Self-Adaptive Software Systems

Doctoral Course
DEIB-Politecnico di Milano
December 2013

Instructor: Carlo Ghezzi

Collection of papers produced by participants
# Table of Contents

On the self adaptiveness in Wireless Sensor Networks ........................................ 2  
Mikhail Afanasov, Sonda Bousnina, Giuseppe Massari and Vincenzo Sciancalepore

Dynamic Update of Distributed Systems .......................................................... 6  
Mohammad Ghafari and Mohammad Mehdi Pourhashem Kallehbasti

Adaptation in the Context of Cloud: Models at Run-time and Model-Driven Adaptation . 10  
Giovanni Paolo Gibilisco, Claudio Menghi and Marco Miglierina

Self-Managing Software Architectures: A Survey .............................................. 14  
Thi Thao Nguyen Ho, Shima Zahmatkesh, Luca Florio and Adnan Shahzada

Achieving Adaptation via Suitable Constructs at the Programming Language Level...... 18  
Ajaz Hussain, Andrea Recchia and Anna Maria Gut

Rainbow: Architecture Based Self Adaption .................................................... 22  
Abdulrahman Kaitoua and Yuriy Vaskin

Dynamic software updating: a survey ............................................................... 26  
Massimo Quadrana and Francesco Venco

Impact of Compositional Reasoning on Incremental Verification .......................... 30  
Alessandro Maria Rizzi and Christos Tsigkanos
A. Architecture

There is a significant part of work devoted to a self-adaptivity on an architectural level. Research in this area is focused mainly on such properties of the network as: topology, positioning of nodes, fault-tolerance and scalability. Unlike the other levels, architectural level implies a set of communicating and collaborating nodes.

An architecture for building self-configurable systems: Subramanian et al. [3] focused on the self-organizing architecture for WSNs and proposed components necessary for building such an architecture. The latter implies following components:

- **Specialized sensors** for monitoring different physical entities.
- **Routing sensors** provide a data dissipation and fault tolerance of the network.
- **Aggregator nodes** combine routing and sensor functionality to provide a better network flexibility.
- **Sink nodes** have a high storage capacity, store and process received data.

All the components mentioned above are sufficient for building a wide range of applications, where infrastructure consists of addressing, routing, broadcasting and multicasting mechanisms.

Given the proposed architectural components, author also propose four steps, which should be performed by the network to self-organize:

- **Discovery phase.** Each node discovers its neighbors.
- **Organizational phase.** Nodes are organizing groups, allocate addresses, build routing table and construct broadcast tree and graph spanning all nodes.
- **Maintenance phase.** Each node keeps track of its energy, constantly updates routing table, broadcast trees and graphs, sends I am alive message and routing table to its neighbors.
- **Self-Reorganization phase.** If node detects the failure of its neighbor, it updates its routing table, or starts Discovery phase in case of the failure of all of the neighbors.

The analysis of the approach shows that the hierarchy of the network is strictly balanced; the complexity of the routing is \(O(\log n)\); the network is extremely tolerant to either node or link failures; the uniqueness property is guaranteed by the presence of a hierarchy; specialized sensors are allowed to be mobile. There are several weaknesses of the approach, though. Thereby, the approach is not optimized for extremely dynamic systems, when the network changes badly or very fast. On the other hand, the required protocol for such networks is not discussed. Despite the specialized sensors can be mobile, they can not move beyond the reach of routers. The latter, however are considered static.

The approach is very well fitted for static WSN, such as one for collecting meteorological data. But it is absolutely not applicable for systems with high dynamics such as wildlife monitoring [4], since the network changes badly and rapidly.

**ASCENT: Adaptive Self-Configuring Sensor Networks**

**Topologies:** In [5] the authors provide an other self-configuring approach in order to deploy micro-sensor for a wide range of environmental monitoring applications. They assumed a very-dense scenario where is extremely necessary to find a trade-off between the coverage and the interferences due to the large number of involved sensors. The real scenario given by that work is a habitat monitoring sensor network that is deployed in a remote forest, where the sensors are dropped by a plane. Different aspects are taken into account and then analysed: power consumption as well as the distributed sensing task. The work seems to be very interesting for two different reasons: first, adaptive techniques implied permit applications to configure the underlying topology based on their needs by trying to save energy and extend the network lifetime, second, the self-adaptive approach is based on the operating conditions measured locally. Specifically, the authors identify two kinds of sensors, namely nodes, in the network: i) active and ii) passive. The active nodes stay awake all the time and perform routing procedure, while the passive nodes listen the channel and periodically check if they should turn into active mode. The active nodes will be in charge of producing messages (sources) or just disseminating them (sink). In the case of low channel condition, the sink nodes could send an help message to other passive node in order to activate them. The self-configuring process starts by turning on randomly some nodes in the network. Such nodes enter firstly in a test mode, where they can exchange data and routing control messages, so that after a prefixed time can be switched to an active mode. If there are too many neighbor nodes active, according to a fixed parameter, the node will be switched off. Afterwards, if the number of active neighbor nodes will be less than a prefixed parameter, the node will be re-activated. Note that the performance of the system depend on the value of the parameters above, e.g., loss threshold and neighbor threshold.

The work provides a good validation for the parameters provided, by analyzing and comparing the network performance in terms of energy savings and network capacity. The gain of power saving is a factor of 3 better in some dense cases.

B. Application

From an application perspective the challenges introduced by WSN are basically related to their distributed architecture, the scariness of resources of the single sensor node, and the reliability issues due to sensor nodes disappearing for unpredictable faults, communication noises or battery discharging. This requires the development of ad-hoc programming models that could fit in the architectural view of the WSN, and that allow the application to react and adapt to environmental changes. This section introduces some approaches addressing this kind of problem.

developer can specify the way the contextual data will be collected, processed and used by the application. COPAL-ML implies such components as:

- **Context type** specifies the format of the data provided by a WSN.
- **Publisher** periodically publishes an information about a specific context type.
- **Listener** handles an event fired by the publisher and receives a context type as an argument.
- **Processor** handles the received data.

Along with possible scenario, authors also proposed several processing patterns, which can be used as solutions for effective self-adaptive application. Despite it is very promising way towards abstraction of WSN on the high level, the framework is Java-based. Thus, it is only applicable for sensors with Java machine built in. It is also can be used on a computer, but in this case the communication between the framework and the node remains unclear.

**Supporting Lightweight Adaptations in Context-aware Wireless Sensor Networks**: A common problem with WSN is to deploy an application, or update it, when the sensor network features a lot of nodes spread over a wide or region. As well, reprogramming a whole WSN deployed in an inaccessible region, represents a cumbersome, and often unfeasible, activity. Thus, the challenge is to support these operations in a feasible way, at the price of the lowest possible overhead.

Apart from this issues, a typical requirement of applications like **environmental monitoring** is to adapt the behavior of the nodes to the dynamics introduced by events, as for instance the presence or people moving around or not.

The work of Taherkordi et al.[7], introduces the WiSeKit distributed middleware, along with the ReWiSe software component model, as a solution to support a lightweight behavioral adaptation of sensor networks. The WiSeKit middleware exposes the following services

- **Local Reasoning** to update the values of components parameters based on a local adaptation policy.
- **Adaptation Proxy** to receive adaptation request from cluster head.
- **Component Repository** to temporarily store a new components image.
- **Component Reconfigurator** to load, reload or remove a running component.

The ReWiSe component model considers the application as an integration of separate components. A component can be dynamically replaced, Reconfiguration in WSN can be a very expensive operation, since it requires to transfer code to sensor nodes, to save and resume of state information, and eventually restart the sensor node. The main costs are paid in terms of **energy consumption** and are proportional to the size of the code images to transfer. For this reason, the goal of this work has been to develop a software component model that i) enables the possibility of a fine-grained reconfiguration of the application, ii) reduce the overhead due to state saving. The framework has been implemented on top of the Contiki operating system. Preliminary experiments reported a decrease of energy consumption of about 75%, compared to a common software component model.

**C. System**

The system level requires a lot of coordination between distinct sensors in the network. Particularly, several works leverage the WSN capabilities in order to reduce power consumption, make the MAC layer more flexible, and perform routing in a very efficient way. We take into consideration some of them in order to cover the main categories of system level improvements in the WSN.

**Proactive Reconfiguration of Wireless Sensor Networks**: Steine at al. in [8] leverage on the exploitation of a design-time exploration, to identify a set of operating modes. At run-time the WSN can adapt its behavior to environmental conditions and QoS fluctuations, by reconfiguring itself into the most suitable operating mode. The paper defines an **operating mode** as a set of values assigned to some controllable parameters of the network protocols. The nodes can dynamically change, at run-time, their operating mode according to specific observable events, that potentially affect the QoS of the WSN. The events are detected using sensors and/or the current time of the day. This figures out a proactive approach in the self-adaptive behavior. Concerning the network parameters, it is worth to distinguish between **local** and **global** parameters. In the former case, the single node can independently reconfigure itself without affecting the others. For instance, the tuning of the transmission power. In the latter, changing a global parameter necessarily requires a synchronization step, in order to maintain the proper functioning of the WSN. Example of global parameters are the TDMA slot-size and the sleep-time of the nodes. Thus, whenever a node detect an event, it can immediately adapt its local parameters or notify the other nodes about the need of change the current global mode. The authors have demonstrated the validity of the approach by testing it in a “cow-health” and a “office” monitoring scenarios. The evaluation metrics considered are the average delivery ratio and the average power consumption. In both the scenarios it has been experienced a significant reduction of power consumption, comparing the approach to a single configuration (worst-case) not adaptive design. This, keeping the packets delivery ratio close the values related to the worst-case based design.

**pTunes: Runtime Parameter Adaptation for Low-power MAC Protocols**: The work of Zimmerling et al. [9], is focused on an adaptation of system level – MAC protocol. The proposed framework – pTunes – allows to adjust the parameters of protocols to adapt to link, topology, and traffic dynamics. To this end, authors analyze the vital parameters – both protocol dependent and protocol independent – of the network, such as: reliability, latency and life-time. The framework also meets such requirements as: minimum disruption, timeliness, consistency and energy efficiency. Please note that pTunes deals exclusively with network protocol adaptation. Although the ASCENT[5] approach also touches network
and time. Therefore, self-adaptation for WSNs becomes an
system, since additional calculations consume more energy
the reduction of both service time and responsiveness of the
is not calculated or applied yet.
the old behaviour is no more suitable, but the new behaviour
states, where in the environmental conditions are changed and
of the environment. Indeed, there also could be transitional
variations, since the adaptation to unpredicted situations is
WSNs are usually built on very resource-restricted platforms,
possible environmental conditions the system can experience.
strictly bound to a single perspective, like shown in [5].
The development of such systems can be a long and
expensive process, since the developer should take care of all
possible environmental conditions the system can experience.
WSNs are usually built on very resource-restricted platforms,
which makes it necessary to predefine all possible behaviour
variations, since the adaptation to unpredicted situations is
very expensive in terms of energy and memory consumption.
Self-adaptation is also expensive in terms of performance,
since the system becomes less responsive during the analysis
of the environment. Indeed, there also could be transitional
states, where in the environmental conditions are changed and
the old behaviour is no more suitable, but the new behaviour
is not calculated or applied yet.
Thus, we can consider as costs of self-adaptivity for WSNs
the reduction of both service time and responsiveness of the
system, since additional calculations consume more energy
and time. Therefore, self-adaptation for WSNs becomes an
hard choice for such systems where dynamic changes do not
frequently occur.
On the other hand, self-adaptation schemes lead to relevant
advantages in terms of performance. The most important bene-
fit, targeted by the majority of the works, is the energy saving.
Specifically, a smart usage of resources, such as switching
off some nodes, reducing the transmitting power, and so on,
provides an useful maximization of the battery lifetime by
improving the energy consumption.
Moreover, the self-adaptation techniques help to perform a
more efficient coverage of the area in the Wireless Sensor Net-
works. Indeed, the self-adaptation provides better performance
by optimizing the placement of nodes. At the same time, this
leads to a more reliable system by assuring fault-tolerance.
Interestingly, the nodes which are not involved in the coverage
process, may be activated for supplying the sinks.
Finally, we can state that the most important feature in-
roduced with the self-adaptive wireless sensor networks is
the flexibility, as well as the ability to proficiently react to
the environment changes. In this sense, the self-adaptation
paradigm plays a key-role in terms of modifying system
behaviours to fulfill the network requirements, by monitoring
the current performance of the system.
IV. CONCLUSIONS
The presented paper shows some noticeable example of
an interesting approach, called self-configuration (or self-
adaptation) in the context of Wireless Sensor Networks
(WSNs). The survey collects different proposals for any of
the above-mentioned categories: the architecture layer, the
application layer and the system layer. The work sheds light
on what we consider a hot topic in the wireless sensor field,
providing for any category the most relevant approaches pro-
posed in the literature. Finally, advantages and drawback are
pointed out in order to evaluate the real impact of introducing
the self-adaptive in the common wireless sensors networks,
showing a substantially gain in terms of performance for a
very dynamic system while an inefficiency when the systems
do not face with recurring changes.
REFERENCES
through social community detection,” in Proc. of EWSN, 2010.
of context-aware applications in wireless sensor networks,” in Proc. of
SESENA, 2011.
[7] A. Taherkordi et al., “Supporting lightweight adaptations in context-
[9] M. Zimmerling et al., “ptunes: Runtime parameter adaptation for low-
power mac protocols,” in Proc. of IPSN, 2012.
Dear reviewers, dear editors,
We would like to thank you for your feedback on our submission, as well as for the insightful comments, which allowed us to further improve the quality of our paper. We have done our best to address all the raised concerns.

