Self-adaptation of event-driven component-oriented Middleware using Aspects of Assembly

Daniel Cheung-Foo-Wo, Jean-Yves Tigli, Stéphane Lavirotte, Michel Riveill

Self-adaptation of event-driven component-oriented middleware using Aspects of Assembly

Daniel Cheung-Foo-Wo*
I3S (UNSA - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
cheung@polytech.unice.fr

Jean-Yves Tigli
I3S (UNSA - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
tigli@polytech.unice.fr

Stéphane Lavirotte**
I3S (UNSA - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
stephane.lavirotte@unice.fr

Michel Riveill
I3S (UNSA - CNRS)
930 Route des Colles - BP 145
06903 Sophia-Antipolis France
riveill@polytech.unice.fr

ABSTRACT

Pervasive devices are becoming popular and smaller. Those mobile systems should be able to adapt to changing requirements and execution environments. But it requires the ability to reconfigure deployed codes, which is considerably simplified if applications are component-oriented rather than monolithic blocks of codes. So, we propose a middleware approach called WComp which federates an event-driven component-oriented approach to compose services for devices. This approach is coupled with adaptation mechanisms dealing with separation of concerns. In such mechanisms, aspects (called Aspects of Assembly) are selected either by the user or by a self-adaptive process and composed by a weaver with logical merging of high-level specifications. The result of the weaver is then projected in terms of pure elementary modifications of components assemblies with respect to blackbox properties of COTS components. Our approach is validated by analyzing the results of different experiments drawn from sets of application configurations randomly generated and by showing its advantages while evaluating the additional costs on the reaction time to context changing.

Categories and Subject Descriptors

D.2.13 [**Soft. Eng.**]: Reusable —*Adaptive software*

Keywords

Dynamic self-adaptive CBSE, aspect-oriented, context awarness

*also CSTB 290, Route des Lucioles, BP 209 06904 Sophia Antipolis, France

**also IUFM Célestin Freinet - Académie de Nice 89, Avenue George V - 06046 Nice Cedex 1, France

1. INTRODUCTION

Though well acquainted with personal computers, we are less aware of the invisible computers at use in our homes; they help us communicate and entertain ourselves. Weiser [14] introduced a "ubiquitous" computer science and stated that IT[1] was not ready to be part of that environment. Although systems become smaller and more numerous and participate in new software applications [12], they still raise complex scientific problems requiring an autonomic-evolving structure to *react* to the movement of devices and devices themselves. By autonomic structures, we mean structures managing themselves along with high-level objectives. Kephart [9] highlighted the main scientific challenges of autonomic computing. We address two of those challenges in this paper: *device and network heterogeneity* and *dynamicity*. A part of this dynamic world is modeled into an environmental and an operating *context* which are *partially unknown*. Moreover, dynamic adaptation requires the ability to reconfigure executing codes. Therefore, such reconfiguration is considerably simplified if adaptive and pervasive applications are component-oriented rather than monolithic blocks of codes. The scope of this paper can now be outlined briefly. (Sec. 2) We will first draw a state of the art on self-adaptive pervasive systems according to the following criteria: *event-driven model* which permits responding to the required reactivity of pervasive computing, *service-oriented architecture* which allows responding to the required dynamicity, and *aspect-oriented methodology* which allows responding to the required modularization and heterogeneity. (Sec. 3) In our architecture, the behavior of the pervasive system is controlled by a set of schemas. We will start with the characteristics of the event-driven software component model, will then present the schema-based self-adaptive architecture and the self-adaptation algorithm which control self-adaptive pervasive systems. (Sec. 4) Finally, we will validate our approach by commenting the results of diverse experiments.

2. SELF-ADAPTIVE MIDDLEWARE APPROACHES

[1]Information Technology

Designing software architectures for a self-adaptive pervasive system (SAPS) is an ongoing research problem which can be resolved only after numerous systems have been designed and tested in situation. Although this research has not yet reached maturity, we can list a set of desired principles of a SAPS. We have divided those principles into two groups: principles which describe the design principles and principles which determine the sensitivity of the program to its surrounding context.

