonstrate through rigorous experimental validation that dynamic adaptation has negligible effect on QoS parameters such as frame rate, and adaptation time.
Contribution 3: We package the framework with several examples on an open source license for external use.

The paper is organized as follows. In Section 2, we present some foundational material to understand the problem and need of dynamic adaptation. In Section 3, we present Girgit's architecture and applications. In Section 4 we present the empirical evaluation of Girgit. Related work is presented in Section 5. We conclude in Section 6.

2 Foundations

The foundations to understand and evaluate Girgit has two dimensions (a) dynamically adaptive systems in Section 2.1 (b) QoS metrics to evaluate Girgit in Section 2.2.

2.1 Dynamically Adaptive Software Systems

An adaptive system is a system whose behavior can be changed during execution according to the needs. *Oreizy et al.* on *An architecture-based approach to self-adaptive software* [9] define the adaptive systems differentiating between systems that can change by means of a pre-programmed set of configurations, from now on, closed-adaptive

systems. Or by adding new configurations during run-time, from now on open-adaptive systems. If the system the system can react to changes in the operating environment the system is called self-adaptive. A definition of an autonomic system is "A system that can manage themselves given high-level objectives from administrators", and can be found on *Kephart and Chess The Vision of Autonomic Computing*[7]

Dynamically adaptive software systems are usually built on the monitor-analyze-plan-execute over a knowledge base (MAPE-K) model shown in Figure 1. The MAPE-K loop is a refinement of the Artificial Intelligence community's sense-plan-act approach of the early 1980s to control autonomous mobile robots. The feedback loop is a control management process description for software management and evolution. The MAPE-K loop presented in Figure 1 monitors and collects events, analyzes them, plans and decides the actions needed to achieve the adaptation or new configuration and finally executes reconfigures the software system.

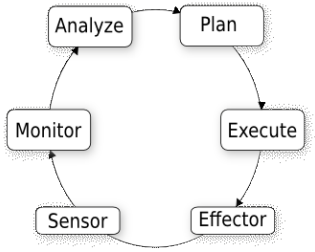


Fig. 1: (a) Vision System MAPE-K View of a Dynamically Adaptive System

2.2 QoS Metrics

In this paper, we evaluate Girgit based on non-functional Quality of service metrics. We define the metrics as follows:

1. *Frame Rate* - It is the number of frames per second (fps) processed by a chain of vision components at the output.
2. *Adaptation Time* - The time it takes to the system to change from the current running configuration to the following taking in account the loading time of the dynamic libraries and components needed to be able to run.

3 The Girgit Dynamic Adaptation Framework

Girgit is a lightweight [1] framework that allows dynamic reconfiguration of *processing chains* and the components inside. Girgit's core manages the relationship between components, that can accept and return any data type. A special data type is that of Events. These objects are treated in a special way by the core. The way in which the components are wired together is described by a *model specification*.

Globally, the core components manage the interaction between components in a dynamically adaptive fashion. The overall architecture for Girgit is presented in Figure 2.

[1] At the submission date 929 lines of Python code according to the sloccount (http://www.dwheeler.com/sloccount/) tool