Review 1

Q: “I would suggest to reduce a bit the “State of the art” section (which takes about the 70% of the paper length) and, if possible, introduce a section with the description of at least one application among those described in the papers considered for the survey”.
A: We disagree on this point, since state of the art is the core of a survey. Regarding the application, since the approaches are application-specific it is very hard to provide a single example that would cover all the problems from all the perspectives.

Q: “In this context, … to an unpredicted environmental dynamics”: the meaning is understood but the sentence should be rephrased to improve clarity.
A: This sentence is rephrased in order to improve the clarity for the reader

Q: “Focusing on the one of the directions … use them all together”: can Architecture and System directions be considered somehow linked? See also my previous comment.
A: As we shown in the paper, architecture level deals with topology dynamics, while system level cope with internal parameter of the node in WSN. These two levels are orthogonal, so we can safely combine them in one effective solution.

Q: “There also could be an intermediate state, when the old behaviour is not applicable in the new situation, but the new behaviour is not chosen yet.”: could you please clarify what it is supposed to happen at system level in such particular condition? Are there strategies to avoid this?
A: The system simply misbehaves, and cannot be relied on. This is a critical problem, but to treat it is out of the scope the report.
However, the sentence has been rephrased as follows:
“There also could be a transitional state, where in the environmental conditions are changed and the old behaviour is no more suitable, but the new behaviour is not calculated or applied yet.”

All the typos reported have been fixed, and sentences have suitably rephrased.
Dynamic Update of Distributed Systems
Mohammad Ghafari, Mohammad Mehdi Pourhashem Kallehbasti
(ghafari|pourhashem.kallehbasti)@elet.polimi.it

Abstract—One of the significant challenges in runtime adaptation is to find best time which the adaptation of a given component results in neither inconsistency, nor significant disruption to the system. Kramer and Magee [1] developed a technique, known as quiescence, that forms a static component dependency model of the system calculates the components that have to be passivated before a component can be safely adapted. In this report we review some of related work addressing this challenge.

I. INTRODUCTION
In real world, some requirements may limit the applicability of offline updates. To provide a few preliminary examples: some software systems need a considerable start-up time to reach the point that they can provide their functionalities with desired non-functional requirements. Some other software systems cannot tolerate blackouts and impose irreversible losses to the environment they are embedded in or assets they are controlling. In this way, services provided by the system become temporarily unavailable for a period of time that is not acceptable in highly available software systems, such as business critical, mission-critical or safety-critical systems, because any disruption in their services may impose excessive financial or human costs.

Dynamic reconfiguration (interchangeably, adaptation) intends to preserve Quality of Service (QoS) of the system during change in terms of continuity of services and availability of unaffected parts of the system [2].

Fig. 1. Dynamic reconfiguration process.

The most important issues which must be addressed in dynamic reconfiguration are depicted in Figure 1. The first issue is reaching a safe application state in which it is possible to change target component(s) without jeopardizing ongoing activities and breaking the system consistency. This step should impose the least possible disruption on the running system. Once the component(s) has been placed in a safe state, it is the time to apply desired changes in a way it preserves expected constraints before and after evolution. Extracting the internal state of old components and initializing new ones regarding these states finalizes the reconfiguration process.

The focus of this report is mostly on the first step which is finding the best time which the adaptation of a given component results in neither inconsistency, nor significant disruption to the system.

In section 2, we discuss two most influential approaches in the area of the best time to perform dynamic adaptation. Section 3 reports dynamic evolution of distributed systems followed by an approach to build an adaptive application server in section 4. Ensuring reliability of changes and an approach to recover system during evolution failure is discussed in section 5 and 6 respectively. The paper is concluded in section 7.

II. DYNAMIC CHANGE MANAGEMENT

In the course of performing configuration changes application information should not get lost and application should be left in a consistent state. To this end, the management system should have an interface with the application which allows it to direct the application towards an appropriate state for reconfiguration. The interface must be generic that makes management independent of the particular application. To meet this objective, application state is abstracted into a set of configuration management states for each node. This set of states provides sufficient information about application state to allow the management system to perform changes which leave the application in a consistent state. The set of states is defined as follows [1]:

- **Active:** A node in the active state can initiate, accept, and service transactions.
- **Passive:** A node in the passive state must continue to accept and service transactions, but it is not currently engaged in a transaction that it initiated, and it will not initiate new transactions.

Kramer and Magee [1] identified a passive state as a necessary, but insufficient condition for updatability, as a node may still be processing transactions that were initiated by other nodes. Therefore, they introduced a stronger criterion, namely quiescence. Vandewoude et al. [3] proposed tranquility as an alternative for quiescence criterion. The remaining of this section covers definitions of quiescence and tranquility, and the reason the latter outperforms the former.

A. Quiescence
Kramer and Magee [1] developed a technique, known as quiescence, that forms a static component dependency model of the system calculates the components that have to be passivated before a component can be safely adapted. For sake
of consistency during change a property is required, which is, that the node is not within a transaction and will neither receive nor initiate any new transactions. This property is called quiescence of a node and is that state in which the node is both passive and has no outstanding transactions which it must accept and service. Such a state depends not only on the node itself, but on the connected nodes. A node is quiescent if:

1) it is not currently engaged in a transaction that it initiated,
2) it will not initiate new transactions
3) it is not currently engaged in servicing a transaction, and
4) no transactions have been or will be initiated by other nodes which require service from this node.

The application state of a node is both consistent and frozen, in the quiescent configuration state. It is consistent in that the application state does not contain the results of partially completed transactions, and is frozen in that the application state will not change as a result of new transactions. Quiescence is significant for dynamic configuration changes since, in cases such as unlinking, it permits a node to make decisions based on a stable and consistent state regarding the particular actions it should take before it is unlinked. For instance, the node may pass a consistent uptodate version of its application state to its environment before it is unlinked.

B. Tranquility

Vandewoude et al. [3] proposed tranquility as an alternative for quiescence criterion. Tranquility drastically reduces disruption to the running program while remaining a sufficient condition for consistency, whereas quiescence has the problem that it causes high disruption to the active program. This criterion is based on following observations:

1) There is no problem in replacing a node while a transaction is active as long as the node to be replaced is not involved in that transaction. This means that a node that has participated in an ongoing transaction may be replaced if it is certain that there will be no more future participation of that node in the transaction. It is equally permitted to replace a node that may, at some point in the future, participate in an ongoing transaction if it has not yet participated.

2) Using a black-box design for system nodes is a good approach for enhancing reusability and decoupling the system parts. This implies that the nodes may require services from other nodes they are connected to, but that they may never rely upon their implementation. If all nodes are a black box by design, then all participants in a transaction either are the initiator of the transaction or are directly connected (adjacent) to the initiator. Nodes that are indirectly connected to the initiator can, by definition, not participate in a transaction driven by the initiator since their existence is unknown to the initiator. Note that any participant in the transaction can, in turn, initiate new transactions in response to a message they process. These subtransactions, however, are part of the implementation of this participant and are not known to the original initiator.

These two observations are exploited by the concept of tranquility, which are introduced as an appropriate status for updatability:

Node is in a tranquil status if

1) it is not currently engaged in a transaction that it initiated,
2) it will not initiate new transactions,
3) it is not actively processing a request, and
4) none of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.

Quiescence is a stronger concept than tranquility in the sense that quiescence implies tranquility but not vice versa. Condition 3 of quiescence implies that the node is neither actively processing a request nor waiting for a new request in an already active transaction. This trivially implies Condition 3 of tranquility. Condition 4 of quiescence states that none of the adjacent nodes have initiated or will initiate a transaction in which N participates. Hence, no such transaction is active, trivially implying Condition 4 of tranquility. Tranquility does not imply quiescence, however, since it does not require that nodes connected with N may not initiate new transactions that involve N. For tranquility, nodes directly connected to N must not reach a passive status.

Tranquility has the distinct advantage that it is much less disruptive than quiescence since only the affected node N must be passivated. Although the third condition of tranquility requires some adjacent nodes to finish a certain transaction, these nodes need not be completely passivated.

III. Dynamic Evolution of Distributed Component-Based Systems

How to preserve consistency during evolution and how to reflect the abstract evolution specification in the concrete configuration implementation are some of challenges in dynamic evolution of distributed component-based systems (DCS). In order to investigate the problem, researchers proposed various architectural description languages (ADLs), enabling evolution techniques, etc. However, different ADLs usually address different concerns and the lack of runtime support for the causal relation between ADLs and the running system easily leads to the mismatch between them, thus inevitably sacrifices their usability.

Zhou [4] proposed an approach based on a runtime architecture which is visually generated from an attributed type graph meta-model, exists through the lifecycle of DCS, establishes the causal relation between architectural topology and system configuration, and directs the dynamic evolution. This research effort on architecture support for dynamic evolution falls in two categories, theoretical, and technical. The former exploits algebra notations or formal language to deduce the structure specification and impose the abstract constraints and evolution rules, while the latter investigates the enabling technology to explicitly instantiate and maintain the architectural model, or
automatic generation of architecture style applied code skeleton. The advantages of each category are the disadvantages of the other: the specification is sound and traceable but difficult to implement, while the manual, ad-hoc implementation and evolution lack the formal basis, thus easily become inconsistent during evolution. This was a motivation to combine the two categories to obtain the complementary advantages of both: using a formalism to derive the runtime architecture model, which is then instantiated and existing through the life cycle of the DCS. Following is 3-step approach.

1. Establish the mapping from an attributed type graph (ATG) meta-model to the runtime architecture model which is in the form of a class.
2. Runtime architecture instance generation, interaction with the DCS, and distribution to the involved containers.
3. Reconfiguration of the graph meta-model and the generation, interaction of evolved architecture model.

Dynamic reconfiguration of component-based distributed systems has been also addressed by Ma et al [5]. They show that dynamic reconfiguration of a system is correct if it happens at such a time instant that all its transactions, including those started before and ended after the update, are kept version consistent. This is because of the correctness of the old and new configurations and the fact that any version-consistent transaction is served—along with all its sub-transactions—as if it entirely completed within the old or the new configuration, no matter when the update actually happens. Also note that a transaction that ends before (starts after) the update cannot have a direct or indirect sub-transaction hosted by the new (old) version of a component being updated.

IV. AN APPLICATION SERVER TO SUPPORT ONLINE EVOLUTION

Authors in paper [6] propose Runtime Software Architecture (RSA) modification of the running application, which may accordingly change the functionality or quality of the application. To this end, they use PKUAS (Peking University Application Server), an architecture-based reflective component operating platform compliant with Java 2 Platform Enterprise Edition.

Component container is the main mechanism in an application server that supports component technology. Container gives component independent running environment, which makes development and management of components more feasible and convenient. We excerpt two important types of component evolution that are discussed in this paper:

1) Evolutions that do not alter requirements. Such as erasing bugs or improving an algorithm by updating component implementations.

2) Evolutions that alter function requirements. Such evolutions lead to adding and removing components, or altering components interface to change components functions.

The primary challenge is to update running components. The proposed solution in this paper is based on fine-grain component management. To support online application evolution, besides the usual component states such as loading and running, they further separate component state into three states:

1) State of ready, which means that the implementation of a component has been loaded, but no instance has been initialized.

2) State of active, which means some instances have been created and initialized from the component, but no instance is executing.

3) State of executing, which means some instance in this component is executing.

When an application is loaded, all of its components are in ready state firstly. When the component receives a creating request from client, the container creates an instance. Then this component enters active state. When a component is in ready state, updating becomes a little more complex. If the component receives an instance request, it will enter the executing state and generates a response to the client. Then it may return to the active state again.

Besides these three states, they also introduce another special state: evolving state, which means that the component is under online evolution. For each component to update, their states should firstly be changed evolving. If the component is in the evolving state, all incoming requests to it will be buffered in the container. For a component in the ready state, its implementation can be updated directly after setting its state to evolving. For a component in the active state, not only its implementation should be changed, but also its instances. For a component in executing state, they do not propose any approach to update the executing instance immediately, because it is hard to keep the system in a safe situation.

V. PARTIAL SCALABILITY TO ENSURE RELIABLE DYNAMIC RECONFIGURATION

Most current approaches for dynamic reconfiguration assume that the evolved system will behave as expected and thus will be reliable if the reconfiguration is consistent [8]. This assumption may not correspond to reality because the delivered quality estimated previously could vary due to parameter changes at runtime. To ensure that the system acts correctly in the field after the reconfiguration, reliability of changes has to be checked at runtime with minimum disruption.