Design principles. Some specific design principles are required for SAPS. Unlike most conventional software engineering projects which begin from a set of requirements, SAPS projects must constantly reevaluate their requirements. Therefore, systems which can easily be modified are highly wanted. We have identified the following principles:
• *Modularity.* Following a general requirement of complex systems, a SAPS should be divided into smaller subsystems that can be designed and debugged separately.
• *Expandability.* Because it takes time to design and test individual components, an expandable architecture is desirable as it facilitates incremental implementation of the diverse situations that pervasive systems need to cope with.
• *Separation of concerns.* Pervasive applications can be modified according to a particular concern which can be duplicated at several places in the code.
• *End-user programming.* End-user programming supposes well defined borders in the system usage. The development of customized pervasive systems must be constituted of process cycles where programmers can develop functionalities that can be used by others [13].
• *Multiple-task resolution.* For SAPS, situations requiring conflicting concurrent actions are inevitable, and the system should provide the means to fulfill multiple objectives.

Context-awareness principles. Context-awareness is important in the design of SAPS. Context is traditionally associated to the localization in space where mobile systems are expected to evolve [11], but can also be largely extended. We have identified context-awareness principles:
• *Robustness.* A system's robustness is its ability to handle imperfect inputs, unexpected events, uncertainties, and sudden malfunctions. Moreover, devices in the pervasive environment can be restricted with regards to processor power, energy, memory, display functionality, and so on.
• *Multi-device integration.* This ability is crucial to reliable SAPS behavior. The system architecture must compensate for the limited accuracy, reliability, and applicability of individual devices by integrating several complementary devices.
• *Reactivity.* Because the real-world environment is unstructured, SAPS should make few assumptions about its dynamics and react to environmental changes. In our system, adaptation schemas are reactive modification specifications.

Existing systems. Adaptation requires the ability to reconfigure the deployed code, which is considerably simplified when applications are component-oriented rather than monolithic blocks of codes. Numerous component-oriented systems have been designed in order to partially respond to SAPS' problems. We distinguish event-driven, service-oriented, and aspect-oriented systems. By *event-driven* we mean systems notified by significant change, by *service-oriented* we refer to the architecture based on service descriptions and interactions, and by *aspect-oriented* we refer to the methodology enabling separation of concerns. Table 1 shows their relative strengths and weaknesses. We see that none can be used alone, but only a combination can meet our event, service, and aspect specifications.

Event-driven systems. Event-driven architectures have been used for SAPS and reconfigurable systems for many years. Their common distinctive feature is the weak-coupling of components meaning individual components do not know the components realizing their required functionalities at design time. The information is set at runtime either by the component itself or another one. The first case is illustrated by the reflective component model OpenCOM v2 where new types of components can be added and function calls can be altered by modifying a *process vtable* [4]. The second case is known as the principle of *Inversion of Control* that has been experimented in a *lightweight container* in [1, 6] as an interactive adaptive system. Weak-coupling offers a high degree of expandability but its relatively low level of abstraction does not allow complex software design.

Event-driven systems are not suitable for very complex design, but adequate for reactivity and dynamicity.

Service-oriented systems. The appealing features of service-oriented systems are their flexibility in handling dynamicity and their suitability to the integration of new devices. [15] provided logical primitives to transfer codes so as to reconfigure software systems and enhance robustness. [8] suppressed a level of complexity by introducing the self-adaptive component model K-Component which enables individual components to adapt to changing environments through a complex decentralized coordination model which simplified the integration of multiple objectives and allowed groups of components to collectively adapt their behavior. [10] focused on the configuration and integration of devices in pervasive computing scenarios which include self-organizing configuration for pervasive computing environments supporting unskilled installation. They coupled a domain specific language (DSL) and middleware but with a centralized approach.

