Causality Control in Dynamic Platforms

Jacky Estublier, Germán Vega
Université Grenoble Alpes, LIG
Grenoble, France
{Jacky.Estublier, German.Vega}@imag.fr

Abstract— **The increasing dynamicity of ubiquitous environments and the rapid penetration of many sensors in our day life are causes of concern for application designers and developers. Indeed, they have to implement reliable applications in a context in which the managed entities have a very low level of abstraction; they are autonomous, heterogeneous, and change in unpredictable ways. To simplify developers work, there is a clear need to define a higher level of abstraction in which these entities can be represented homogeneously and managed systematically, irrespective of the many technical details. To be used safely, this representation must be causally related to the represented entities. Providing a high level causal representation is very challenging, because its implementation depends on the nature of the managed entities, and because in ubiquitous systems the representation and the system are evolving simultaneously and independently, sometimes in incompatible ways. The paper describes a systematic and extensible way to define and implement causality, and presents the experience with the Apam system in the domain of service platforms.**

Keywords- component; model; services; platform; causality; operational; OSGi.

I. INTRODUCTION

Almost every piece of information managed by a program is a representation of something, either abstract concepts (integers, strings), or real entities (persons, cars). An important part of computing sciences has been devoted to representations. In the 2000s, modelling proposed to make more formal the relationship between a representation (a model) and the system being represented, the System Under Study (SUS).

When a part of the SUS is not directly accessible to the machine (e.g., a part of the “real world”), building a representation is a preliminary step before writing a program that works on the SUS. A fundamental property of a model is to provide a convenient representation of the SUS: it should only represent what is needed at the right level of abstraction, making the understanding easier, and making the programs simpler. Therefore, even when the SUS is itself abstract, it is often convenient to build “on top” of it, a representation that fits better the needs.

Note that the SUS itself can be a representation of a lower level system, making SUS and representation relative concepts. Indeed, computer sciences make heavy use of chains of representations, like abstraction layers in an operating system.

The intuition often makes a distinction between SUS that are part of the real world (e.g., cars and houses represented in a database), and SUS that are electronic entities (files and ports in an operating system). This intuition is often misleading, machine world and real world are not two separate worlds; after all, the machine too pertains to the real world, and the SUS can include entities pertaining to the machine.

However, what is relevant is that for electronic entities changing the representation can be translated automatically, and almost instantaneously, into corresponding changes on the represented entity (e.g., closing a port or changing the value of an integer Java variable). We say that the representation is operational. It is of course not the case for real world entities (changing the color of a car registered in a data base does not actually change the color of the car itself).

For electronic entities, system changes can be directly observed and translated into the corresponding representation. We say that the representation is sensitive to its SUS. For real world entities, it is a program, or an administrator, that keeps the representation up to date, not the entity itself.

A representation that is both operational and sensitive (as illustrated in Figure 1) is said to be causally related to its SUS, and causality is the relationship between a representation and its SUS. Operationality and sensitivity are reciprocal properties, making causality symmetric, and making relative the concepts of SUS and representation.

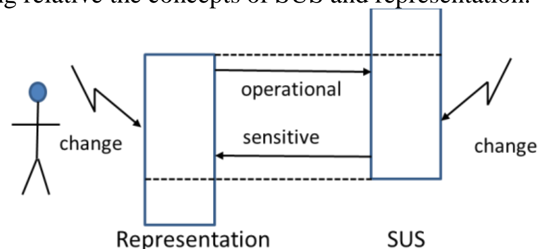

Figure 1. Causality.

With the advent of modeling as a discipline, the representation has become more formal and higher level, often based on Object-Oriented concepts, making program and representation pretty close, blurring even more the boundary between system and representation.

With the recent irruption of many sensors and actioners (ubiquitous computing, home automation, games, and so on) the machine and the real world became intertwined, because electronic devices have the property to be both in the machine world and in the real world; we call it the shared world.