The principle idea behind this work is to follow a parallel conversion approach in which the new component is privately deployed on idle resources, i.e., cloud in this work, and allow parallel operation of the old and new components within the system under evolution. Meanwhile, stakeholders are able to (re-)assess compatibility and conformance of the new component in the operational environment. Therefore, during evolution, the system handles a growing amount of work related to examination of new components (referring to scalability) while only those parts of the system affected by the change are temporarily scaled up (referring to partial). Finally, if the evaluation process has been passed successfully, the new component replaces the old one through a consistent reconfiguration approach like quiescence [1] or tranquility [3].
as discussed before. The reconfiguration steps in this approach are depicted in Figure 2:

Once a change request to replace a component is issued, it is first required to plan desired changes in the running system. In this step, named reconfiguration preparation, the new component, known as CUT (Component Under Test), is plugged into the existing configuration as a new component but with the same bindings as its old version. Next, the system goes into the evaluation step in which CUT is transparently examined under real operational conditions. During this period, the results and quality of CUT is monitored to decide whether the changes should be accepted or not. Finally, the old component is safely replaced with CUT, if the latter conforms to its role; otherwise, the changes will be rolled back, so the system goes back to its earlier configuration.

VI. ARCHITECTURAL RUNTIME CONFIGURATION MANAGEMENT

Enabling the runtime modification of adaptation policies can result in unpredictable self-adaptive systems for which reaching undesirable configurations during adaptation becomes a real possibility. Georgas et al. [9] improved dependability and overall system usefulness by maintaining a record of reconfigurations and providing support for architectural recovery operations. Their vision of the Architectural Runtime Configuration Management (ARCM) supports adaptation process dependability through: (1) the monitoring and recording of changes in architectural configurations during system runtime, (2) the presentation and graph-based visualization to architects of configuration changes over the entire system lifetime, and (3) the provision of recovery facilities to explicitly reverse harmful or undesirable configurations. Using these facilities, a system architect may easily access a history of self-adaptive behavior expressed in terms of overall system configurations reached after enacted adaptations, as well as manually modify architectural configurations through rollback and rollforward operations.

A key feature of ARCM is the system’s architecture-centric focus: the problem of enhancing the dependability of self-adaptive software is tackled solely through explicit architectural models. In a fully decoupled and modularly applied manner— independent of the adaptation management process and the self-adaptive application itself—ARCM promotes the visibility of self-adaptive behavior and provides facilities which can be used to recover from undesirable states. Proposed solution borrows techniques from the field of configuration management for capturing, storing, and manipulating multiple versions of an artifact. Specifically, each transactional adaptation is captured as a change, and changes are related to each other in a version graph. This change graph is based on software architectural configurations and expressed in xADL.

Recovery Operations: Some self-adaptive systems may only modify themselves toward specific configurations carefully and explicitly defined during design. However, allowing the rules by which the self-adaptive process is governed to change during run-time introduces the possibility of undesirable modifications, which indicates a need for facilities supporting recovery. It is important to note that the determination of whether a configuration is undesirable is architect or user-determined, at this point in ARCM’s development.

While both the maintenance of the ARCM version graph and the implementation of recovery operations are not coupled to a particular method or criteria for detecting the merit of architectural configurations, presented work requires human judgment and intervention for the determination of whether a configuration must be recovered. ARCM provides rollback and rollforward operations in order to address this recovery aspect of dependability.

VII. CONCLUSION

The main contribution of this paper is a report on the domain of the best time to perform dynamic evolution. Regarding the ultimate goal of dynamic evolution, improving reliability of the system, we also report related work to recover system if evolution fails. Despite all extensive researches in this area, there is a lack of trust in using merely software-based approaches and real world critical systems are mostly still relied on hardware-based approached.

It is worth to mention that due to space constraints, the intention of this work is just to provide a brief explanation about this area and interested readers are strongly required to study the referenced papers for more information.

REFERENCES


Adaptation in the Context of the Cloud: Models at Run-time and Model-Driven Adaptation

Giovanni Paolo Gibilisco, Claudio Menghi, Marco Miglierina
DEEPSE group - DEIB, Politecnico di Milano, Milano, Italy
{giovannipaolo.gibilisco, claudio.menghi, marco.miglierina}@polimi.it

Abstract—Self-adaptation in the context of the Cloud concerns the design of controllers that guarantee the required Quality of Service (QoS), while minimizing costs of allocated resources. During run-time, controllers have to provision resources according to the current status of the system (e.g., its workload), which is continuously monitored. These controllers are usually designed to perform adaptation in a model-driven fashion, that is keeping a model of the system alive and updated at run-time and continuously referring to it for selecting the most suitable adaptation action.

The problem of designing self-adaptive controllers has been widely studied in literature. Many control approaches have been proposed to fulfill the different requirements of Cloud environments. These approaches differ in many aspects such as the type of the model used to describe and control the system, the elements that are monitored at run-time, the set of control variables.

This work aims to (1) identify a possible classification of the various approaches presented in literature (2) categorize some representative works according to this classification, highlighting differences among them.

I. INTRODUCTION

The Cloud computing paradigm obtained a huge success in the last few years due to the simplicity offered to customers in acquiring computing resources for distributing their applications to the public, without huge upfront investments, rather paying only for what they actually use. This new appealing paradigm requires the design of controllers able to find the best trade off between cost minimization and QoS assurance.

These controllers, however, have to face one of the main drawbacks of the Cloud, that is, the lack of exclusive access to resources, which causes – often unpredictable – variations in performances.

This paper provides an overview of the most critical aspects to consider when building or evaluating an adaptive controller for the Cloud environment (Section II). This aspects includes generic elements related to the self adaptation policy and characteristics more specific to the Cloud. Section III presents some related work and describe them according to the proposed classification aspects. Section IV briefly summarizes the main contributions and concludes the papers.

II. CLASSIFICATION

The policies that have been proposed in literature to control Cloud computing platforms differ on many aspects, such as in their architecture, assurance, flexibility. This section aims to identify the main aspects that differentiate the various control approaches that have been presented in literature.

These aspects will be used as guidelines in our review. To this purpose two categories of aspects have been identified: adaptation and Cloud computing aspects. Adaptation aspects refers to features, such as flexibility and accuracy, that concerns the way in which adaptation actions are selected and performed. In the context of Cloud computing these aspects have been considered by Gambi et al., in [5]. Cloud computing aspects concern the set of assumptions on the behavior of the controller that strictly refer to the Cloud environment and how the controller interact with this environment. These aspects have been considered in different papers such the one by Calcavecchia et al., [3] that is used as a reference in this work.

A. Adaptation aspects

Gambi et al., [5] classify the controllers employed at run-time according to their flexibility and assurance. Flexibility refers to the “capability of the model to cope with changing and evolving configurations, and thus to the level of adaptability of the system”. Assurance is defined as “the possibility of measuring the predictability of the run-time behavior, and thus to the level of reliability of the system”. These two aspects are orthogonal, on one hand we want to ensure that the system is able to adapt when the conditions in which the system is working change (e.g., the workload changes), on the other hand reliability can be guaranteed only on the basis of some hypothesis on the system behavior.

Flexibility and assurance depends on the way in which the controllers are developed. Some approaches require the full definition and the complete tuning of the model of the system and the relative controller at design-time. In these cases adaptation concerns the ability of the system of modifying its behavior in response to changes in its workload profile. Usually, these approaches privilege reliability over flexibility, since the models can be statically analyzed at design-time, but the system is not able to manage unexpected changes that occurs at run-time. In contrast, other approaches autonomously define and tune the model of the system and the relative controller at run-time. These approaches privilege flexibility over assurance since the system is able to cope with each type of change that can occur at run-time, but there is no formal guarantee on its correct behavior. Finally, hybrid approaches combine design-time and run-time techniques to conjugate the flexibility of models updated at run-time with the assurance gained with the treatment of models at design-time. These approaches usually identify at design-time a set of aspects
(e.g., parameters) that can change at run-time and consider adaptation only regarding these aspects.

\textbf{B. Cloud computing aspects}

Calcavecchia et al., [3] classify several controllers proposed in literature considering several aspects strictly related to the Cloud computing area. The first distinction concerns the applicability of the control approaches in provider-centric and customer-centric. According to Sharma et al., [11] provider centric approaches aim to “maximize revenue while meeting an application’s SLA in the face of fluctuating workloads”, while customer centric try to “minimize the cost of renting servers while meeting the application SLA”. Note that these two goals are usually competitive.

Another aspect that discriminate the different controllers proposed in literature is the service model on which the controller is based. Three fundamental models are used in the Cloud computing area: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). In an IaaS model, Cloud providers offer the infrastructure (e.g., virtual machines) on which the customers run their applications. Customers are in charge of maintaining both the infrastructure (the operating systems of their virtual machines) and their application software. In a PaaS setting customers run their applications in a Cloud platform but do not need to manage the underlying hardware and software layers. Finally, in a SaaS context, Cloud providers install and execute software applications, while customers only need to access these software from Cloud clients.

One more aspect considered in [3] divides the approaches that are presented in literature according to the architecture of the system and its controller in exogenous and endogenous. An exogenous self-management architecture specifies that the management part is separated from the functional part of the application (i.e., there is a component in charge of managing the system). In the endogenous case, each component is in charge of managing both its functional and non-functional part. In this case, the adaptation results from the collaboration of the different system components. In turn, exogenous self-management architectures are classified in three main categories: 	extit{centralized management}, 	extit{hierarchical management} and 	extit{multi-agent} management. Endogenous self-management systems are by their nature either hierarchical or multi-agent.

Additionally, the resource provisioning mechanism (which have also been considered in [3]) refers to the way in which the resources are provided to the different applications. Three different provisioning mechanisms are identified: replication, resizing and migration. Replication concerns the allocation or the de-allocation of virtual resources (e.g., scale in/out of virtual machines). This mechanism is particularly suitable in an IaaS setting. Resizing regards the possibility to control the performance of the resources that are allocated to the applications (e.g., increasing the speed of a virtual CPU). Finally, Migration refers to the possibility of moving parts of the application (e.g., components) from one resource to another.

Last but not the least the \textit{autonomic mechanism}, that is considered both in [3] and in [5], describes the methods and the policies that are used to specify the autonomic behavior of the system. In other words, the autonomic mechanism refers to the type of controller that is employed at run-time and is strongly related to the model on which it is based, such as Markov Decision Process (MDP) and Queue Networks (QN).

\section*{III. RELATED WORK}

This section describes several works that have been proposed in the Cloud computing area to control the behavior of Cloud platforms and customer applications. The textual description provides an overview on the proposed approach and its main characteristics, while, due to space limitations, the full classification that contains all the aspects identified in Section II is presented only in Table I\footnote{Since several aspects that have been identified in Section II are not related with measurement scales, we select an appropriate set of fuzzy values. Furthermore, all the works presented in Table I rely on the IaaS model.}.\footnote{The first identifier (M) describes the time between arrivals to the queue, the second one (M) is the size of jobs and c is the number of servers at the node of the queue. The identifier M stands for Markov or memory-less, which means that the arrivals occur according to a Poisson process.}

\subsection*{A. Design-Time}

We first consider a set of bearing approaches that require the full definition and the complete tuning of the model of the system and the relative controller at design-time.

Bi et al., [2] present a dynamic technique to determine the number of virtual machines to employ at each tier of a multi-tier application based on a hybrid queue model. Queues are used to model the different ways in which the system fulfill the user requests. Note that in this context the term hybrid refers to the queue model employed to model the system and is not related with the term hybrid that have been used to classify adaptation aspects. The contributions of this paper include (1) the use of an hybrid model derived from the combination of a M/M/c model and multiple M/M/1 model to provision computing resources to virtualized applications. (2) the definition of a non linear, constrained, optimization problem based on the hybrid model previously described that computes the number of virtual machines to allocate in response with the current workload. The use of the performance models previously described (Queuing Networks) gives formal guarantees on the response time of the system. The approach relies on the assumption that the model of the system is fully defined at design-time, and the set of changes that can be managed only refer to the current workload conditions.

Huber et al., [6] use the Palladio Component Model at design-time to describe the set of services that the user want to support and the set of available virtual resources. The Palladio Component Model is then transformed into an equivalent Layered Queuing Network (LQN) to make performance estimations. At run time the workload conditions are continuously monitored and Layered Queuing Network is used to evaluate the impact of the different adaptation actions. The proposed adaptation behavior is based on a two phase heuristic.
algorithm. The push phase is activated when there is a change in the incoming workload or a SLA violation and uses both resizing and replication to fulfill the SLA. The pull phase is executed after changes in the workload or after the execution of a push phase to minimize the number of resources utilized.

Benanni et al., [1] tackle the provisioning by using separate performance models for Online Analytical Processes (OLAP) and Online Transactional Processes (OLTP). The use of multiple performance models allows the controller to optimize the solution of the problem according to different dimensions, in particular the utility function of OLAP processes aims at maximizing the throughput while the one of OLTP minimizes the response time. To assess the performance of a provisioning configuration according to OLAP processes a closed QN model is used and for OLTP an open QN is preferred. The controller is divided into a Global controller and some local controllers. Local controllers hold models for each application and are used to update the values of the utility functions for each of them. The global controller asks local controllers information about the current configuration and the utility functions and optimizes the new configuration.