Service-oriented systems allow robustness, coordinating services in a programmatic decentralized collaboration.

Aspect-oriented systems. Aspect-oriented systems consist of a set of join points, pointcuts, advice, and weaving loops, which operate at runtime or design-time to construct an executable program. Their dominant characteristics consist in considering adaptations as cross-cutting components and in weaving them as classical AOP aspects. [7] designed a DSL and expressed adaptation concerns as aspectual components in order to monitor self-adaptive systems. He also proposed to express pointcuts in terms of binding scripts. However, this approach does not provide a collaborative combination and does not avoid semantic conflicts by the bindings declaration.

Aspect-oriented systems provide an enhanced modularity as they include separation of concerns, but are not intended to achieve service collaboration.

Table 1 reflects the need to integrate all the introduced principles to build SAPS. We propose our middleware approach called WComp taking into account, at best, all the previ-

Table 1: Comparison of self-adaptive approaches

	Event	Service	Aspect
Modularity	Medium	Medium	High
Expandability	Yes	Yes	Yes
Sep. of concerns	No	No	Yes
End-user prog.	Medium	Yes	No
Multiple-task	Difficult	Yes	Medium
Robustness	Yes	Yes	No
Multi-device	Yes	Yes	No
Reactivity	High	Medium	Medium

ously explained principles for pervasive application design. Firstly, it federates an event driven, component based approach to compose web services for devices. Secondly, it introduces a self-adaptation approach dealing with separation of concerns, using the aspect of assembly concept, logically mergeable in case of conflicts.

3. OUR SELF-ADAPTIVE MODEL

With the ambition of building an expandable multi-device system capable of multiple-task conflict resolution, we have developed an extended component-based model (Sec. 3.1) composed of adaptation mechanisms as aspects selected either by the user or by a self-adaptive process and composed by a weaver with logical merging of high-level specifications (Sec. 3.2). The result of the weaver is projected in terms of pure elementary modifications (PEMs) – add, remove components, link, unlink ports – with respect to blackbox properties of COTS components. We call those adaptation mechanisms, Aspects of Assembly (AAs) which rely on the event-driven model of the lightweight component framework in our middleware platform called WComp [6].

3.1 Event-driven composition for services for devices

The WComp component model is a slightly modified JavaBeans model adapted to other programming languages with the concepts of input and output ports, properties, and hierarchy. Still a class instance, but not necessarily serializable, a component has a unique name and an interface which has two sets composed of events[2] and methods. We consider C the set of components, E the set of events characterized by their unique name, and M, the set of methods. We gather the declaration of events and methods in the term 'port'. We consider a set of links L which are lists composed of an event and of a list of methods. An assembly consists of a subset of C and L. The *container component* implements an API to dynamically control this assembly, and consequently the addition and removal of elements in C and L. Roughly speaking, we must use event – also known as late-bindings, "*push*" *mechanism*, or *Inversion of Control* – in lightweight containers which is now shared characteristics of adaptive component models [3]. The assembly can be modified by using an *assembly designer* as shown in the top window in Fig. 1. And finally, the model is hierarchical. Composite components can then be constructed by decorating a lightweight container with adequate ports and populating the container with composite components and links.

[2]event's name are prefixed by '^'

This way, the interior of components can be addressed and reconfigured using common existing designers.