Unfortunately, the electronic side of devices being very low level, the need for an abstract and convenient representation does not disappear (we still need to hide heterogeneity, and many technical details like communication and discovery protocols). Such a representation of the shared world can be operational, i.e., changes on the representation can be automatically translated into the corresponding actions on the associated device(s).

Conversely, devices being part of the real world, their state can be changed by the real world itself (a temperature sensor, for example). Being in the shared world, this (real) change can be translated into the corresponding change into its abstract representation, making the representation sensitive to its SUS.

Causality is transitive allowing the definition of chains of representation, of increasing abstraction, each layer being still causally related to the "lowest" one. This property is well known for operationality (the usual abstraction layers), with causality it allows, for example, to represent and manage sensor networks at the relevant abstraction level. Therefore, causality allows program to work on the representation as if working on the SUS itself, even for dynamic and autonomous SUS. This is an important property that simplifies dramatically the writing of program. However, causality is a relationship that is very difficult to enforce in practice, which explains why it is so uncommon.

We have experimented how causality can be defined and managed in a systematic way in the case where the representation is a model, close to Extended Entity-Relationship (EER), and the SUS is a software services platform (like OSGi [1]) both running on same computer. However, this simplification does not reduce significantly the generality because, in our system, everything is represented as a service: the shared world entities (sensor drivers are services), remote entities (their proxy are services), and so on.

This paper is structured as follows: Section II describes the representation layer (a component model), the execution platform (the SUS) and how causality is defined. Section III describes how the representation and the execution platform are synchronized; Sections IV and V show how this representation can be extended to handle provisioning and how it can cope with failure; finally, we conclude with a discussion of our validation and experience, the related work, and perspectives.

II. THE APAM REPRESENTATION LAYER

The Application Abstract Machine (Apam) platform proposes to its users (program and administrators) the mechanisms to build models that are causally related to their SUS. All representations in Apam conform to the meta-model depicted in Figure 2.

Apam proposes a generic Entity-Relationship meta-model (left part of the figure) that can be used to build any abstract representation, particularly for real-world entities. This generic meta-model has been specialized into a component meta-model (center of the figure) that is used specifically to represent services and running applications (the machine-world) of a service platform like OSGi.

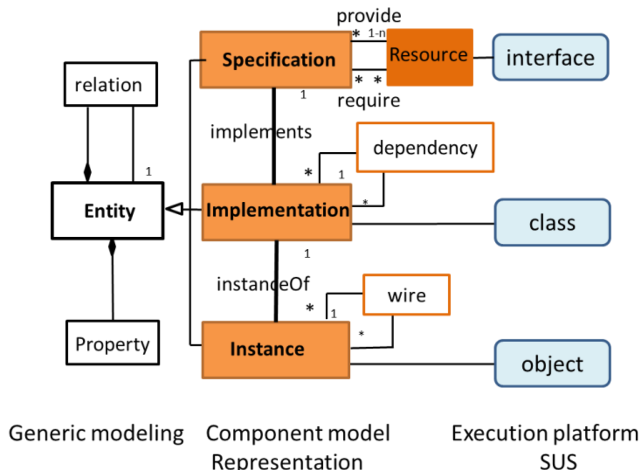

Figure 2. Apam Metamodel (simplified view).

Apam maintains a causality relationship between the abstract representation of the application and its concrete code artifacts (Java code in our case), both at development and runtime.

At development time, causality is enforced by the Apam compiler. The compiler ensures that the abstract relationships defined by the component model are actually implemented at the code level. For example, in the meta-model the relation *implements* means that the resources provided and required by the associated *specification* must be provided and required by its *implementations*; the compiler checks that the associated class really provides (implements, in the Java sense) and requires (imports, in Java code) the interfaces associated to the resources. The complete component model, and its Java mapping, is fully presented in [2]. The Apam compiler also performs byte code instrumentation to enable monitoring and management at execution.

The causal relationship established at development-time between the component representation and the actual code allows reasoning about the application completely in architectural terms. It also enables to control the execution of the application by manipulating the model at runtime, as presented in the following section.