B. Hybrid

Previously described works consider adaptation as a set of actions to be performed in response to changes in the current workload. However, in turn, Cloud computing systems themselves change over time and implicitly call for controllers capable of managing these changes. Hybrid approaches try to solve this problem by identifying at design-time a set of predefined aspects (e.g., parameters) that changes at run-time.

Calcavecchia et al., [3], present a probabilistic algorithm to permit the auto-scaling of services over multiple Cloud computing infrastructures. The idea is that each service is an autonomous entity that can decide whether create an other service, that is its replication, or remove itself from the running system to keep the utilization close to a given threshold. This makes the approach endogenous since the service is both responsible to process requests and to manage its non functional aspects. Services are represented as nodes of a graph which have only partial information about the global structure of the system, that is their neighbors and the relative loads. In other words, the model of the system is distributed at run-time over the different nodes of the network. The adaptation concerns the decision of adding or removing services in a probabilistic way. The decision is performed independently by each node of the network and it is related with the current load of its neighbors and two thresholds $L_{\text{min}}$ and $L_{\text{max}}$. If the load is less (higher) than $L_{\text{min}}$ ($L_{\text{max}}$) a new service is added (removed) with a probability that depends on the result of the computation of an algebraic formula. Note that this approach is classified as hybrid since the adaptation depends on a model of the system that is fixed but a set of parameters (e.g., the neighbors of a node) are monitored and updated online.

The approach presented by Patikirikorala et al., in [10] is not directly related with the Cloud computing area (even if it can be applied also in this context), but it is discussed here since it is a classical example of hybrid approach and it has also been considered in the Cloud computing review of Sharma et al., [11]. Patikirikorala et al., describe a control theoretic approach based on Multi-Model Switching and Tuning (MMST) adaptive control and allows developers to design self-managing control systems that guarantee a desired level of performance at run-time. The controller contains a set of (multiple) static models that describes the system behavior in different operating conditions. The adaptive control system dynamically reconfigure its behavior by selecting the most suitable model to be employed at run-time (switch). Furthermore, a set of additional models with a fixed structure and a set of parameters that are updated at online (tuning) is added to the suitable controllers to be selected at run-time. The approach is hybrid since the structure of the controller is fixed, but its parameters are estimated online to tune the controller and to select the most suitable model at run-time.

Migliorina et al., [9] propose a hybrid approach to deal with availability constraints and cost minimization in a multi-Cloud environment. An application is supposed to be deployed on multiple Clouds and two different levels of exogenous control are performed: one layer is responsible of balancing the workload among the different Clouds, the other is responsible of provisioning the required number of machines in each node. A control theoretical approach is used in both cases. Load balancing is set so to reduce the error between the actual and the expected availability, modeling the system as a DTMC, and

<table>
<thead>
<tr>
<th>Id</th>
<th>Adaptive Model</th>
<th>Flexibility</th>
<th>Assurance</th>
<th>Main Phase</th>
<th>Point of View</th>
<th>Adaptation Management</th>
<th>Structure</th>
<th>Provisioning Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>[2]</td>
<td>QN</td>
<td>Low</td>
<td>High</td>
<td>Design-time</td>
<td>Customer</td>
<td>Exogenous</td>
<td>Hierarchical</td>
<td>Replication</td>
</tr>
<tr>
<td>[1]</td>
<td>Open and Closed QN</td>
<td>Low</td>
<td>High</td>
<td>Design-time</td>
<td>Provider</td>
<td>Exogenous</td>
<td>Hierarchical</td>
<td>Replication</td>
</tr>
<tr>
<td>[10]</td>
<td>ARX</td>
<td>Medium</td>
<td>Medium</td>
<td>Hybrid</td>
<td>Both</td>
<td>Exogenous</td>
<td>Centralized</td>
<td>Session allocation</td>
</tr>
<tr>
<td>[9]</td>
<td>DTMC and Analytic Model</td>
<td>Medium</td>
<td>High</td>
<td>Hybrid</td>
<td>Customer</td>
<td>Exogenous</td>
<td>Hierarchical</td>
<td>Replication</td>
</tr>
<tr>
<td>[8]</td>
<td>Model free</td>
<td>High</td>
<td>Low</td>
<td>Run-time</td>
<td>Both</td>
<td>Exogenous</td>
<td>Multi-agent</td>
<td>Replication and Migration</td>
</tr>
<tr>
<td>[4]</td>
<td>Surrogate Model</td>
<td>Medium</td>
<td>Medium</td>
<td>Run-time</td>
<td>Customer</td>
<td>Exogenous</td>
<td>Centralized</td>
<td>Replication</td>
</tr>
</tbody>
</table>

TABLE I: CLASSIFICATION OF A SET OF BEARING APPROACHES
Models: QN (Queueing Network), LQN (Layered Queueing Network), DTMC (Discrete Time Markov Chain), ARX (AutoRegressive eXogeneous).


delivering as much traffic as possible to the more convenient Cloud. Machine provisioning is keeps the CPU usage within a minimum and a maximum threshold. The approach has formal guarantees that the convergence is obtained in a finite number of steps. However, the convergence rate is chosen at design-time under certain assumptions on the workload type. Fine tuning of this value may be manually performed in case of changes.

C. Run-Time

Finally, a set of works have been proposed in literature to manage each possible change that occurs at run-time. These approaches are able to autonomously define and tune the model of the system while the system is running. After any change the model of the system is updated and the relative controlled is identified.

Jiang et al., [7] divide the adaptation in an estimating stage and in a controlling stage. The estimating phase refers to the prediction of the capacity needed to support the incoming workload. The paper proposes an ensemble of different prediction algorithms. The final output of the predicting phase is obtained by using a weighted majority voting strategy on the results computed by the different predictors. The weight of each predictor is then updated according to the error of the prediction so that the system learns which predictor to use according to different working conditions. Authors look at the provisioning problem from the provider point of view by minimizing the costs due both to idle resources and penalties due to service level agreements (SLA) violations. This work is included in the category of run-time approaches since only the initial set of estimating methods is defined a priori while the structure of the internal model of each estimator and the way the different results are combined are updated at run-time.

Li et al., [8] propose a reinforcement learning approach to deal with both the provisioning problem, i.e., finding the minimum number of servers required to satisfy resource requirements for all the deployed applications, and the dynamic placement problem, i.e., distributing the applications among servers to satisfy their response time and availability requirements. In this approach they discretize usage and performance metrics in three different categories (NORMAL, WARNING and CRITICAL), and define seven possible actions that can be performed either to manage the servers instances or to manage application deployment. For every state-action pair a reward is given based on the impact of the action. By means of a widely used technique, the Q-Learning algorithm [13], servers distributively learn the best actions to perform in order to obtain the best reward in each combination of states. The evaluation shows that the approach is highly flexible and able to adapt the system in face of unpredicted workload patterns. The main advantages of this approach include not requiring the system model and the convergence to optimality theoretically proved. On the other hand, the algorithm cannot deal with continuous actions or states.

Gambi et al., [4] use Kriging models as means to design the behavior of Cloud controllers to solve the provisioning problem. These controllers are able to adapt to unpredicted changes or events at run-time. The authors claim that, in a highly dynamic context such as the Cloud, static approaches (e.g., rule-based) have limited applicability, whereas other black box approaches (e.g., reinforcement learning-based) become expensive both to update and to bring to a steady state. Consequently, the authors proposed a black-box approach based on surrogate models, that is Kriging models, which are able to be efficiently trained online and with sufficient accuracy even with small data-sets [12]. Cons of this approach is the inability of the controller to consider temporal aspects due to its nature and the fact that system performances are modeled only in terms of the incoming workload and the system configuration (number of virtual machines).

IV. Conclusion

This paper presents a classification of control approaches of Cloud based applications that has been derived from those presented in [5] and [3]. All the papers that have been reviewed tackle the problem of adapt a pool of resources to a variable workload fulfilling some service level agreement and minimizing a cost function. The main limitation identified during this review is the absence of a standard benchmark to compare and validate the different control approaches. Moreover, a standard (set of) formalism(s) to describe the Cloud environment is still missing.

References

Self-Managing Software Architectures: A Survey

Ho Thi Thao Nguyen, Shima Zahmatkesh, Luca Florio and Adnan Shahzada
Politecnico di Milano
Dipartimento di Elettronica Informazione e Bioingegneria
Piazza L. da Vinci, 32 - 20133 Milano, Italy
Email: {thithao.ho | shima.zahmatkesh | luca.florio | adnan.shahzada}@polimi.it

Abstract—The current generation of software systems is inherently complex due to the constantly changing user needs and availability of resources. This dynamicity calls for new and innovative software engineering approaches. Moreover, ubiquity of mobile and sensing devices, explosion of information and evolution from the conventional static software systems to large scale dynamic and complex systems have triggered the need for software systems to be more flexible, scalable, dependable, self-configurable and self-organizing by adapting to the changing operating conditions and context. In this paper, we analyze various existing architecture-based self adaptation approaches both in terms of the architectural styles they adopt and the problems they claim to address. Some adaptation properties for different self-managing software architectures are also discussed. We find most of the studies performed in this area to be focused on the design aspect of domain specific software systems and primarily targeted towards improving performance, flexibility and reliability of the system. Very few of the studies have actually provided the empirical or realistic scenario to validate their claims, instead, they use simplistic examples to support their hypothesis.

I. INTRODUCTION

Most of the software systems today are built based on common assumptions to have stable system requirements, static operating environment, control over various components, and separation between design, development and runtime [1]. Therefore, the focus of research has been on the development methodologies that facilitate cost-efficient (both in terms of time and money) and robust software systems. The advent of various ubiquitous mobile and sensing devices has led us towards a newer generation of systems constituted by components built by different vendors. These systems operate continuously in large-scale dynamic environments where resources vary frequently with time. For such systems, conventional software engineering methodologies fail due to the fact that exhaustive software testing and verification does not remain a possibility. Further, given the scale and dynamic nature of these systems, manual reconfiguration of the components is not longer an option. Therefore, we need new innovative methods and techniques for building and managing large complex software systems that will be able to adapt themselves to changing operational context and mobility of the components and users [2]. Systems should be self-managing to become more flexible, available, scalable and fault-tolerant to support the 24/7 support to the users without failing.

In literature, the terms managed subsystem and managing subsystem are used to denote the primary constituents of a self-adaptive software system situated in an environment [3]. The environment refers to the external factors that self-adaptive software system needs to interact with. Similarly, the effects of software system in the environment will also be observed and evaluated. Managing subsystem contains logic for adaptation. It monitors the environment to realize system objectives; whereas managed subsystem adapts to changes by following adaptation plan introduced by managing subsystem. This is the general architecture model for self-adaptive systems and others layers can be added to support the management if required. There have been many solutions proposed for self-managing software systems that include network-based, control-based, language-based and architecture-based approaches [3], [4].

II. ARCHITECTURE-BASED ADAPTATION APPROACHES

Architecture-based adaptation is mainly concerned with structural changes at the level of software components [4].

The rest of the paper is organized in the following manner: Section II describes some of the important existing architecture-based approaches. In section III, we analyze the different approaches according to some adaptation properties. Section IV sums up the general quantitative analysis put together by some of the most active researchers in Software Engineering. Section V concludes the discussion.
A component should have the ability to configure itself in a manner that enables it to interact with other components and contribute towards achieving the general goal of the system. Architecture-based adaptation offers various benefits [2]: (1) the underlying architectural concepts and principles are applicable to a wide range of application domains, that enable software architectures to be the general solution for building systems with different needs. (2) It can provide an appropriate level of abstraction to describe dynamic changes in a system by using components, bindings and composition, rather than at the algorithmic level. (3) Architectures generally support various kind of component compositions which are very useful for development of large-scale complex applications. (4) There are lot of architecture description languages and notations which include some support for dynamic architectures and for formal architecture-based analysis and reasoning. In this regard, applications from many domains have employed architecture-based approach for achieving self-adaptivity. According to [3], embedded system, e-commerce, information systems, robotics are the dominances.

There exist a number of research and engineering challenges in the domain of architecture-based self-adaptation such as handling a wide variety of systems with different architectural styles, properties, and modification mechanisms, and reducing costs in adding external control mechanisms. Some of the architecture-based adaptation approaches that we will describe in this paper are: Goal driven self-adaptation, Model based self-adaptation and Pattern based self-adaptation.

A. Goal driven Self-Adaptation

Kramer and Magee proposed an approach [2] for self-management at the architectural level in which components configure their interactions autonomously to be compatible with the overall goal of the system. A three-layer reference model is introduced that consists of component control, change management, and goal management (see Figure 1). The component control layer contains a set of interconnected components where each component implements a set of provided services and need some required services to be implemented by other components. In addition, a component has externally visible mode, which is an abstract view of the internal state of the component. Some component level operations are also provided such as creating or deleting components, binding or unbinding connections, and setting values for component mode. Change management layer is responsible for handling changes by receiving the reported changes in states from the lower level or changes in goals at higher level. This layer contains a set of precompiled set of plans and strategies for predicted class of changes. The goal management layer process the state of the system and the goals specification and try to generate a plan to achieve that goal. The reference model is used to provide a various research challenges in different layers of the proposed self-management architecture. At the component layer, the most important challenge is the provision of change management capable of component reconfiguration to avoid the undesired transient behavior. Decentralized configuration management that provides the resilience in case of inconsistent system state is an important challenge to cater at change management layer. Similarly, the goal management layer requires constraint based planning to translate goals into plans. These challenges exhibit a certain level of difficulty and complexity to be overcome in order to apply this reference model to realistic scenarios.