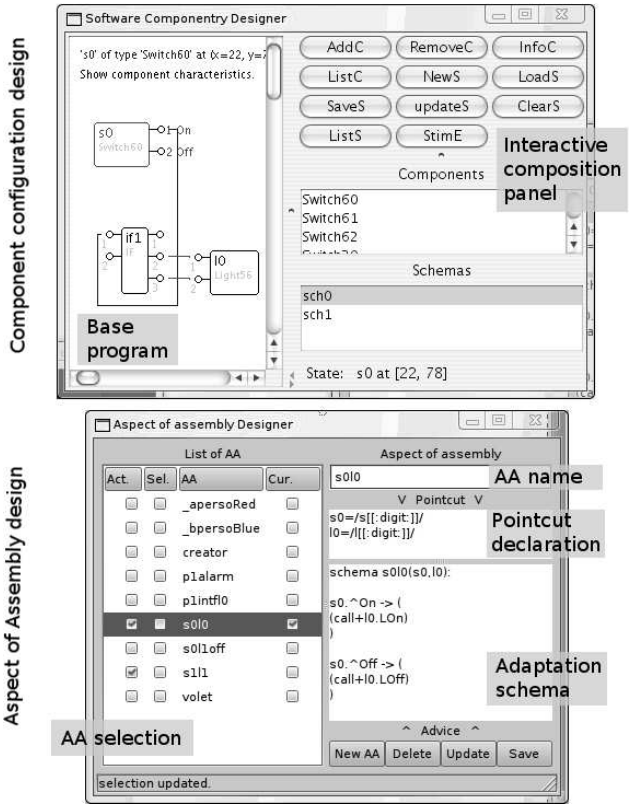


Figure 1: Screen capture of designers

3.2 Aspects of assembly

We propose a component-oriented integration which takes into account the adaptation characteristics in Sec. 2. Our architecture is twofold: it consists of an extended model of AOP for adaptation schemas and of a weaving process with logical merging. We implemented a toolkit (Fig. 1) which includes AAs as the central concepts. We introduce here concepts used in the rest of the paper:

Base assembly: an assembly of components.
Join point: components and ports of the base assembly.
Pointcut: a description of a set of join points for a particular adaptation advice.
Adaptation advice:adaptation schema describing architectural reconfigurations.
Weaver: mechanism integrating advice according to specified pointcuts selecting join points from a base assembly. It is also responsible for the merging of conflicting advice.

An AA is structured as an aspect with a pointcut and advice (adaptation schema) which is specified in a DSL using interaction specification firstly defined in [5]. This DSL has been then enhanced in [6] to integrate event-driven declarations. With our approach, self-adaptive pervasive software developers can reason, plan, and validate AA-based assemblies at all stage of the development phase. Using logical predefined

validation rules, logical configurations' incompatibilities can be detected at runtime.

Advice. We present an example of advice which is used in a practical situation for response-time on observed components. The schema called 'Ex' redefines an input and an output port and is applied to a set of components symbolically represented by the *observed* and *timeout* variables:

```
1  SCHEMA Ex (observed, timeout):
2      observed.^Out ->
3             (  IF ( timeout.Check ) CALL )
4      timeout.Check ->
5             (  timeout.Start ; CALL      )
```

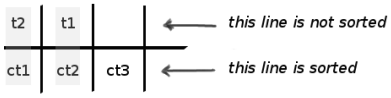
Description. Firstly, it redefines the *^Out* output of the *observed* component, which specifies that actions possibly defined in the base assembly are executed only if the *timeout* component gives its authorization. Secondly, it redefines the *Check* input of *timeout*, which specifies that before the execution of the input possibly required by other components, *timeout* must be started, i.e. the *Start* input must be executed.

We defend a minimalistic approach in order to be able to cope with scalability. And for this reason, those specifications are translated into a set of PEM. Any modification can be regarded as an assembly-to-assembly transformation. Thus, the AA designer depicted in the bottom window in Fig. 1 communicates its PEM to a *container* (Sec. 3.1).