III. CAUSALITY CONTROL

The component model and the causality control in Apam have been primarily intended to monitor (sensitivity) and drive (operationality), at high level, the execution of applications on top of a service platform. The represented SUS is the execution of application services. Those services in turn can represent the state of devices, the sensed activity or actions provided by actuators.

The service execution layer is based on the OSGi [1] and iPOJO [3] platforms. OSGi provides the basic mechanisms for deployment, live update and dynamic service discovery. iPOJO provides the component container and dependency injection mechanisms. In the execution platform, an application is, at any given point in time, a particular assembly of concrete OSGi service instances. The execution

Copyright (c) IARIA, 2014. ISBN: 978-1-61208-367-4

platform handles the deployment and instantiation of the actual Java code of services, and service binding is automatically performed by the iPOJO container, using injected fields in the class of the client service.

At the representation level, the application is represented as a dynamic and reconfigurable architecture, composed of component instances linked by wires. The APAM platform controls the execution by continuously resolving the required dependencies and changing the model.

To effectively drive the execution of the application, this layer must be causally related to the actual service execution layer, as illustrated in Figure 3. Each change of the architecture, like creating components and wires is transformed into actions in the execution platform. For example, creating a wire from source instance *s* to target instance *t* at the architectural level produces the injection of the address of *t* into the fields of *s* in the Java code of the *s* implementation. In this regard, the Apam representation is a virtual machine executing the architectural application description, on top of the underlying OSGi execution platform.

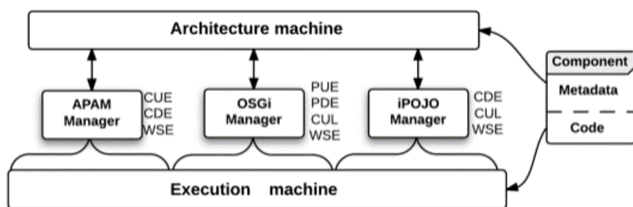

Figure 3. Causality Managers.

However, in a dynamic and ubiquitous context there are a number of external and uncontrolled events that may affect the application execution; for instance, new devices can be discovered / removed that need to be integrated / removed in the application, legacy components can be installed in the platform offering services required by the application, other applications can be installed that may interfere with the application. The execution platform automatically detects three kinds of changes:

- Components that appear and disappear
- Property changes of a component
- Service binding request from a component

The representation also manages components, properties and wires; indeed, in some cases, context changes, detected by the execution platform must be transformed into the corresponding change in the representation. For example, the apparition of a new device is important for the application architecture since it may trigger application adaptation to this new context (like making use of such a device). This requires bidirectional synchronization between the two platforms; that is the responsibility of the causality managers.

A. Causality Managers

In a top-down view of the execution, the application description presented in Section II is a specification that must be enforced in the execution platform. In a bottom-up view, the context changes detected by the execution platform must be represented in the architectural layer, in order to trigger the appropriate adaptations. There is thus a need to enforce a causal relationship between the two platforms.

Both platforms share the concept of components having properties and wires; at different levels of abstraction. For the architecture platform, a component is a description (its metadata), and wires are relationships between these descriptions; while for the execution platform, components are classes and objects, and wires are addresses into Java fields. Properties are similar in both platforms.

Causality managers are in charge of keeping the two platforms synchronized. Each causality manager is driven by a model (illustrated in the middle part of Figure 3) expressing its synchronization strategy along three axes:

1) What to change: as expressed above, the three shared concepts to synchronize are components, properties and wires (labeled C, P, W respectively in the figure).
2) Direction to change: a causal manager may be operational, propagating changes from the architectural platform to the execution platform, (labeled D for Downwards); or sensitive, propagating changes from the execution platform to the representation (labeled U for Upwards). In some cases, both platforms collaborate to take a decision; (labelled S for Symbiotic).
3) When to change: propagation can be Eager (labeled E), meaning that it happens as soon as the change occurs, or it can be Lazy (labeled L), meaning that the propagation will be done only on demand by the other platform.