B. External Model based Self-Adaptation

Rainbow framework [6] provides reusable infrastructure with some mechanisms for customizing it according to the needs of specific target systems. The framework is constituted of adaptation infrastructure and system-specific adaptation knowledge. The adaptation infrastructure provides some common adaptation functionalities that are reusable across all target systems whereas the adaptation knowledge is always specific to target systems. Figure 2 shows the adaptation cycle used by the Rainbow framework. Probes and Gauges observe the running system and report the observation to model manager that updates the architectural model. Rainbow uses the architectural style building blocks to create an architectural model by abstracting the behavior of target system. The provided building blocks are component and connector type, constraints, properties, analysis, adaptation operators, and adaptation strategies. The architecture evaluator checks for any violation of constraints upon receiving the updates from the target system. If the evaluator finds any violated constraint, it signals the Adaptation Manager that in turn selects an appropriate adaptations strategy using utility theory to address the problem. The Strategy Executor executes the strategy on the running system and the action is accomplished via system-level effectors. Regarding to the drawback, Rainbow architecture may suffer from single-point failure due to central management in architecture layer.

C. Pattern based Self-Adaptation

Jose Luis et. al. propose an approach that suggests architectural styles based on some bio-inspired patterns [7]. The Patterns are used as the building blocks that can be composed further to form more complex patterns [8]. Patterns can operate
as modular primitives to produce predictable effects while applied to various situations within self-adaptive software systems. Figure 3(a) shows the different design patterns suggested by the authors.

The architecture follows a Layered organization (See Figure 3(b)). At the bottom layer we find the Computational Environment composed of Core Services (low-level self-organizing mechanisms based on basic patterns and triggered by the environment), and the Blackboard (shared repository of data that can trigger the execution of Core Services). In the higher layers there are Agents (computational component which runs some self-organization algorithm as its own individual behavior) and Services (composed self-organization mechanisms based on complex patterns that may need some additional logic to be performed). Core Services are invoked by the Blackboard through Implicit Invocation according to the data injected by the Agents or high-level Services. The proposed Blackboard architecture is distributed that allows the coordination between different Agents belonging to different applications. The main disadvantage of this architecture is that Agents and Services cannot rely on a pre-defined Core Services flow of control.

### III. COMPARATIVE ANALYSIS

While the aim of self-adaptation systems is to react to changes to maintain system’s high level goals, it’s also important that adaptation mechanism needs to maintain non-functional qualities. Despite the different properties of presented architectures, there are some important non-functional properties [1] that are provided by the architectures although in different ways: reusability, fault tolerance/robustness, scalability.

#### A. Non-Functional Requirements

In self-managing system, reusability is the ability to use existing resources to build system that can satisfy predefined goals with capability of adapting to changes. Goal-driven approach exhibits reusability through translation from system high level goals to the executable plans affecting to behaviors of system components. In other words, this mechanism makes use of different existing components to fulfill functional requirements through executable plan, while still maintaining the abstraction layer between systems goals and concrete execution to adapt changes. On the other hand, model-based approach, with the example of Rainbow architecture displays the reusability through adaptation infrastructure and reusable architectural style components. Reusability in pattern-based approach is disclosed through low level self-organizing mechanism and implicit invocation.

Fault tolerance and robustness is a critical requirement for self-adaptation system. In fact, some researchers consider this quality as motivation for self-adaption because unpredicted changes in the system are the potential source of error. Many self-adaptation architecture-based approaches guarantee their fault tolerance capability through the way they manage and react to changes. In goal-driven approach, fault tolerance and robustness is ensured through the mechanism of change management; model-based approach exhibits its fault tolerance capability through adaptation engine; pattern-based approach provides robustness through service publication on Blackboard of different nodes.

Scalability in self-managing system can be classified as functional and non-functional scalability. Goal-driven approach can provide both functional and non-functional scalability through aggregation and composition of components; while functional and non-functional scalability of pattern-base is ensured through modularity of Core services and complex services. In the case of Rainbow architecture, only non-functional scalability is ensured by adaptation engine and model manager.

#### B. Self-Adaptation Properties

In order to evaluate and compare different architecture-based approaches, we need a common set of vocabulary to specify self-adaptation systems properties. In this section we discuss several modeling dimensions of self-adaptation system w.r.t three presented architecture-based approaches.

1) Type of Change: changes are the cause and motivation for system to be self-adapt. The nature of change can be classified into functional, non-functional and technological. Functional change refers to functional requirement system aimed to satisfy; non-functional change relates to QoS such as system performance, reliability, etc.; technological change refers to software and hardware aspects responsible for delivery of services. The ability of system in response to one or more types of change can be used to evaluate adaptability power of system.

2) Adaptation type: consists of parametric, structural or a combination of these. Parametric adaptation occurs when there is change in components parameters, structural adaptation occurs when change is happened in system structure.

3) Autonomy level: identifies the autonomous degree of adaptation which is ranged from autonomous to assisted. The former indicates the occurrence of adaptation without external influence, the latter indicates adaptation with some external assistance.

4) Adaptation organization: adaptation could be centralized, i.e. a single element is responsible for adaptation, or distributed, i.e. no single element has complete control over the system but the control is distributed among various components.

5) Adaptation scope: can be local or global, indicates whether the adaptation is localized or involves the entire system.

In table I, we have summarized the adaptation properties for the presented approaches.
TABLE I: Properties comparison

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Goal Driven</th>
<th>Model-Based</th>
<th>Pattern-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change type</td>
<td>Functional</td>
<td>Non</td>
<td>Functional</td>
</tr>
<tr>
<td>Adaptation type</td>
<td>Parametrical and Structural</td>
<td>Structural</td>
<td>Structural</td>
</tr>
<tr>
<td>Autonomy level</td>
<td>Can be assisted</td>
<td>Autonomous</td>
<td>Autonomous</td>
</tr>
<tr>
<td>Adaptation organization</td>
<td>Centralized</td>
<td>Centralized</td>
<td>Distributed</td>
</tr>
<tr>
<td>Adaptation scope</td>
<td>Local, Global</td>
<td>Global</td>
<td>Global</td>
</tr>
</tbody>
</table>

IV. RESEARCH FOCUS

A systematic study is required to evaluate the current status of the research and to have a better understanding of the research direction in this field. A current survey [3] has performed a systematic literature study covering 20 leading software engineering conferences and journals in the field to provide an up-to-date review of the field. This section will summarize the most interesting results.

A. What is the focus of research in architecture-based self-adaptation?

The study has showed that research in architecture-based self-adaptation has been driven by subject of the studies, feedback loop architecture and application domain. The 48% of the studies in self-adaptive software architectures focus on software design, while other subjects such as software quality (17%), requirements (8%) and testing (8%) are less popular. However the studies on requirements are all from 2006 onward, pointing out the increasing importance of this topic in software engineering. In term of feedback loop architecture, the dominant focus of studies is on single control loop (about 80%) using the MAPE loop as a reference model [9]. There is a growing interest in systems with multiple control loops, with the 92% of studies published in the last four years. Only 69% of the studies do consider and explicit application domain, while the remaining studies refer to abstract applications.

B. What are the claims made for self-adaptation and what are the trade-offs implied by self-adaptation?

The top three concerns related to self-adaptation are efficiency/performance (55%), reliability (41%) and flexibility (28%). Very few studies (less than 6%) address other aspects such as security, maintainability, etc. The majority of the studies consider only a single concern (57%) and only 3% consider more than 2 concerns, proving the narrow view of most researchers on engineering self-adaptive systems. Another noticeable trend is that most of the studies focus on the concerns with positive effect (91%) rather than on concerns with negative effect like the trade-offs implied by self-adaptation (less than 20%).

C. How much evidence is available for the claims and what are the types of evidence?

The research results on architecture-based self-adaptation have almost no practical application: out of 121 studies, only two studies were performed in a joint effort between academic and industry and only the 1.7% use as assessment method experience on real-world examples. Example application accounts for 67.8% of the studies, simulation for 19.8%, rigorous analysis for 8.3% and empirical study for 2.5%. As a consequence mostly of the studies provide minimal evidence from demonstrations or simple/toy examples (95.8%), evidence from expert opinions or observation account for 1.7%, and 2.5% provide (weak) empirical evidence.

V. CONCLUSION

Architecture-based self-adaptation has been employed widely due to its key properties like abstraction layer, scalability and generality that make it a suitable candidate for the design of self-adaptive systems. In this paper, we have presented the current status of research in architecture-based self-adaptive systems. We have studied and analyzed some of the important papers and discussed their features and limitations. We have highlighted the different approaches proposed by the researchers and analyzed them according to various adaptation properties. Most researchers focus only on the improvement provided by their self-adaptive architecture to qualities of the software, such as efficiency/performance, reliability and flexibility without taking into account trade-offs, neither with respect to other qualities nor the effects on concerns such as effort and cost. The lack of empirical studies and industrial application of architecture-based self-adaptive systems, with most research assessed using simple example applications with minimal level of evidence, point out that research in architecture-based self-adaptation is still more exploratory than exploitative.

REFERENCES


Achieving Adaptation via Suitable Constructs at the Programming Language Level

Ajaz Hussain*, Andrea Recchia*, and Anna Maria Gut*
*Politecnico di Milano,
Dipartimento di Elettronica, Informazione e Bioingegneria,
Via Ponzio 34/5, I - 20133 Milano, Italy
Email: {hussain,recchia}@elet.polimi.it, annamaria.gut@mail.polimi.it

Abstract—Implementation of context-aware systems can be supported through the adoption of techniques at the architectural level such as middle-wares or component-oriented architectures. It can also be supported by suitable constructs at the programming language level. Context-oriented programming (COP) is a novel paradigm tackling the issue of developing context-aware systems at the language-level, introducing ad-hoc language abstractions to manage adaptations modularization and their dynamic activation. In this paper we review the state of the art in the field of COP in the perspective of the benefits that this technique can provide to software engineers in the design and implementation of context-aware applications.

I. INTRODUCTION

Ubiquitous or pervasive computing is an advanced concept where computing is made to appear everywhere and anywhere, in contrast to classical desktop computing. It originates from the spreading of mobile devices, especially with users of smartphones. To make ubiquitous computing possible devices need to be able to smoothly adapt to the computing context. Context awareness refers to the idea that devices can both sense, and react based on their environment. The notion of context traditionally adopted in COP is open and pragmatic – any computationally accessible information can be considered as context [1].

The paper is organized as follows. Section II discusses preliminary concepts of the COP. Section III provides models and features of COP. Section IV presents emerging COP application areas. Section V presents the literature review. Section VI presents discussions and COP related research roadmap. Conclusions are drawn in Section VII.

II. CONTEXT ORIENTED PROGRAMMING MAIN CONCEPTS

In this section we present the foundations of the COP paradigm. We adopt a top-down approach, which starts from the abstract notion of context and then focuses on behavioral variations that conceptually enable context-aware adaptation.

A. Context definition

The initial idea of COP was that mobile devices could provide different services in different contexts, where context was strongly related to the location of a device. Much of the initial research of context-aware computing hence focused on location-aware systems. In this sense, the widely-used satellite navigation systems in cars today are context-aware systems. With the prosecution of the research the concept of context broadened, not limiting the context to the information reaching the system from outside (i.e. the environment), but also encompassing information originating inside the system boundaries, such as performance monitoring or intrusion detection. Such a general approach does not prescribe any restriction to the level of abstraction through which the context is represented inside the system. In this sense an application can exist in more than one context, so that different parts of the application, for example different threads, can adapt their behavior differently. Uniqueness of context is still under debate, while from a conceptual point of view a unique context leads to a more elegant and intuitive model, the possibility of exploiting multiple contexts in the same application is more flexible in practice.

B. Behavioral variations and layers

A context-dependent application varies its behavior according to conditions arising during execution. Three kinds of behavior variations can be identified [1]

- Actor-dependent Behavior Variations: An example of actor-dependent behavior variation is that of multiple visualizations of the same system or system entity, such as the rendering of statistical data as in different kinds of charts. Here, an actor determines which information provided by, or obtainable from, the system will be shown and in what form, contributing to the context-dependent behavior of the system. A system needs to behave differently not only in response to different requests, but often also in response to the same request by different actors.

- Environment-dependent Behavior Variations: An environment-dependent behavior variation is essentially any conditional guarding a subset of application behavior or execution. Examples are anything that is not given implicitly by the flow of control in the execution of a piece of code but needs to be checked explicitly, such as the object a variable is referring to, the time of day, the battery status of the current device, or the temperature read out of sensory equipment. Often, related code is scattered over several system parts to coordinate activation or blocking of related environment-dependent behavior variations. A system’s response to a stimulus initiated by an actor may need to be adjusted to take properties of the computational environment into account. Here,
the system’s behavior can be affected by anything adjunct to the immediate interaction between actor and system.