Pointcut. We define pointcut descriptions as sets of filters on base assembly meta data – component ID, their types, etc. Those filters construct a list of parameters satisfying the list of variables of a schema for the latter to be integrated in the base assembly. If only one list is constructed, the schema is integrated only once in the base assembly and the symbolic variables are syntactically replaced in the schema to match the base assembly join points. If several lists are constructed, the schema is duplicated and each set of variables are respectively replaced. For our experiments, we choose for convenience to express filters in the AWK language [2] and define a simple grammar to make AWK responses correspond to schema variables: '<variable>:=<AWK filters>;...'. Example:

```
1     observed := /t*/ ;
2     timeout :=
3         /ct*/  { a[substr($1,3)]=$1 }
4         END    { for(i=1;i<=NR;i++){print a[i]} } ;
```

Description. The *observed* variable is matched against component ID starting with '*t*' and *timeout*, against those starting with '*ct*'. The second filter (lines 3-4) consists in two matchings: component ID beginning with '*ct*' and the 'END' of the component ID list. Both matchings are completed with a program between braces. The first program assigns a unidimensional table '*a*' with a matched component ID – represented by '$1' – at an index specified by the numerical part of the matched string 'ct*' which corresponds to the substring starting at position 3 (ex: the index of *ct*42 is 42). Then, in the second program (line 4), the sequence of components ID beginning by 'ct' is returned to the AA designer orderly from 1 to NR[3]. The order of the components is not specified and can be random when a specific program in AWK is not written (line 1). In this example, the first pointcut is unordered and the second is ordered. We consider a base assembly composed of five components: *ct*1, *ct*2, *ct*3, *t*1, and *t*2. The schema is duplicated into two applicable schemas (Ex1, Ex2). The global result is a two dimensional table whose duplicated schemas' parameters the columns represent:



Consequently, in the two duplicated schemas Ex1 and Ex2, the parameters of Ex1 and Ex2 are not associated with the parameter with respectively the same ID: t2 is rather associated with ct1 and t1 is associated with ct2.

```
1 SCHEMA Ex1(t2,ct1):
2  t2.^Out ->
3   (IF(ct1.Check) CALL)
4  ct1.Check ->
5   (ct1.Start ; CALL)
```

```
1 SCHEMA Ex2(t1,ct2):
2  t1.^Out ->
3   (IF(ct2.Check) CALL)
4  ct2.Check ->
5   (ct2.Start ; CALL)
```

The decision to integrate adaptation advice according to specified pointcut follows the following rules: (1) only the first complete columns of the table become parameters of the duplicated schemas (in this example, only the two first columns became parameters). (2) the order of the ID in the first line {*t*2, *t*1} can change. Therefore, to apply a schema deterministically, lines must be sorted.

Weaver with logical merging. The logical integration rules can be represented by a matrix representing the two-by-two merging of the operators introduced in the beginning of Sec. 3.2. We give few examples of logical rules in Fig. 2. We present an example of weaving two schemas called 'Ex' and 'AA0' (see line 1 and 6 for their code). Hypothesis: two pointcuts respectively specifying the '*observed*' variable and the '*worker*' variable are in conflict (they produce the same join points):

```
1  SCHEMA Ex (observed,timeout):
2    observed.^Out ->
3        (  IF ( timeout.Check ) CALL  )
4    timeout.Check -> ( timeout.Start ; CALL )

5  SCHEMA AA0 (producer,worker,consumer):
6    producer.^Out -> ( worker.In )
7    worker.^Out -> ( consumer.In )
```

Merging example. The specification rules (SR) at line 4 and 6 are not conflicting. Thus, they are copied in the resulting schema (line 5 and 6). However, the SR at line 2 and 7