For components directly specified using the APAM component model, at development time, the Apam compiler automatically includes the metadata described in Section II. The Apam causal manager extracts this metadata from the packaged component, and builds the corresponding component in the architecture platform.

For other legacy component technologies, a causal manager is in charge to extract the available information and to build the corresponding architectural object. However this requires a deep knowledge of each technology, hence a specific manager (for instance, the legacy OSGi and iPOJO managers in the figure).

B. The Apam Causal Manager

The strategy used by the native Apam component is an immediate causality for components: CUE and CDE, i.e., as soon as an Apam component appears (C for component), whether in the architecture or execution platform (Upward, and Downwards), it is immediately (Eager) synchronized on the other platform.

The code of Apam native components is injected to intercept all references to the fields of the required dependencies. The need to resolve a wire (Wire) is detected by the execution platform which decides, in symbiosis with the architecture platform (Symbiotic) to immediately (Eager) resolve the wire in both platforms, hence WSE synchronization. Properties are not synchronized since they are only known and used by the architecture platform.

C. Legacy Causal Managers

For the OSGi causal manager, the available information is limited to the properties published in the OSGi registry; and properties can be modified in both the architecture and execution platforms, hence the PUE, PDE synchronization: Properties are synchronized Upward and Downward Eagerly. OSGi component can be created only by third parties in the execution platform (they do not have factories), hence the CUE synchronization.

iPOJO causal manager is still another case: iPOJO factories can be used to create and instantiate components at the architecture and at the execution platform layers. Components created by the architecture platform must immediately affect execution, hence the CDE synchronization. Conversely, legacy iPOJO components are synchronized up only when required: CUL synchronization. Wires are Symbiotically, Eagerly synchronized (WSE). Properties are only visible and used in the architecture platform, and thus are not synchronized.

IV. PROVISIONING EXTENDED CAUSALITY

Thanks to the sensitivity property, in our case, the representation allows monitoring the services currently running in the execution platform and deployed by third parties, using platform specific mechanisms. The operationality property requires the capability to add/remove/create entities (components and instances) at the representation level, not only to manage those already existing in the execution platform.

To satisfy this requirement, Apam includes the capability to perform component provisioning. At the representation level, this provisioning capability is used to satisfy the dependencies of the application, when a resource is required. In practice, to find the needed component(s) and resources the Apam kernel looks into a number of search spaces.

Search spaces in turn are mapped to concrete service repositories, of diverse and open-ended nature: it may include components repositories, existing cloud services, networked devices, or even remote platforms.

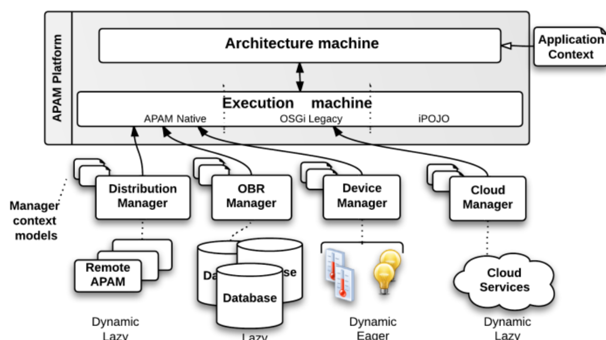

Figure 4. Provisioning managers.

Apam proposes provisioning managers as an extensible mechanism to control the search spaces. Figure 4 shows the currently defined provisioning managers, with their behavior.

A. Provisioning managers

We qualify as provisioning managers the managers in charge of synchronizing the execution machine with other platforms. We call platform any repository containing services that provisioning managers can access from the execution platform; directly by deployment, or indirectly through a proxy. Provisioning managers synchronize an “external” platform with the execution platform. We distinguish Lazy vs. Eager and Dynamic vs. Static behaviors for provisioning managers. Lazy and Eager have been already discussed; Dynamic means that changes in the external platform are “immediately” synchronized with the execution platform; Static means that changes, if any, are not synchronized.