- System-dependent Behavior Variations: An example of system-dependent behavior variation is that of change notification, the dissemination of information about what system parts have changed, and how. The way change notifications can be observed should depend on the relationships between the various system parts involved or affected. Often, multiple and redundant notifications are generated for a single change, resulting in cascading updates or other undesirable effects. A system may need to vary its behavior depending on its own current state, historical information, or dependencies to other system parts or subsystems. In such case, the context that influences behavioral variation is not determined by different actors, but by the system itself.

COP makes available means to specify behavioral variations, typically consisting of new or modified behavior, but may also comprise removed behavior. They can be expressed as partial definitions of modules in the underlying programming model such as procedures or classes, with complete definitions representing just a special case. Variations may also be expressed as edits, wrappers, or even general refactoring or transformations. Layers are the linguistic mechanism that enables the programmer to group variations. Therefore, the actual behavior of a COP program is carried out by a dispatching procedure that selects the program fragments to be executed depending on the context contents. Layers are composed in reaction to contextual information. Based on information available in the current execution context, specific layers may be activated or deactivated. COP languages and environment extensions provide the mechanisms for expressing, activating and composing layers at runtime, but it is the application domain that determines which contextual information is relevant.

### III. CONTEXT ORIENTED PROGRAMMING MODELS AND FEATURES

In this section we present COP in more detail, discussing an overview of the features of the available implementations and the variations to the basic model described so far. At first, the implementation techniques adopted for COP languages, which have a significant impact on their usage, are being discussed. Then we consider the dynamic aspects of COP, i.e., how behavioral variations are activated. We analyze the static aspect of COP, which is about the way behavioral variations are modularized in the codebase. Finally we give an overview of how COP deals with the problems of initialization and consistency between dynamically activated adaptations.

#### A. Implementation approaches

COP languages have been implemented as extensions to existing idioms by adding the context-adaptation features. The implementation strategies strongly depend on the underlying language. In-language approaches are based on the use of libraries, which make COP features immediately available to the programmer [2]. The main advantage of this approach is that libraries seamlessly integrate with the existing language.

<table>
<thead>
<tr>
<th>Activation</th>
<th>Features</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically scoped</td>
<td>Adaptation propagates with the flow of execution</td>
<td>When all the entities in the control-flow are context dependent</td>
</tr>
<tr>
<td>Indefinite</td>
<td>The effects of a layer configuration extends indefinitely</td>
<td>When behavioral variations heavily crosscut the application</td>
</tr>
<tr>
<td>Implicit</td>
<td>Before calling a layered method all the layers in the application are checked for being active</td>
<td>Interesting from a modularization standpoint but limited</td>
</tr>
<tr>
<td>Per-object</td>
<td>Layer activation is controlled on single objects, and do not propagate along the control flow</td>
<td>When behavioral variations do not crosscut the application</td>
</tr>
<tr>
<td>Message-driven per-agent</td>
<td>Context related messages enable a behavioral variation on the receiver of the message</td>
<td>Highly concurrent systems</td>
</tr>
</tbody>
</table>

This makes the introduction of COP in an established development system easier. In many COP library-based extensions, metaprogramming, a technique that allows one to programatically inspect the entities that constitute a program, and possibly modify their behavior, has a central role. Through metaprogramming [3], libraries can change the pre-defined semantics of a language, introducing COP behaviors such as layer-aware method dispatching. Opposed to the in-language approach is the usage of source-to-source compilers exploited to implement several COP extensions.

Opposed to the in-language approach is the usage of source-to-source compilers exploited to implement several COP extensions [4]. Source-to-source compilers map the contextual code to standard code in the original language, rearranging the source structure in order to implement the context-adaptation features. Compiler-based implementations can achieve better performances but compatibility with existing tools supporting the development process is almost certainly broken, since the compilation process not only adds new keywords, but also disrupts the original source structure.

#### B. Activation mechanism

Behavioral variation activation as originally proposed in COP is performed through the ‘with’ statement and is dynamically scoped. Over the years researchers have recognized the need for activation mechanisms other than with-triggered dynamic scope. The choice of which adaptation mechanism to use depends on the design of the application. For example, if the adaptation concerns only specific entities that require local modifications, per-object activation can be the best solution. Otherwise, if the adaptation must be performed on all the objects in control flow, dynamically scoped mechanisms allow the activation to be locally controlled and let the behavioral adaptation propagate along the flow of execution. The freedom of choosing the activation mechanism that best fits the programmer’s needs is limited by the fact that not all the solutions are available in each language. Behavioral variation can be activated in different ways according to the design of the application. Table I describes activation mechanisms for behavioral variation. A recent research effort considered the option of providing an open implementation the programmer can rely on to develop the activation mechanism which best fits his needs [5].

#### TABLE I. ACTIVATION MECHANISM OF BEHAVIOURAL VARIATIONS.

<table>
<thead>
<tr>
<th>Activation</th>
<th>Features</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamically scoped</td>
<td>Adaptation propagates with the flow of execution</td>
<td>When all the entities in the control-flow are context dependent</td>
</tr>
<tr>
<td>Indefinite</td>
<td>The effects of a layer configuration extends indefinitely</td>
<td>When behavioral variations heavily crosscut the application</td>
</tr>
<tr>
<td>Implicit</td>
<td>Before calling a layered method all the layers in the application are checked for being active</td>
<td>Interesting from a modularization standpoint but limited</td>
</tr>
<tr>
<td>Per-object</td>
<td>Layer activation is controlled on single objects, and do not propagate along the control flow</td>
<td>When behavioral variations do not crosscut the application</td>
</tr>
<tr>
<td>Message-driven per-agent</td>
<td>Context related messages enable a behavioral variation on the receiver of the message</td>
<td>Highly concurrent systems</td>
</tr>
</tbody>
</table>
C. Variation modularization

Besides dynamic variation activation, COP directly addresses the problem of modularizing context-related adaptations. Since COP support must integrate with the existing language, the modularization technique strongly depends on the underlying idiom.

In the class-in-layer model, a layer definition is spread among many modularization units. This model is adopted in languages which already enforce a main modularization direction. The introduction of layers must not break this design. In the so-called layer-in-class model, layer declarations are outside the lexical scope of the code unit they alter. This model is usually adopted in languages which does not strictly enforce a modularization policy based on language constructs [6].

From a software engineering perspective, the main advantage of the class-in-layer approach is that it facilitates software evolution. In fact, adding new behavioral variations only affects those modules defining layers and modifying the already existing modules is not required. On the other hand, with the layer-in-class strategy, variations are declared together with the basic behavior they alter, simplifying the comprehension of the program. Moreover, all the behavioral variations are included in the entity they augment. The resulting code is self-contained and each class encapsulates all its possible behaviors. The layer-in-class solution is more effective in those cases in which it is not possible to recognize a basic behavior and a variation, but the system switches between two or more alternative behaviors.

D. Open points in COP

In a context-aware application, adaptation can be traced back to two aspects: adaptation of computation, and adaptation of state. Adaptation of computation deals with selection of alternative methods or functions. Adaptation of state deals with the choice among different state values. COP main focus is centered on adaptation of computation (i.e. context aware method dispatching) rather than state.

The practical implementation of context-aware software raises some problems. When a behavioral change is triggered in the system, it is often needed to perform an initialization of the state before starting the adapted execution. Safe initialization has been long recognized as a problem by COP researchers. COP languages were augmented with constructs that allow arbitrary code to be executed when a behavioral variation is activated or deactivated.

For example, EventCJ introduced the activate and the deactivate keywords, which inside a layer definition within a class declare code blocks that must be executed when the layer is activated or deactivated. In the pedestrian navigation system (Section IV) mobile application, which can use GPS or WiFi connection to get the device location, this mechanism is used to initialize and shutdown the GPS receiver.

In COP applications, behavioral variations are activated at runtime depending on the contextual conditions. When software grows and becomes complex, the resulting system behavior can evolve in a manner that is not easy to foresee. Since active variations dynamically combine with one another and with the basic application logic, inconsistencies and conflicts can arise. The COP research community has given several contributions to tackle these issues, devising solutions that can achieve better control over variation activation and enforcing constraints among variations. These approaches encompass reflection, declaration of constraints in the language or through domain specific languages (DSL), and encapsulation of variations management into abstract data types.

Behavioral consistency is strictly related to variations activation. Activation policies constitute per se a form of consistency enforcement among layers. For example, dynamically scoped activation enforces a strict discipline: the layers enabled in the with statement are consistently kept active for the whole dynamic scope and automatically removed when the scope expires. It is not surprising that more fine-grained activation approaches, such as per object layer activation, required to be regulated by an additional constructs specifying layer transitions.

IV. EMERGING COP APPLICATION AREAS

Three examples of COP applications are presented hereafter, one for PDA devices (Pedestrian navigation system using EventCJ language), second desktop software (CJEdit, JCop and EventCJ languages) and third is server-side software (ContextChat, ContextErlang language).

The pedestrian navigation system is a mobile application which main aim is to display a map depending on a position of device [7]. It uses features of EventCJ such as event declaration and layer transition rules. The system obtains the position based on either GPS coordinates, for outdoors, or wireless LAN based positioning system for inside building locations. According to that fact there are two layers distinguished: GPSNavi and WiFiNavi. However, only one can be active at the same time. Both implement the run method which is responsible for updating the screen. The former is scrolling the map to the most recent position whereas the latter displays floor plan (downloaded from a local database). Switching between these two layers is based on events. The run method is called when a change of the position is recognized. The pedestrian navigation system supports also on board situation. Whether boarding both GPSNavi and WiFiNavi layers are disabled, whereas arriving one of them is activated.

CJEdit is a programming environment supporting rich text formatting features [4]. The aim was to avoid separation of source code and documentation. CJEdit switches between two modes: text editing and programming. Both require different functionality which is provided by GUI elements. Context change is triggered directly by a user or automatically, depending on the text cursor position. While editing the source code, GUI elements remain in programming mode, whereas writing a new text leads to rich text formatting mode. CJEdit encapsulates GUI elements into layers and uses behavioral variations depending on cursor position or an explicit user’s action.

ContextChat [8], is a prototype of adaptable chat server. Inside the system users are represented as context-aware agents. One of two variations can be activated: online (when the client is connected) or offline (if there is no connection). Being online means forwarding messages to the user, whereas active
offline variation causes storing messages. Clients are allowed to activate an optional backup in order to save messages on a remote server. Moreover, the system offers tracking functionality to gather data on users' traffic. This information allows self-adaptive behavior, moving clients, who exchange messages frequently, to the same node, thus reduction of cross-node communication could be archived.

V. RELATED APPROACHES

Literature upon implementing context-aware software using design pattern is vast [9], [10]. Problems like turning existing legacy software into a context-aware application, monitoring the current context of the system, or performing the adaptation according to a set of predefined rules, has been catered by state-of-the-art approaches. Aspect Oriented Programming (AOP) [11] focuses on modularization and shares some common features with COP whose focus is on runtime activation of variations. Moreover, literature provides support to the dynamic AOP frameworks. In contrast, COP offers directly the required facilities. The dynamic features provided by dynamic AOP are not widely spread and implemented in the field of autonomic computing mainly [12]. AOP was used in conjunction with computational reflection using TRAP/J toolkit, discussed in [13]. These works show that computational reflection is used to support runtime change, while AOP allows separation of adaptation concerns.

Feature-oriented programming (FOP) is about synthesizing programs in software product lines [14]. Both COP and FOP supports variations of original program with language-level techniques. Other approaches similar to COP are DSPL and Subjective dispatch, as discussed in [15], [16]. The latter focuses on Object-Oriented languages. Dynamic adaptation from architectural viewpoint have been explored to find novel approach to support self-adaptation and solution for the runtime problem, as discussed in C2 [17] and Rainbow Framework [18].

VI. DISCUSSION AND RESEARCH ROADMAP

The literature on COP related approaches is vast, but still the implementation aspect is limited. There is an essential need to further explore the implementation dimensions of COP in software engineering paradigm. A formal specification of the behavioral change introduced by each variation is required to effectively plan the activation in a fully automated manner. Literature on specification related to COP is very limited, and have been an open challenge so far.

Although, literature provides support in representation of COP abstractions using UML, but still there is a lack of suitable formal constructs to represent COP abstractions using UML. Hence, a formal specification in UML shared notation is needed. Kamina et al. [7], proposed first attempt to verify COP based implemented system using SPIN model checker, but with few limitations. Additionally the area of testing for COP based applications is unexplored.

VII. CONCLUSION

This paper presents an overview of COP techniques in context-aware systems perspectives. The research on COP languages is promising in supporting dynamic adaptation through proper language-level abstractions. Thus, the development process could be improved by just using those abstractions by programmers. However, there is still need to adopt COP languages to a large size project in order to validate existing approaches and determine further steps.

REFERENCES

Dear Editor-in-Chief,

We are highly grateful to the editor and all referees for their valuable feedback on the submitted manuscript and for their helpful suggestions for our revisions. Below is a note summarizing the main changes to the submitted manuscript.

We have completely re-set the outline of the paper, by providing a clearer research setting and by better defining the aims of the proposed approach in order to address the comments of all reviewers. We hope that the revisions made address the recommendations of the editors and all referees and that the additional effort has significantly improved the quality of the manuscript.