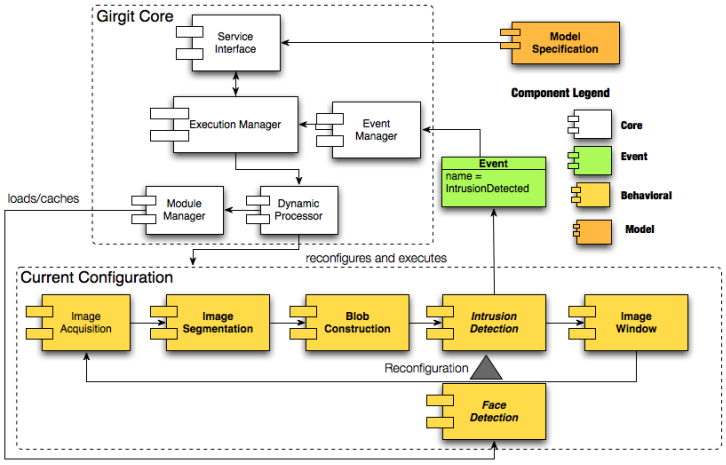


Fig. 2: Girgit's Architecture

3.1 Architecture

Girgit contains core components are shown in Figure 2 and described below:
Model Specification: The model specification specifies, a set of configurations and event/action pairs (rules) to change between configurations. A particular configuration is defined as a set components, their parameters, the method that must be called, and the interconnection between these components. A model specification is provided using the Service Interface.
Service Interface: Provides the interfaces to interact with Girgit.
Dynamic Processor: Executes the current configuration as described by a model specification, finding out during runtime the component to execute, the method to call and the set of parameters to use for the call. Also reconfigures either the processing chain or the components.
Execution Manager: Manages the execution. Orchestrates the calls to the Event Manager and the Dynamic Processing Chain.
Event Manager: Manages incoming events and suggests actions such as changing of components or the complete processing chain. The rules to map events to actions are specified in the model.
Module Manager: Loads and caches instances of components.

When Girgit is running it contains also dynamically loaded components interconnected in a graph that we call *Processing Chain*. This graph contains information about how the components are interconnected, as well as the temporal information of the data, this way we can connect a component A to the data generated as output of component B in some previous loop making feedback possible.

In this paper, we evaluate Girgit by building a vision system using it. In this case, the processing chains are defined by components encoding vision algorithms such as acquisition, segmentation, and blob construction. The components that can return events use information from vision components to return a boolean value for a given event. For instance, if an intrusion detection is an event as shown in Figure 2. A complex event

may be encoded in only one event by a component that defines function over a set of events.

3.2 Girgit's Implementation

Girgit is implemented in Python due to its ability to introspect and load modules at runtime/dynamically. Girgit can accept multiple *configurations*. Each of those configurations define a *processing chain*, the event that triggers the configuration, the execution order of the *components* in the processing chain and the parameters taken and returned by the components. A *processing chain* is defined by a directed graph describing the interconnection between the components. When multiple configurations are passed at initialization, the initial configuration must be passed as well and the framework will start running the configuration tagged as initial.

We explain a regular activity cycle in Girgit using the input model specification. The main loop of Girgit is shown in Figure 3. The **MainLoop** component calls the *process* in the **ExecutionManager** component. The **ExecutionManager** calls the *process* method in the **DynamicProcessor**. The **DynamicProcessor** call all the components (using *callComponent*) in the current configuration of a processing chain. The components may *return* events to the **DynamicProcessor** which in turn sends the list of events to the **ExecutionManager**. The **ExecutionManager** sends the list of events to the **EventManager** by calling *addEvents*. The **EventManager** accumulates these events in a queue and maps each event to an action in the order they are popped out. The mapping of an event to an action is implemented in a Python dictionary (available from the model specification) where every event is identified by a name. When an event matches a rule the **EventManager** returns the pre-encoded action. The processed events are saved (up to an adjustable length) in a history queue for rollback if necessary.
An action can be one of the following:
A1: Adaptation of an entire processing chain and its reconfiguration as shown in Figure 4 (a)
A2: Reconfiguration of a single component as shown in Figure 4 (b)

When action **A1** is sent to the **ExecutionManager** it executes the encapsulating operation *reconfigure*. The reconfigure operation invokes *setConfig* in the **DynamicProcessor** as seen in Figure 4 (a). The operation *setConfig* first verifies that the appropriate processing chain is loaded with a new configuration. This is verified by *checkConfig*. If a new processing chain with a new configuration is invoked then we update a dictionary containing the new components of the chain and its execution order/orchestration in the configuration. We next verify if each component in the dictionary is already cached from a previous step. If a component is not loaded then we execute *loadComp* to load the new components using the **ModuleManager**. It also removes any unrequired components from memory. The **ModuleManager** then returns control to the **DynamicProcessor** which finally returns control to the **ExecutionManager**.

When action **A2** is sent to the **ExecutionManager** it executes *reconfigureComponent* in **DynamicProcessor** from within the general operation *reconfigure*. The **DynamicProcessor** calls *update* on the **Component** to change its parameters and reload it while keep the rest of the chain intact.

Girgit can load multiple processing chains in the same time, for instance, to conduct several experiments on the same real-time input. We use namespaces to avoid collisions between components of different processing chains. The namespaces (Figure 5) are created by the **ModuleManager** which maintains references to components and their respective processing chains in a dictionary. Each namespace is named after a processing chain. When components are shared we have a *global* scope namespace that is accessible to all configurations.