B. Causal provisionning managers

Eager and Dynamic provisioning managers are those that define what constitutes the execution context of the application, since each change in their platform is immediately perceived by the execution platform, which, depending on its causality, the manager transfers its perception to the architecture platform. Here, the context is made of Apam and all the devices controlled by the device manager. Other context managers can be defined and added (dynamically or not) to the Apam system.

C. Lazy managers: an extended search space

The Lazy managers define the search space in the following way: when Apam tries to resolve a wire, it looks for a satisfactory target in the architecture machine. If the target is not found Apam delegates resolution to available Lazy managers, because lazy managers may know components in their platform that are not (yet) present in the architecture machine. These managers must implement the method *resolve (Dependency d, Composite context)* which returns if found an instance in the execution machine satisfying the dependency d in the provided context.

The architecture machine invokes each lazy manager in their priority order until one returns an instance *t’*. Apam invokes the relevant causality manager to reify *t’* as a *t* instance in the architecture machine, and return *t* as the resolution solution. If no manager finds a solution, the resolution fails.

Many Lazy managers can be defined, Apam provides with the standard distribution the OBR manager that can deploy components from a list of bundle repositories; the Distribution manager that looks for a component in another remote Apam machine and returns a proxy towards the selected remote component. The Cloud manager, based on the Rose framework [4], returns a proxy toward a remote service (WS, etc.). Other managers can be defined; they will be called if registered as a dependency manager.

V. FAILURE HANDLING

Even with different search spaces, it is possible that the execution platform fails to find a suitable service to satisfy the dependencies of an application component.

In Apam, the failure reaction is specified at the architecture level, in the component model. A dependency

declaration can express what should be the policy in case of failure. The currently supported policies are:

- Optional dependency: nothing is done; the field (wire source) will contain “NULL” as target value.
- Wait (duration): the current thread for which a wire could not be resolved is halted until a valid target is found, or until the time limit is reached.
- Exception (name): throws the exception specified by the component.
- Hide: the source component is hidden for all subsequent resolution; all its incoming wires are broken, which may hide its clients and so on.

The hide strategy allows to backtrack the current architecture (as long as dependencies are in the hidden mode) and thus to explore automatically a wide range of possibilities. But since all current architectures must be valid, it is not allowed to remove (hide) components that are explicitly part of the Application architecture.

VI. VALIDATION AND EXPERIENCE

Owing to its flexibility, adaptability and reliability, Apam has been experimented, by academic and industrial teams, as the central layer (often referred as the “dynamic middleware”) of two large projects for home automation.

In OpenTheBox project, Apam is mostly used as the central manager for set top boxes, in charge of providing isolation, controlled collaboration between applications[5], including the conflicting accesses to the shared devices [6]. In this case, each application is modeled as a composite, and the contextual properties described in Section IV allow specific policies for each application to be applied.

In the AppsGate project, the set top box is powerful enough to support high level services, advanced functionalities and innovative user interactions. In this project, Apam builds an abstract “model of the world” based on sensors and devices. The high level services are defined as applications at specification and implementation levels, and the execution automatically links the service to the relevant devices.

VII. RELATED WORK

The use of models to represent a system at an appropriate level of abstraction is generalized in software engineering. However, as systems become more dynamic and directly related to the physical world, there is a need to carefully consider the representation relationship, as discussed in [7].

Our approach can be regarded as an example of the general principle of models at runtime [8]: the Apam architectural description is a model of the underlying physical execution. As explained, this model is both an abstract representation (sensitivity) and a prescriptive specification (operationality) of the reality [9].

The abstract Apam application description is a model of the valid space of application’s configurations, which evolves by changes at both the execution and component level. Thus, Apam model can be characterized as a “Configuration space and variability model”, according to the classification by Vogel et al.[10].

Apam uses architectural models as enabling technology for runtime adaptability. As such, it can be related to many works in dynamic architectures [11][12]. The main idea that we borrowed is that runtime reconfiguration must be reasoned and performed at the architectural level.