ANSWERS TO REVIEWER # 1

Reviewer comment: “---The main concern with this paper is the lack of a critic contribute regarding the presented papers. Authors describe in details the work of Hirschfeld to introduce the reader to the subject but then briefly cite other papers reporting the problem they deal with without clearly identify the benefits and drawbacks of each approach. A comparison of the presented approaches would be useful to understand the peculiarity of each one of them. A minor comment is that in some points, the explanation is not very clear; probably the presence of specific coding examples could be useful to clarify some concepts...”

Answer: We have completely re-set the outline of the paper, by providing a clearer research setting with proper citations and explanations in order to address the comments of all reviewers.

Reviewer comment: “Minor stylistic comments”

1. Avoid re-using the same sentences in Abstract and Introduction. Second paragraph of Introduction is almost identical to the abstract.

Answer: We have revised Abstract and Introduction section for better understanding.

2. Define AOP before using the acronym in Section V

Answer: We have defined AOP in section V, at start of paragraph 2.

3. Remove the repetition of the last sentence of page 3.

Answer: Removed.

4. Rephrase the first paragraph of page 4.

Answer: Rephrased.

Reviewer comment: In Section II B three types of Behavior Variations are introduced but they are not explained later. The flow of Section III is somewhat misleading since it first deals with design-time aspects like the implementation approach (A) then talks about how behavioral variations are activated (B) that affects the run-time then switches back to the design-time by explaining the modularization of the variation (C). A re-arrangement of this section could improve the overall readability of the paper.

Answer: Explanation of the three types of Behavior Variations added.
ANSWERS TO REVIEWER # 2

Reviewer comment: “---I. Introduction: introduction is good and well structured. I can find only a grammatical error in line 5 of first paragraph: the word "smart phone" is a single word…”
Answer: The word is corrected to “smartphones”.

Reviewer comment: “---B.: few words explaining the three kinds of behaviour variations that can be identified.”
Answer: Explanation of the three types of Behavior Variations added.

Reviewer comment: “---C. I would put in italics also the name "layer-in-class" at line 5 of the second paragraph…”
Answer: All instances of the word “layer-in-class” are re-formatted to italic fonts, for coherence.

Reviewer comment: “---D.: the second paragraph is not in the right context. Maybe this part and the subsection E. could be merged in an "Issues" section…”
Answer: Section D and E have been merged in a section named “Open points in COP”.

Reviewer comment: “---grammar: line 1: choose between ‘There are’ and ‘are’…”
Answer: Sentence rephrased.

Reviewer comment: “---AOP should be written at least the first time with the full name (Aspect Oriented Programming)…”
Answer: We have defined AOP in section V, at start of paragraph 2.

Reviewer comment: “---grammar: second paragraph, last line: delete the "(" before the citation…”
Answer: Removed the “(" before citation.

Reviewer comment: “---improve the content of the section VI with more details…”
Answer: Section VI revised and contents improved.

Reviewer comment: “---4th paragraph, first line: citation [2] may be a citation by author is more appropriate…”
Answer: Cited by authors’ name.

Reviewer comment: “---grammar: second paragraph, last line: delete "and have been an open challenge" (duplicated sentence)…”
Answer: Sentence corrected by removing duplication.
ANSWERS TO REVIEWER # 3

Reviewer comment: “---As a suggestion, it is possible to refer to the existing works after describing each feature. For example in section “Implementation approaches”, author could refer to works with “In-language” or “source-to-source compiler” approaches...”
Answer: References added in Section III – A.

Reviewer comment: “---The paper has mentioned some of the work related to the COP, but it does not exist any critical comparisons between the existing works according to the features that they have described. For example in section “Emerging COP application area”, it is better to specify features for each example of COP application...”
Answer: The pedestrian navigation system uses features of EventCJ such as event declaration and layer transition rules. CJEdit encapsulates GUI elements into layers and uses behavioral variations depending on cursor position or an explicit user’s action. In ContextChat case, variations of a context-aware agent are encapsulated in context abstract data type which provides an interface for managing such a structure in order to achieve behavioral consistency.

Reviewer comment: “---As a minor comment, there exist a few typos errors. As an example in “In contrast, COP offer directly the required facilities”, offer should be changed to offers...”
Answer: Sentence corrected.

ANSWERS TO REVIEWER # 4

Reviewer comment: “---The paper structure is not very well organized. In particular, section V was said to present literature review, but it turns out that this section mentioned some related approaches besides COP. Therefore, the reviewer recommends the authors should state this clearer. Moreover, this paper aims to provide a survey on COP, therefore, unless the authors can analyze the influent relationship between COP and other approaches, this portion of knowledge can be removed...”
Answer: We have completely re-set the outline of the paper, by providing a clearer research setting and by better defining the aims of the proposed approach in order to address the comments of all reviewers. We hope that the revisions made address the recommendations of the editors and all referees and that the additional effort has significantly improved the quality of the manuscript.

Reviewer comment: “---Regarding to the portion of content, the reviewer’s opinion is that the part of Context definition and Behavioral variations and layers could be shorten and explained briefer. Besides, three kinds of behavior variation should be described to help audiences understand what they are...”
Answer: Explanation of the three types of Behavior Variations added.

Reviewer comment: “---While representing the survey, the authors miss many citations to their claims. For example, in section III, there are no any citations of the work which make the audiences not able to reference original works. Therefore, the reviewer recommends that the authors should add citation to their statement where needed...”
Answer: References added in Section III.

Reviewer comment: “---Implementation approaches are discussed with two different approaches: in-language approach and source-to-source compiler. These two approaches are not stated explicitly which make the audiences difficult to follow.
Answer: Description of the two approaches has been improved.
Reviewer comment: “---Moreover, in-language approach is not explained how it can be done, while source-to-source compiler approach is described a little. The reviewer’s opinion is that the authors should give a more detail description to these approaches…”
Answer: Description of the two approaches has been improved.

Reviewer comment: “---The authors state that adaptation in COP focuses more on the aspect of computation, but not analyze briefly how it is done in COP. Moreover, the last paragraph of this section is seemed irrelevant to the discussion of adaptation direction…”
Answer: Section D and E have been merged in a section named “Open points in COP”. Adaption directions is strictly related to initialization problem whose description has been improved in the section.

Reviewer comment: “---In this topic, the reviewer’s opinion is that the authors discuss too briefly about the issue. The reviewer expects that some mentioned solutions dealing with behavioral inconsistency should be described how they can be done…”
Answer: Section D and E have been merged in a section named “Open points in COP”. Unfortunately, as stated in the section there is no general solution for behavioral consistency in any case section has been extended to describe more clearly the point.

Reviewer comment: “---The reviewer does not see a clear discussion on maturity level of this research line, therefore is difficult to have knowledge of the current result of this research area…”
Answer: The paper is a form of survey in stated research-line, and we tried our best to summarize the recent trends and challenges in AOP. We have completely re-set the outline of the paper, by providing a clearer research setting and by better defining the aims of the proposed approach in order to address the comments of all reviewers.

Reviewer comment: “---In conclusion, the review’s opinion is that the survey paper provides a broad survey rather than deep survey on the current state of the art of this research line. Some models are mentioned but not actually explain how it can be done. The survey may answer the question what are there in COP, but not how…”
Answer: We tried our best to summarize the broad survey in provided paper size limit. Although, We have completely re-set the outline of the paper, by providing a clearer research setting and by better defining the aims of the proposed approach in order to address the comments of all reviewers. We hope that the revisions made address the recommendations of the editors and all referees and that the additional effort has significantly improved the quality of the manuscript.
Rainbow: Architecture Based Self Adaption

Abdulrahman Kaitoua
DEIB
Politecnico Di Milano
Milan, Italy
Abdulrahman.kaitoua@outlook.com

Yuriy Vaskin
DEIB
Politecnico Di Milano
Milan, Italy
Vaskin90@gmail.com

Abstract—Increasingly, systems must have the ability to self-adapt to meet changes in their execution environment. Existing solutions require human oversight, or are limited in the kinds of systems and the set of quality-of-service concerns they address. The automation might be made by capturing human experience in a set of programmed rules that can run on computers. Rainbow is architecture based adaption model uses Closed-loop control based on external models to dynamically monitor and adapt to system changes while the system is running. Most of the Rainbow code is reusable across systems and this makes it easy to plug in different type of systems with minimal modifications. The authors of Rainbow separated the generic architecture from the system layer and added a translation layer to make it possible to reuse the adaption strategies and the infrastructure across the systems. In this survey paper we describe the Rainbow framework and the new extensions that have been added to the framework recently.

IndexTerms—Architecture based adaption, closed-loop adaption, software self-adaptation.

I. INTRODUCTION

The goal of the adaption is to automate human tasks in system management. One common approach is putting experience of an expert in a computer code which reacts to certain inputs and does the limited adaptation. This computer code usually contains a set of rules that imitate the reaction of the system administrator and produce a set of changes to the system.

One goal of the system adaption is to allow the software engineer to be able to specify an objective, conditions of change, and strategies of change and in order to adapt the system to changes. This system adaption has to be formalized and do not require much effort from the software engineer. For example, a software engineer can take an existing client-server system and make it self-adaptive in regards of performance and response time. The system might adapt for different requirements like adapting the client-server system for security threats, network performance fluctuation, or a threshold latency. Reasons for modern systems to strive for adaption systems are like variable resources, system errors, and changing user priorities. There is a lack of tools to make software adaptation able to maintain systems that they can meet the goals with minimal human oversight.

Architecture based framework can meet the goals of a reusable adaptation system. Fig. 1 shows the primary elements of architectural based self-adaptive system in an environment. We use the general terms managed subsystem and managing subsystem to denote the constituent parts of a self-adaptive software system [1,4,7,8]. The environment refers to the part of the external world with which the self-adaptive system interacts and in which the effects of the system will be observed and evaluated.

The distinction between the environment and the self-adaptive system is based on the extent of control. The subsystem managed comprises the application logic that provides the domain functionality of the system.

Our paper is organized into eight sections. Section II describes the overall Rainbow architecture, and section III refers to the reusability of the framework. In IV there is an example of using the framework. Section V describe particular issues of multiple objectives, usage of preemption, and security example. Finally, section VIII concludes the paper.

II. RAINBOW FRAMEWORK ARCHITECTURE

Rainbow framework is based on an architecture based approach that provides reusable infrastructure along with additional mechanisms which are added to make it easy to specialize the infrastructure to the needs of specific systems. These mechanisms make it possible for the software developer code to simply monitor the system, take adaption and then run the suitable strategy to recover the system status with the minimal interaction of the system administrator [4,6].

The system is divided into two parts; 1. The adaption infrastructure and 2. The system specific adaption knowledge. This division make it easy to identify the reusable parts of the system while the reusability is the major contribution of the Rainbow framework. The adaption infrastructure, reusable part, is divided into A. system, B. architecture and C. translation layer as show in Fig. 2. While the adaption knowledge is
system specific and it is represented by rules, types and properties, operators, and strategies.

**Fig. 2. Rainbow framework.**

1) **Reusable Rainbow units**

   A. **System layer infrastructure**

   The framework, at this level, contains the system access interface and the infrastructure that implements it which includes the system APIs. Probes are the system monitoring and measuring tools that observes and measures various system states while the measurements results then pushed to the upper layer as a result of an enquiry. In extendable systems, the resources are added while the system is running and thus it needs to be discovered by the framework using the Resource discovery mechanism which can be queried for new resources based on resource type and other criteria. Finally, an effector mechanism carries out the actual system modification.

   B. **Architecture layer infrastructure**

   Properties in the architectural model are updated by information gathered using gauges. Gauges collect information from the probes. The system is triggered for adaption when the constraint evaluator finds a violation of the constraints, while the constraint evaluator checks the model periodically. Constraint violation triggers the Adaptation engine, Fig. 2, which selects the appropriate adaption strategy.

   C. **Translation infrastructure**

   A repository is used to store information about the translation that is needed to close the gap between the system and the model. The repository contains information like the real IP address of the host name while the translation infrastructure translate the host name into the IP address when there is a query from the model to the system and vice versa. This layer is built to adapt architectural layer to the system and make most of architectural layer parts reusable across different systems.

2) **System Specific adaption Knowledge**

   Adding self-adaptation to a system using the functionalities that the adaptation infrastructure provides requires using the system-specific adaptation knowledge to tailor that infrastructure. This knowledge includes the target system’s operational model, which defines parameters such as component types and properties, behavioral constraints, and adaption strategies.

**Fig. 3. Client Server system software architecture.**

Fig.3 shows one example of an architecture in which the components represent Web clients and server clusters. Each server cluster has a sub architecture consisting of one or more server components. This architectural model provides a global perspective on the system by revealing all the components and how they connect. The model also contains important properties such as each server’s load, each connection’s bandwidth, and the response time experienced by each client. Section IV, shows this example in details.

### III. Architectural style

While reusable infrastructure helps reduce the costs of adding self-adaptation to systems, it is also possible to leverage commonalities in system architecture to encapsulate adaptation knowledge for various system classes.