[3]number of component ID given at the input of the pointcut

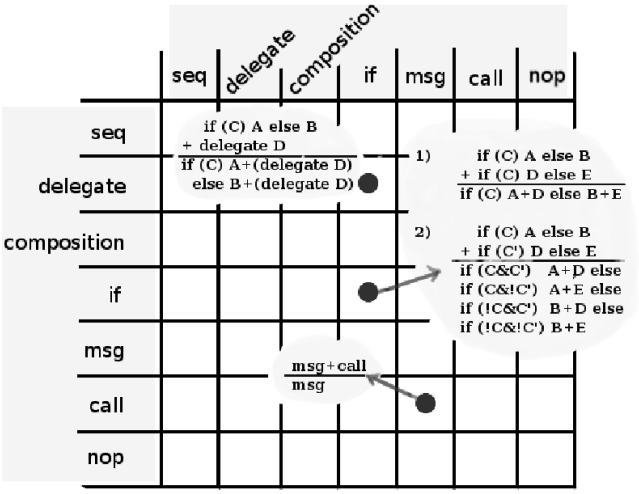


Figure 2: Operator merging matrix

are conflicting because they redefine the same output *ˆOut* of the confounded *observed/worker* component. Therefore, their respective specification programs are logically merged and the resulting 'AA0+Ex' schema is calculated using the merging matrix (Fig. 2). The '+' operator corresponds to the unordered or undetermined couple of operations to execute. The merging process replaces *CALL* at line 3 of 'Ex' by *CALL + consumer.In* in 'AA0+Ex'. Finally, the resulting AA is translated into a set of PEMs. For instance, the operator *IF* is translated into the addition of a generic component of type IF.

```
1  SCHEMA Ex+AA0 (observed,timeout,
2                 producer,consumer):
3    observed.^Out ->
4      ( IF ( timeout.Check ) { CALL + consumer.In } )
5    timeout.Check -> ( timeout.Start ; CALL )
6    producer.^Out -> ( observed.In )
```

We saw the AA-specific design process as well as one cycle of the adaptive pervasive application. In the next section, we present the process cycles used to perform self-adaptation.

3.3 Self-adaptation cycles

Self-adaptation consists in reacting to modifications from the user or the environment. Self-configuration is processed by the decoupled AA designer. We describe the user-driven approach and the process which permits to adapt the application to its environment (Fig. 3).

The *user-driven adaptation* consists in selecting or deselecting AA in order to integrate or erase certain behaviors and functionalities in the system. The user can also intervene on the base assembly and operate directly on the assembly. Concerning the area of end-user programming, we distinguish expert and end users. Expert users can design new AAs for new situations whereas end-users do not have to create AA, but only select predefined AA. In that case, the interaction with the user is simplified. The *context-driven* consists in scanning the underlying infrastructure periodically in order to verify if devices are still present in the environment. New devices can asynchronously inform the system of their presence by broadcasting a notification. Therefore, when a device is removed from the system's environment, the software component representing the device is unlinked and removed from the base assembly. Conversely, when a new device appears, a new software component representing this new device is added to the assembly. Consequently, the self-adaptation process consists in detecting those structural changes in the base assembly and each cycle of the process checks if either new AA are applicable, or applied AA are not valid anymore. If a notable change occurred, it recalculates PEMs to be applied on the base assembly.

However, two cases should be considered when an adaptation calculation occurs. The base assembly can be empty (at least no links between components I/O). In such a case, the application – more precisely the interactions between components – is constructed by iterations of the application of AAs. Conversely, the base assembly can be composed of interconnected components. In that case, before adapting the assembly by iterations of application of AA, the base assembly (under the form of ADL) is translated into an AA which is always selected to be composed so that the composition of PEMs takes into consideration this initial state. For example, the schema 'AA0' explained in Sec. 3.2 is the AA result of the transformation of a base assembly.