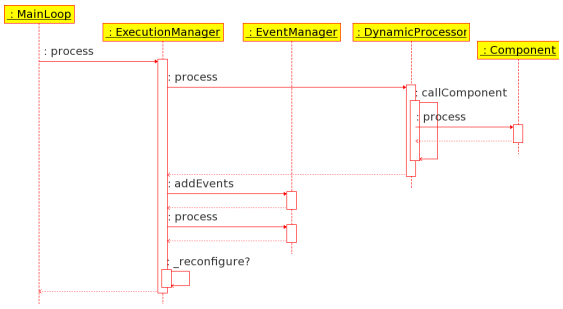
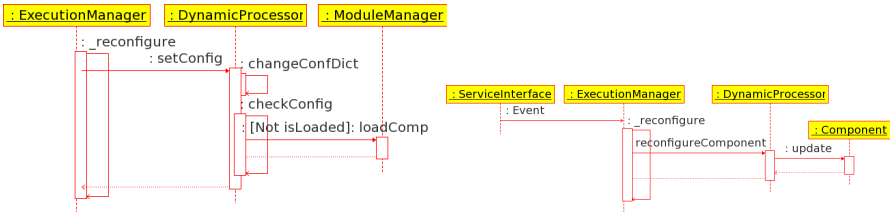


Fig. 3: Diagram that shows the sequence of calls for a cycle in the main loop



(a) Reconfiguration of a Processing Chain (b) Component Reconfiguration

Fig. 4: Reconfiguration

In Figure 2 we present the *snapshot* of a reconfiguration where the *Face Detection* component replaces the *Intrusion Detection* component due to an *Intrusion Detected* event. We first update the configuration dictionary and the execution order list with all the new elements and erase the components that should not exist any more. We then check the configuration and load uniquely the libraries and components that are not already cached. In this case, we remove movement detection and replace it with blob construction, face detection, and image window for visual inspection.

3.3 Example Execution in Girgit

We demonstrate dynamic adaptation in Girgit using Figure 6. Girgit starts in intrusion detection mode which is the initial and current configuration. During the loop execution, When an intrusion is detected as shown in Figure 6a, an event is generated as shown in Figure 2. The Execution Manager gets the event and asks the event manager

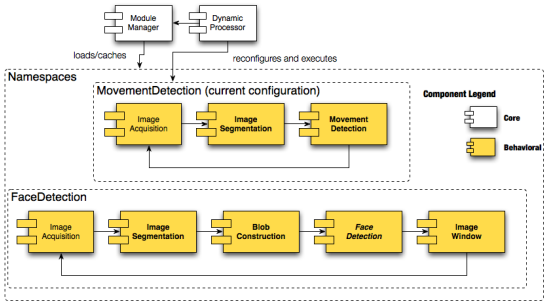


Fig. 5: Representation of namespaces, showing an active processing chain (*SimpleMovementDetection*) and a passive processing chain(*Facedetection*).

what to do, and this responds with an action corresponding to a change in configuration that contains a new component for face detection. The Execution Manager instructs the Dynamic Processor to change the configuration for the new one, this last component asks the Module Manager to load the face detection component from hard-disk or from memory cache. Finally, the processing loop continue and the result with the face detection component ensues as shown in Figure 6b.

(a) Intrusion Detection (b) Face Detection

Fig. 6: Dynamic Reconfiguration from Intrusion to Face Detection. Figure 6a is before and Figure 6b is after

4 Empirical Evaluation

We empirically evaluate Girgit to answer the following questions:
Q1 How long does Girgit take to reconfigure or adapt?
Q2 How does adaptation in Girgit affect continuity of operation?
Q3 How does the execution with Girgit compares to a pure C++ configuration implementation?

4.1 Experimental Setup

We evaluate Girgit using a total of eight different configurations. Each configuration contains a different set of components and/or their parameter settings. Most components encapsulate libraries in OpenCV [2] such as Pyramid segmentation and HAAR object detection. The primary goal was to build dynamic adaptive vision test systems that switch between configuration according to arriving events, for the evaluation purposes the events where set to 5 seconds. We present the number and name of the different configurations on the left side in Figure 7. The components used in the configurations are shown on the right of Figure 7. We also provide the order in the processing chain for these components in the configuration. For instance, OpenCV AVI reader is first in the order in all configurations. The symbol $\times$ indicates absence of the component in the configuration.