If we consider a top-down approach, based exclusively on operationality, the application model is a prescription of the execution, and, the Apam component meta-model can be regarded as an Architecture Description Language. Our meta-model combines the classical concepts of Software Component Models [13] with the intrinsic evolution typical of Service-Oriented Computing [14], in which the concrete architecture is incrementally built as new services are required or made available and bound at execution. In this respect, our proposition can be related to other structural service composition approaches, like SCA [15] or CALM [16], however, these approaches do not define any runtime reconfiguration mechanisms.

We can also think of the Apam runtime platform as a middleware that manages the application execution. Our approach shares then similar goals with reflective middleware platforms [17] that propose an introspection layer that reifies in a causal model the execution elements.

Similarly, some component models propose a reflective runtime to allow introspection and reconfiguration [18][19]. The main difference is that these approaches make the implicit assumption that architecture evolution is an exogenous process, performed by external agents, like administrators or autonomic managers. In our vision, architecture evolution is a continuous, endogenous process, intrinsic to the execution of each application.

Other experimental platforms have been designed specifically for ubiquitous computing. For example, DiaSuite [20] proposes a domain-specific component model to describe the architecture and properties of Sense/Compute/Control applications. The specialized model enables static analysis and verification, beyond what is proposed in Apam, however it doesn’t manage runtime dynamicity.

Without surprise, it was the double synchronization (upward and downward) that raised the most difficult technical issues, and the trickiest bugs. Indeed, conflicting changes on the “same” entity can happen “simultaneously” in the model and in the platform. A large fraction of the code is dedicated to solve (reconcile, choose, merge, prevent, notify, etc.) these special cases. It also explains that full causality is difficult to provide, and indeed, is not often provided.

VIII. CONCLUSION

Best practice in software engineering emphasizes the need to work with representations that are simple, homogeneous and at the relevant abstraction level. For that reason, many techniques like levels of abstraction or modeling have been developed. In all case, there is the need to closely control the relationship between the representation and the system represented. Most often, this relationship is operational only: the changes performed on the representation are propagated to the underlying system, supposed to be passive.

The ever increasing interpenetration of the numeric world and our life (smart phones, ubiquitous computing, home automation, etc.) makes abstract representations even more needed, but in the same time the represented system is dynamic, autonomous and its changes are unpredictable. In this case, both the system and its representation are active and both are subject to unpredictable changes, possibly simultaneous and incompatible. Therefore, the representation must be both operational and sensitive, i.e., causal. The realization of a causal representation is very challenging, but at the same time, it is almost needed if reliable applications are to be developed in such a context. The issue we have addressed is a general approach to the development of a causal representation.

In our work, the representation is similar to a traditional model but metaclasses can be explicitly associated with the kind of entities they represent. This association is extensible in the sense that it is implemented in the form of plug-ins: Maven plug-in at development time and Apam causality managers at run time. The platform knows the association and dynamically delegates the causality management to currently plugged-in managers.

In our experimentation, the system represented is an OSGi service platform. It is a limitation because the entities represented are 1) only services, and 2) only those service currently running in OSGi. We have overcome these two limitations by making "everything" a service (proxies, sensors, applications, etc.) and extending the OSGi platform by an extensible provisioning layer, also made of plug-in managers very similar to causal managers. An entity required in the representation layer is automatically deployed in the system (OSGi), and by causality it is created into the representation. The different extensibility mechanisms (causal metaclasses, causal managers, provisioning managers) provide a fairly general framework for the development and management of a causal representation.

The experience shows that causality can be provided systematically and efficiently making much more feasible the reliable development of the new kind of applications like ubiquitous computing.

The Apam platform is available in open source, see [21].

ACKNOWLEDGMENT

Parts of this work have been supported by the European CATRENE project AppsGate and the French ANR "Investissements d'Avenir" project Open-The-Box.