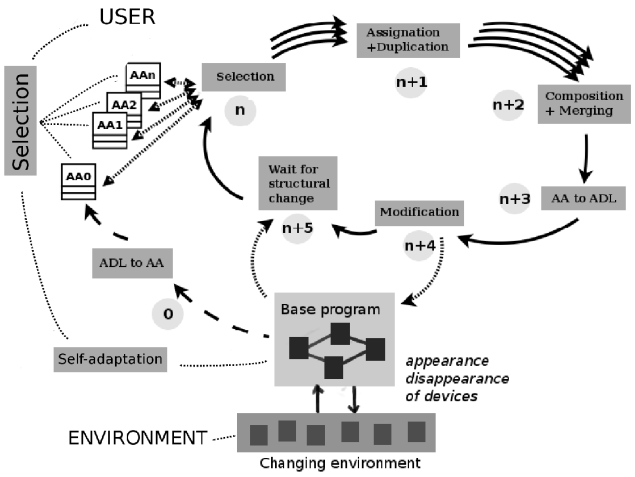


Figure 3: Self-adaptation cycles

Finally, the adaptation process is projected on a set of services and composite services as defined in Sec. 3.1 and is considered as a distributed system. Consequently, selected AAs are duplicated at every level of the application which is bounded by lightweight containers. But, in this paper, we do not focus on this particular property. We have presented our SAPS model by introducing extended aspect concepts and the self-adaptive process. And the next section measures the concepts through comparisons and experiments.

4. VALIDATION

We validate our approach by commenting the results of experiments on sets of randomly-generated assemblies which

show the advantages of AAs while evaluating the additional costs concerning the reaction time when changing context.

Merging process. Our weaver with logical merging combines selected AAs. This approach differs from more classical approaches such as [7, 10] by proposing a higher and logically-composable language intended to be systematically merged into a single coherent program according to a set of logical rules. This program is then translated at runtime to produce a set of pure elementary modifications. This allows an inherent and dispatched distribution of coordination specifications through AA schemas.

Expressiveness of pointcut. We construct an indicator from the number of PEMs to measure the degree to which applying our techniques can simplify adaptation. This measure would show how many elementary modifications are required to approach similar services in component-oriented systems and how easier it is to use each service. For instance, for a set of random programs, the simple AA in Fig. 1 produces 2 PEMs for 1 single integration. For 30 duplications, it reaches an average of 36 PEMs. Our approach allows regrouping elementary modifications into an AA which can then be duplicated by specifying adequate pointcuts. An AA is then defined once and applied n times. Moreover, pointcuts separate the schema from the base assembly by being responsible for assigning the schema variables (application-independent) to join points (application-dependent). This enforces the independence and the reuse of AA schemas.

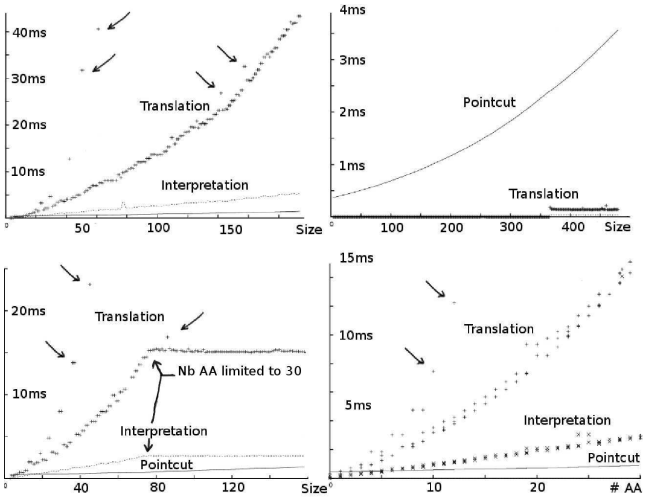


Figure 4: Performances (Intel T2300 1.66GHz)

Performance. The *assembly size* is the number of components and links. The adaptation process is separated: *pointcut* calculation, *composition* of AAs, *translation* from an AA to elementary modifications, and *interpretation* of modifications by the *container*. Overheads are shown in Fig. 4. Pointcut and interpretation grows slowly while translation evolves rapidly according to the number of AAs. Composition-time varies up to 150 ms for 30 AAs. Those experiments show the required reaction time to perform adaptation. The system is highly responsive for a reasonable size of program and number of AAs. One can notice irregularities of logical merging process (in Prolog) on the performance by looking points indicated by arrows due to atoms' garbage collection.