Configurations

ID	Name
C1	SMOOTH_SEGMENTATION
C2	FGD_SEGMENTATION
C3	PYRAMID_SEGMENTATION
C4	INTRUSION_DETECTION
C5	FACE_DETECTION_PYR
C6	FACE_DETECTION_FGD
C7	FACE_DETECTION
C8	VIDEO_PLAYER

Configurations Details

Component	C1	C2	C3	C4	C5	C6	C7	C8
OpenCV AVI Reader	1	1	1	1	1	1	1	1
Image Smoothing	2	×	×	2	×	×	×	×
FGD Background Subtraction	×	2	×	×	×	2	×	×
Pyramid Segmentation	×	×	3	×	3	×	×	×
HAAR Detection	×	×	×	3	3	3	2	×
Image Window	3	3	4	4	4	4	3	2
Profiler	×	×	×	×	×	×	4	3

Fig. 7: Experimental Configurations

Using the first six configurations we perform the following experiments to answer questions Q1 and Q2.

Experiment E1: For a single configuration (configuration **C1**) we execute 15 reconfigurations of the same configuration (a) With caching and (b) Without caching. The constant factor here is the configuration that remains fixed. The goal of this experiment is to study stability in adaptation times and frame rate due to dynamic reconfiguration.

Experiment E2: In this experiment, we execute all pairs of configuration transitions possible using the first six configurations (a) With caching and (b) Without caching. The goal of this experiment was to introduce variation in configurations and check if this affected adaptation times and frame rate. How stable is the adaptation time when configurations change? The configurations were changed every 5 seconds.

For both experiments we measure the frame rate and adaptation times. We execute the same experiment 100 times to validate the stability of our results.

Experiment E3: This experiment aims to evaluate the cumulative impact of the dynamic resolution of of the python wrappers and dynamic resolution of calls. We compare the same behavior written in a pure C++ implementation and using Girgit. Two configurations where used, a simple one that only reads and shows the image **C7** (7) and a more complex one that uses the HAAR algorithm **C8**.

For this experiment we measure only FPS as it is the most meaningful QoS value for video processing. The looping was left free to achieve the maximum FPS rate possible. A sequence of the first 1000 frames of the same video where used for 4 runs. The FPS was measured every 10 frames.

For all the experiments the input video was a long sequence from an office space where there are multiple people entering and leaving the scene. The execution took place in the following platform: Linux Fedora 14 x86_64, Intel(R) Core(TM) i7 CPU Q720 @ 1.60GHz, Memory: 8GB.

4.2 Results and Discussion

The results of experiments E1 and E2 are summarized in this section. In Table 8, we show the statistical results obtained during the experiment E1. We observe that there is a considerable difference between the adaptation times with and without caching system activated, having the cached system a much better absolute performance. Considering that the minimum values in frame rate obtained are instantaneous values for the reconfiguration instant, that vision systems usually vary in performance when the video situation change and that the mean frame rate is almost equal to the one with caching system we can conclude that the system without cache actually do not have a meaningful impact on the vision system overall performance. This result addresses question Q1 and demonstrate that Girgit indeed has a low adaptation time.

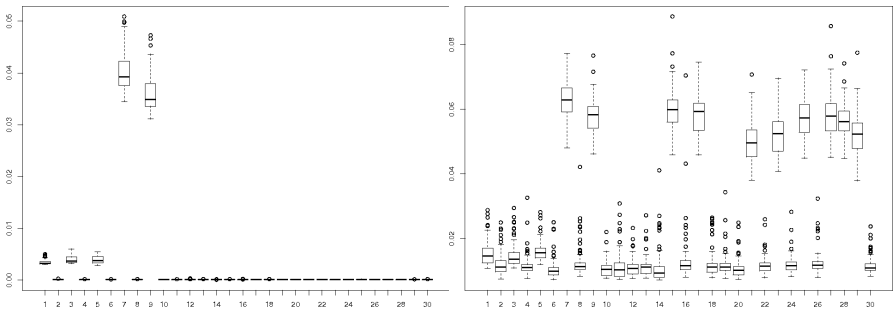
Stat	Cache ON	unit	Cache OFF	unit
Adaptation Mean	1.82	μs	8.00	ms
Adaptation Max	16.5	μs	15.6	ms
Adaptation Std Dev	8.16	μs	1.18	ms
Frame rate Mean	23.98	fps	23.74	fps
Frame rate Max	25.72	fps	31.35	fps
Frame rate Min	23.90	fps	16.25	fps
Frame rate Std Dev	0.38	fps	0.80	fps

Fig. 8: Adaptation times and fame rate for Experiment E1

Using the dynamic system with caching system we see *no loss* in frame rate. While without caching there is a small loss in frame rate that does not seems to be significative for this type of application. These results address question Q2. The continuity in frame rate is largely preserved in Girgit.