REFERENCES

[1] OSGi Alliance, "OSGi Service Platform Core Specification Release 4", Aug. 2005. [Online]. Available from http://www.osgi.org [retrieved: July, 2014]

[2] E. Damou. "ApAM : A development and execution environment for ubiqutous applications". PhD dissertation. Université de Grenoble, France, Oct. 2013. [In french] [Online]. Available from http://tel.archives-ouvertes.fr/tel-00911462 [retrieved: July, 2014]

[3] C. Escoffier, R. S. Hall, and Ph. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework", in Proceedings of the International Conference on Services Computing, pp. 474-481, July 2007.

[4] J. Bardin, C. Escoffier, and Ph. Lalanda "Towards an Automatic Integration of Heterogeneous Services and Devices", in Proceedings of the Services Computing Conference, pp. 171-178, Dec. 2010.

[5] J. Estublier and G. Vega. "Managing Multiple Applications in a Service Platform", in Proceeding of the ICSE workshop on Principles of Engineering Service-Oriented Systems, pp. 36-42, June 2012.

[6] J. Estublier, G. Vega, and E. Damou. "Resource Management for Pervasive Systems", in Proceedings of the International Conference on Service Oriented Computing, Lecture Notes in Computer Science, vol. 7759, pp. 368-379, Nov. 2012.

[7] M. Jackson, "Aspects of abstraction in software development", Software & System modeling, vol. 11, no. 4, pp. 495-511, Oct. 2012.

[8] G. Blair, N. Bencomo, and R. B. France, "Models@ run.time", IEEE Computer, vol.42, no.10, pp. 22-27, Oct. 2009.

[9] P-A. Muller, F. Fondement, B. Baudry, and B. Combemale. "Modeling modeling modeling", Software & System modeling, vol. 11, no. 3, pp. 347-359, July 2012.

[10] Th. Vogel, A. Seibel, and H. Giese, "The Role of Models and Megamodels at Runtime", in Proceedings of the Workshop on Models in Software Engineering, Lecture Notes in Computer Science, vol. 6627, pp. 224-238, Oct. 2010.

[11] P. Oreizy and R. Taylor, "On the role of software architectures in runtime system reconfiguration" Software, IEE Proceedings, vol.145, no.5, pp.137-145, Oct. 1998.

[12] J. Magee and J. Kramer, "Dynamic structure in software architectures", in ACM SIGSOFT Software Engineering Notes, vol. 21 Issue 6, pp. 3-14, Nov. 1996.

[13] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A Classification Framework for Software Component Models", IEEE Transactions on Software Engineering, Vol 37, No. 5, pp. 593-615, Sept. 2011.

[14] J. L. Fiadeiro and A. Lopes, "A Model for Dynamic Reconfiguration in Service-oriented Architectures", in Proceedings of the European Conference on Software Architecture, Lecture Notes in Computer Science, vol. 6285, pp. 70-85, Aug. 2010.

[15] OSOA, "Service Component Architecture Assembly Model V1.00", March 2007. [Online]. Available from http://www.oasis-opencsa.org/sca-assembly [retrieved: July, 2014]

[16] G. Jung and J. Hatcliff, "A type-centric framework for specifying heterogeneous, large-scale, component-oriented, architectures", Science of Computer Programming, vol. 75, no. 7, pp. 615–637, July 2010.

[17] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case-for reflective middleware", Communications of the ACM, vol. 45, no. 6, pp. 33-38, June 2002.

[18] M. Léger, T. Ledoux, and T. Coupaye, "Reliable Dynamic Reconfigurations in a Reflective Component Model" in Prooceedings of the International Symposium on Component-Based Sofware Engineering, Lecture Notes in Computer Science, vol. 6092, pp. 74-92, June 2010.

[19] L. Seinturier, Ph. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J-B. Stefani, "Reconfigurable SCA Applications with the FraSCAti Platform" in Prooceedings of the International Conference on Service Computing, pp.268-275, Sept. 2009.

[20] D. Cassou, E. Balland, C. Consel, and J. Lawall. "Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications". in Proceedings of the International Conference on Software Engineering, pp. 431-440, May 2011.

[21] Apam [Online]. http://adeleresearchgroup.github.com/ApAM [retrieved: July, 2014].