5. CONCLUSION

In this paper, we introduced a middleware approach called WComp which federates an event-driven component-oriented approach to compose services for devices. This approach is coupled with adaptation mechanisms dealing with separation of concerns. In such mechanisms, aspects (called Aspects of Assembly) are selected either by the user or by a self-adaptive process and composed by a weaver with logical merging of high-level specifications. The result of the weaver is projected in terms of pure elementary modifications of components assemblies with respect to blackbox properties of COTS components. We finally commented results indicating the expressiveness and the performance of such an approach, showing empirically that the principles of aspects and program integration can be used to facilitate the design of adaptive application. We further plan to decouple the DSL from the AA concept in order to specify schemas by means of assemblies of components making up 'good practice' schemas. Thus we also intend to generalize AA-merging algorithm allowing the expert user to define its own merging strategies.

6. REFERENCES

[1] M. Ahmed, R. Ghanea-Hercock, and S. Hailes. MACE: adaptive component management middleware for ubiquitous systems. In *Proc. of the 4th Intern. Workshop on Middleware for Perv. and Ad-Hoc Comp.*, page 3, New York, NY, 2006. ACM Press.

[2] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Lang.* Addison-Wesley, 1988.

[3] N. Bencomo, G. Blair, and P. Grace. Models, reflective mechanisms and family-based systems to support dynamic configuration. In *Proc. of the 1st workshop on MOdel Driven Development for Middleware*, pages 1–6, New York, NY, USA, 2006. ACM Press.

[4] G. Blair, G. Coulson, J. Ueyama, K. Lee, and A. Joolia. OpenCOM v2: A component model for building systems software. In *IASTED Software Engineering and Applications*, 2004.

[5] M. Blay-Fornarino, A. Charfi, D. Emsellem, Anne-MariePinna-Dery, and M. Riveill. Software interactions. *Jo. Of Obj. Tech.*, 3(10):161–180, 2004.

[6] D. Cheung-Foo-Wo, J.-Y. Tigli, S. Lavirotte, and M. Riveill. Wcomp: a multi-design approach for prototyping applications using heterogeneous resources. In *17th IEEE Intern. Workshop on Rapid Syst. Prototyping*, pages 119–125, Crete, 2006.

[7] P.-C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive Fractal components. In *Softw. Comp.*, pages 82–97, 2006.

[8] J. Dowling and V. Cahill. Self-managed decentralised systems using K-Components and collaborative reinforcement learning. In *Proc. of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 39–43, New York, NY, USA, 2004. ACM Press.

[9] J. O. Kephart. Research challenges of autonomic computing. In *ICSE '05: Proceedings of the 27th Intern. conference on Software engineering*, pages 15–22, New York, NY, USA, 2005. ACM Press.

[10] J. Robinson, I. Wakeman, and D. Chalmers. Composing software services in the pervasive computing environment: Languages or APIs? *Journal of Pervasive and Mobile Computing*, April 2007.

[11] B. Schilit and M. Theimer. Disseminating active map information to mobile hosts. *IEEE Netw.*, 8(5), 1994.

[12] R. Want, K. P. Fishkin, A. Gujar, and B. L. Harrison. Bridging physical and virtual worlds with electronic tags. In *SIGCHI conference on Human factors in computing systems: the CHI is the limit*, pages 370–377, Pittsburgh, Pennsylvania, USA, 1999.

[13] T. Weis, M. Handte, M. Knoll, and C. Becker. Customizable pervasive applications. In *4th IEEE Intern. Conf. on Perv. Comp. and Communications*, pages 239–244. IEEE Comp. Soc., 2006.

[14] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, Sept. 1991.

[15] S. Zachariadis, C. Mascolo, and W. Emmerich. The SATIN component system - a meta model for engineering adaptable mobile systems. *IEEE Trans. on Softw. Eng.*, 32(11):910–927, Nov. 2006.