In experiment E2, we vary the configurations to see its effect on adaptation time and frame rate. As seen in Figure 9a, the caching has high adaptation times for the first few configurations as components are loaded and cached. However, after the components are cached the adaptation time drops drastically (peak not seen in the plot is 0.16 ms and less than 2 μs mean). However, when the caching system is not used (Figure 9b), the adaptation times are higher peaking at 15.6 ms (not showed in the boxplot) and with a mean of 8ms. This results sheds light on question Q1. It demonstrates that when configurations change caching allows reduction in adaptation times later in the runtime life of Girgit. With respect to Q2 there is also more stability in the adaptation times. The

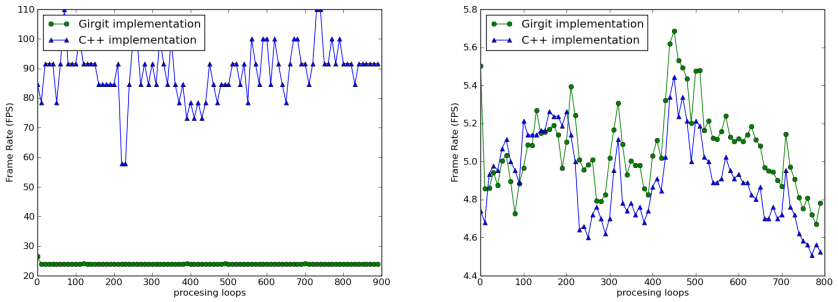
frame rate with caching is also stable (not shown in the paper due to space limitations) when multiple configurations change.



(a) Caching ON (b) Caching OFF

Fig. 9: Adaptation times for the 30 possible configuration transitions (transition every 5 seconds) all pairs of the 6 configurations with/without caching. X-axis represent the transitions, Y-axis represent adaptation time in seconds

In the experiment E3, we compare the performance, in terms of frame rate (FPS) for 2 different configurations: **C7** and **C8**. Each configuration was programmed: (a) entirely in C++ in a dedicated non modular program for achieving the maximum performance; (b) using components with a Girgit configuration file. Was not a surprise that the FPS is higher in C++ than with Girgit [2] for the example of a video player 10a. But, when the algorithms in the modules are more processor intensive (Figure 10b), we can observe that Girgit's throughput shows no significative difference with the C++ execution.



(a) Video Player (b) Face Detection

Fig. 10: Frame rate comparison between a pure C++ program and the same behaviour implemented in Girgit

[2] that has to resolve the execution modules, methods, calls and order in runtime, and is implemented in Python, an interpreted language running on a Virtual Machine

We conclude dynamic adaptation, and in particular, our platform, is not only suitable for applications where dynamic adaptation is needed and where the algorithms to execute are processor intensive but has many advantages respect a static program.

4.3 Threats to Validity

The experiments performed on Girgit is within certain bounds. We execute experiments for eight vision components. Although, our experiments represent a realistic vision system we may imagine a case study with 100s of components. Therefore, the scalability of Girgit to hundreds of components is an issue that needs investigation. Using components with badly managed memory can result in errors in experimental observations such as memory usage. In our case study, we thoroughly verify the 8 components for non-existence of memory leaks. We demonstrate continuity of Girgit in terms of frame rate and adaptation times for a given set of configurations. However, continuity can have different semantics. For instance, continuity of tracking an object when context changes. Finally, we perform experiments using a long video sequence from a large office with several people coming in and out of a scene. We need to validate Girgit for various application scenarios and case studies. Our study shows that this dynamic adaptive framework is not suitable for non processor intensive processing chains that need a high loop throughput. Debugging components in Girgit is an important challenge which needs further investigation. At present, individual components have to be tested and debugged separately. In case we have to deal with faulty components at runtime we may consider applying approaches based on Acceptability Oriented Computer [11] and replacing the faulty components to keep the system running.

5 Related Work

Building dynamically adaptive software is a hot area of work in software engineering. This interest in dynamic adaptation comes with maturity in component-based/service-oriented software, dynamic and introspective languages such as Python, and distributed publish-subscribe systems [4]. Dynamic adaptation between a number of components with different parameters presents a large space of variability that is best managed using a high-level model [1] [12]. Models@runtime [8] is the current trend to manage/reason about dynamically adaptive software. Examples of dynamically adaptive software frameworks include MOCAS [3], and RAINBOW [5]. Girgit maintains a specification of the adaptive system configuration which is a model@runtime. Girgit introspectively adapts to the changes made to the model. We apply Girgit's framework to the large case study of a vision system.

We evaluate Girgit for QoS using rigorous empirical studies. Empirical studies in dynamically adaptive systems validate its functional and non-functional behavior by exploring the domain of its variability [6] [10]. In our experiments, we cover all possible component configurations of a vision system built within Girgit and compare behaviour of Girgit to a pure C++ application.

6 Conclusion

We presented Girgit, a framework for building dynamic adaptive systems that allows creating open and closed dynamic systems as well as autonomic systems. In first in-

stance we show the architecture used for the framework. Later we used Girgit to build a dynamic adaptive vision system case study that clearly separates the dynamic adaptation details from the actual vision components. With which we demonstrate that there is negligible effect on performance due to dynamic adaptation. This is especially true with *caching* is used and for applications where the modules are processor intensive. One of the main problems with dynamic adaptation in vision is the resilience of people in using them with performance being a big question mark. With this article we hope clarifying the scope and use of dynamic hybrid (in more than one programming language) adaptive systems and helping to eliminate the apprehension in the mind of a vision expert about the performance related feasibility of building dynamically adaptive vision systems.

The framework facilitates creating and prototyping dynamically adaptive systems and allows open and closed dynamic adaptive systems. The case of open adaptive is being used for interactively modify the systems in study. We are currently using the framework in the PULSAR team at INRIA Sophia Antipolis for research and setup tasks. More information can be found at `http://www-sop.inria.fr/teams/pulsar/Girgit/`.

References

1. Diva project http://www.ict-diva.eu/diva/.
2. Opencv project http://opencv.willowgarage.com/wiki/.
3. CYRIL BALLAGNY, N. H., AND BARBIER, F. Mocas: a model-based approach for building self-adaptive software components. *ECMDA* (2009).
4. EUGSTER, P., FELBER, P. A., GUERRAOUI, R., AND KERMARREC, A.-M. The many faces of publish/subscribe. *ACM Computing Surveys 35* (2003), 114–131.
5. GARLAN, D., CHENG, S.-W., HUANG, A.-C., SCHMERL, B., AND STEENKISTE, P. Rainbow: architecture-based self-adaptation with reusable infrastructure. *Computer 37*, 10 (oct. 2004), 46 – 54.
6. KATTEPUR, A., SEN, S., BAUDRY, B., BENVENISTE, A., AND JARD, C. Variability modeling and qos analysis of web services orchestrations. In *Proceedings of the 2010 IEEE International Conference on Web Services* (Washington, DC, USA, 2010), ICWS '10, IEEE Computer Society, pp. 99–106.
7. KEPHART, J. O., AND CHESS, D. M. *The Vision of Autonomic Computing*. IEEE Computer Society, 2003.
8. MORIN, B., BARAIS, O., JEZEQUEL, J.-M., FLEUREY, F., AND SOLBERG, A. Models@ run.time to support dynamic adaptation. *Computer 42*, 10 (oct. 2009), 44 –51.
9. OREIZY, P., GORLICK, M., TAYLOR, R., HEIMHIGNER, D., JOHNSON, G., MEDVIDOVIC, N., QUILICI, A., ROSENBLUM, D., AND WOLF, A. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE 14*, 3 (may/jun 1999), 54 –62.
10. PERROUIN, G., SEN, S., KLEIN, J., BAUDRY, B., AND LE TRAON, Y. Automatic and scalable t-wise test case generation strategies for software product lines. In *International Conference on Software Testing (ICST)* (Paris, France, April 2010), IEEE.
11. RINARD, M. Acceptability oriented computing. In *ACM SIGPLAN* (2003).
12. ZHANG, AND CHENG. Model-based development of dynamically adaptive software.