To capture system commonalities, Rainbow adapts the notion of an architectural style. Traditionally, the software engineering community has used architectural styles to help encode and express system-specific knowledge. An architectural style characterizes a family of systems related by shared structural and semantic properties. The style is typically defined by four sets of entities:

- **Component and connector types provide a vocabulary of elements, including components such as Database, Client, Server, and Filter; connectors such as SQL, HTTP, RPC, and Pipe; and component and connector interfaces.**
- **Constraints determine the permitted composition of the elements instantiated from the types. For example, constraints might prohibit cycles in a particular pipe-filter style.**
- **Properties are attributes of the component and connector types, and provide analytic, behavioral, or semantic information. For example, load and service time properties might be characteristic of servers in a performance-specific client-server style, while transfer-rate might be a property in a pipe-filter style.**
- **Analyses can be performed on systems built in an appropriate architectural style. Examples include performance**
analysis using queuing theory in a client-server system, and schedulability analysis for a real-time-oriented style.

These four entity sets primarily capture a system’s static attributes. Rainbow extends this notion of architectural style to support runtime adaptation by also capturing the system’s dynamic attributes, both in terms of the primitive operations that can be performed on the system to change it dynamically, and how the system can combine those operations to achieve some effect. Specifically, it augments the notion of architectural style with adaptation operators and strategies, which together determine the system’s adaptation style.

• Adaptation operators determine a set of style specific actions that the control infrastructure can perform on a system’s elements to alter its configuration. For example, a service coalition style might define the operators AddService or RemoveService to add or remove services from a system configuration in this style.

• Adaptation strategies specify the adaptations that can be applied to move a system away from an undesirable condition. For example, a service-coalition system might have a system wide cost constraint. Upon violating it, an adaptation strategy might progressively replace the most costly service with lower grade services until the overall cost falls within acceptable bounds. Strategies are defined using—and therefore constrained by—operators and properties.

Although strategies use operators and properties to adapt systems of a particular style, they are designed for particular system concerns. A system concern outlines a related set of system requirements—such as performance, cost, or reliability—and determines the set of system properties on which self-adaptation should focus, and hence the set of strategies. The system concerns form a subset of the properties in a system’s style. For example, a client-server system may have a style that includes load, bandwidth, and cost properties, while a particular performance concern might focus only on the system’s load and bandwidth properties. Adaptation style and system concerns together comprise two important dimensions of variability from system to system. How much of Rainbow’s system-specific adaptation knowledge can be reused will depend on how similar two systems are in terms of their styles and system concerns. More specifically, types, properties, adaptation strategies, and operators may be reusable if the two systems have matching styles and concerns.

IV. WEB-BASED CLIENT-SERVER SYSTEM EXAMPLE

The case study system consists of a set of Web clients, each of which makes stateless requests of contents from one of several Web server groups, as Fig. 3 shows. The client and server components are implemented in Java and provide remote method invocation (RMI) interfaces for the effectors to use in performing adaptation operations. Clients connected to a server group send requests to the group’s shared request queue, and servers that belong to the group grab requests from the queue.

The system concern focuses primarily on performance, specifically, the response time the clients experience. A queuing theory analysis of the system identifies that the server load and available bandwidth are two properties that affect the response time. Based on this system concern and analysis, the developer defines a client-server style for the system. The major parts of the style include

- ClientT, ServerT, ServerGroupT, and LinkT types;
- ClientT.responseTime, ServerT.load, ServerGroupT.load, and LinkT.bandwidth properties; And
- ServerGroupT.addServer() and ClientT.move (ServerGroupT, toGroup) operators.

The ServerGroupT.addServer() operator finds and adds an available ServerT to a ServerGroupT to increase the capacity. The ClientT.move (ServerGroupT, toGroup) operator disconnects ClientT from its current ServerGroupT, then connects ClientT to the toGroup ServerGroupT. Associated with each client is an invariant that checks to see if its perceived response time is less than a predefined maximum response time. If the invariant fails, an adaptation strategy is invoked.

An example invariant and adaptation strategy is:

```plaintext
invariant (self.responseTime < maxResponseTime)
let responseTimeStrategy = responseTimeStrategy(sell);
strategy responseTimeStrategy (ClientT C) {
  let G = findConnectedServerGroup(C);
  if (query("load", G) > maxServerLoad) {
    G.addServer();
    return true;
  }
  let conn = findConnector(C, G);
  if (query("bandwidth", conn) < minBandwidth) {
    let G = findBestServerGroup(C);
    C.move(G);
    return true;
  }
  return false;
}
```

In this specification, the invariant defines a predicate that determines whether a client’s perceived response time (self.responseTime) is below a threshold (maxResponseTime). If this invariant is violated (indicated by "_"), the adaptation engine executes the strategy responseTimeStrategy.

This strategy first checks to see if the current server group’s load exceeds a predefined threshold. If so, the engine adds a server to the group to decrease the load and thus decrease response time. If, however, the available bandwidth between the client and the current server group drops too low, the engine moves the client to another group, resulting in higher available bandwidth and lower response time.

V. RAINBOW EXTENSIONS

This section shows a recent work on Rainbow are done by [5,3,2] to extend Rainbow for complex systems and for adaptation of security systems.

A. Adaptation to the Presence of multiple objectives

The first version of the Rainbow framework has limited the ability to adapt to the subtleties of choice that is captured form the system administrator tasks. Rainbow was designed to capture and adapt to simple set of requirements which might be performance, cost, or reliability but it does not adapt to...
Adaptation: A Systematic effective software adaptive

Protecting are often manually developed and statically deployed, a self-adaptive framework and named it as architecture based self-protecting software on top of Rainbow architecture based was a work at 2013 by E. Yuan and et al. They developed a multiple objectives that might be contradicted to each other, such as performance and cost of a commodity of servers. The subtleties of trade-off decisions in the presence of multiple objectives. The extended version of Rainbow contained additional keywords for the adaptation language that make it easy to write a tree of actions to make, which formulate a strategy, in case of multiple objectives.

In the extended Rainbow, the strategy is defined as a tree of tactics that tackles common quality issues, with conditions describing each branch. Intermediate system observation occurs after each tactic is executed in order to decide what successive branch to take in the strategy tree [5].

B. Improve Rainbow using preemption

Rainbow framework monitors the target system and make the self-adaptation by incorporating the control layer supervising the system and detecting the problems and then react with the best strategy to adapt and fix the problem. This approach showed that it was effective to do self-adaptation but it is limited because it deals with only one adaptation at a time, this will lead to delay other adaptations until the running adaptation strategy finishes. Some problems are time-critical while delaying such problems until another adaptation is finished might miss the time window opportunity to fix the problem and do the adaptation. Raheja et. al. [3] discuss how to coordinate multiple loops, responsible for multiple concerns (tasks), using preemption in Rainbow. In [3] the authors added the preempted on Rainbow to allow for other time-critical adaptations to be scheduled. Scheduling is based on an algorithm that maximizes time related utility for a set of concurrently executing adaptations. Rainbow with supported preemption of execution adaptation strategy is able to choose adaptation strategy, start new adaptation strategy, and resume preempted strategies. To make it possible to Rainbow to do so the authors of [3] added an adaptation time utility dimension for the self-adaptation while this give Rainbow the ability to priorities among multiple objectives at runtime and choice the most convenient one to run.

C. Rainbow Architecture based self-protecting

The latest addition has been done on Rainbow framework was a work at 2013 by E. Yuan and et al. They developed a self-protecting software on top of Rainbow architecture based self-adaptive framework and named it as architecture based self-protection (ABSP). Conventional techniques for security are often manually developed and statically deployed, a situation that is no longer sufficient against today's sophisticated and evolving cyber security threats.

The detection and mitigation of the security threats, in ABSP, are informed by an architectural representation of the software that is kept in sync with the running system. An architectural focus enables the approach to assess the overall security posture of the system and to achieve defense in depth, as opposed to point solutions that operate at the perimeters. By representing the internal dependencies among the system's constituents, ABSP provides the means to tackle issues such as insider attack. The architectural representation also allows the system to reason about the impact of a security breach on the system, which would inform the recovery process. They conducted three patterns on top of Rainbow. The framework became more reuse across applications and at the same time reduce the effort to have self-protection capabilities [2].

VI. CONCLUSION

The Rainbow approach leverages the notion of software architectural style to characterize and define explicit customization points for tailoring common, reusable infrastructures of the framework to specific styles for multiple quality concerns. The Rainbow approach prescribes an adaptation engineering process that guides a systematic, incremental customization of Rainbow to target systems [6].

REFERENCES

Dear Reviewers,

Please find below the detailed response to the reviewers. In brief, we reworded the introduction, added a paragraph that shows the flow of the paper, and added the missing conclusion section. We did several edits to the paper text in order to reflect the reviewers’ points on the paper quality.

Regards,

------------------------ REVIEW 1 ------------------------

PAPER: 9
TITLE: Rainbow: Architecture Based Self Adaption
AUTHORS: Abdulrahman Kaitoua and Yuriy Vaskin

OVERALL EVALUATION: 3 (strong accept)
REVIEWER'S CONFIDENCE: 5 (expert)

-------- REVIEW --------
The paper is important and it is overall well written. Nonetheless, it’s not easy to follow. In the introduction part, it’s recommended to precise the organization of the rest of the paper in order to make it easy to the reader to follow the paper structure.

We added a part to show the flow of the paper.

Moreover, a conclusion part summarizing the paper is missing.

We added the missing conclusion to the end of the paper.

Regarding the technical content, the information provided are interesting. The paper is consistent and well referenced in all its parts. However in the abstract, the authors mentioned "More than eighty percent of framework code is reusable across systems and this makes it easy to plug in different type of systems with minimal modifications." it will be better to not mention a percentage in the abstract because you need to indicate the source of this information and I didn't find later in the report any indication about it or a reference or source of this percentage.

We took out the percentage number. We still have section II and III that show the reusable components of Rainbow across several systems and this highlights the minimal modifications.

Concerning the formatting of the paper, it is well formatted and presented.

------------------------ REVIEW 2 ------------------------

PAPER: 9
The paper presents an overview of the Rainbow framework used for system performance’s improvement by its ability to self-adaptation. The authors explain its architecture, show examples and highlight its architecture based self-adaption behavior.

Strong points of the paper are as follows:
- clear figures,
- good and easy to follow examples,
- well described architecture style section.

However, the weakest point is the introduction section which I suggest to rewrite,
As the reviewer suggested, we rewrote the introduction to reflect the below points and to make it easier to follow. especially call your attention to:
- using more formal language (i.e. replace 'like' with more formal expression)
  Fixed
- language mistakes and typos (i.e. 'too much efforts' - to many efforts, 'form' - from, 'as best that' - do you mean 'as well as' or 'the best')
  Fixed
- being more precisely (i.e. 'some kind of adaption', 'somewhere in the future')
  Fixed
- figure 1 - consider moving it, with its description, to another section (or new one)

We reorganized the first section to introduce the self-adaption problem and show the architecture based self-adaption, that Rainbow follow, as one of the solutions to the problem.
I would also recommend adding a short description to the introduction section which states what can be found in each section.
We added a paragraph to the end of introduction to show the flow of the paper.
In the second section, in the sentence 'These mechanisms make it possible for the software developer to monitor the system make decision to react and then run the suitable strategy to recover the system status with the minimal interaction of the system administrator [4,6]' is an ambiguity, did you miss a comma?
  Fixed
In the second section, system layer infrastructure:
  'In the environments which are extendable, the resources are added while the system is running [...].’ - there are no environments in the figure 2.
  I assume you refer to the figure 1.
  I was referring to the system itself. The system might add servers or other resources while the system is running. Fixed
In the second section, architecture layer infrastructure:
  'When a constraint is violated then some action is triggered to adapt the change in the system.' - could you develop more elaborated explanation (especially according to the figure 2.)?
  Fixed
In the second section, System Specific adaption Knowledge, figure 3. description - is this example described in section 4.? If so, I would recommend to add a note to keep the paper more coherent.
  A note is added to the end of this section to refer to the example section,
IV.
Section 5., you mentioned about the adaptation for multiple goals and the strategy tree. I believe that adding more information to these topics and provide an example would enrich the paper.
This section was reworded in order to be clearer to the reader. Since we are limited by the paper size, we just referenced examples and details. You can see more information in [5].
I would recommend writing a conclusion section to summarize what was said. We added the conclusion section to the end of the paper.

<This review contains an attachment, see the file review_2.pdf attached to this letter.>

----------------------- REVIEW 3 -------------------
PAPER: 9
TITLE: Rainbow: Architecture Based Self Adaption
AUTHORS: Abdulrahman Kaitoua and Yuriy Vaskin

OVERALL EVALUATION: 1 (weak accept)
REVIEWER'S CONFIDENCE: 3 (medium)

-------- REVIEW --------
This paper explains the Rainbow framework and elaborates to present recent extensions that have been added to this framework. Rainbow framework attempts to address some challenges in the domain of architecture-based self-adaptation by providing reusable infrastructure together with mechanisms for specializing that infrastructure to the needs of specific systems. Rainbow abstracts the behavior of the executing system into the architectural style building blocks and creates an architectural model. It provides an approach to monitor a system and its executing environment and reflect the observation into the architectural model, evaluate the model for constraint violation, and performs adaptation on the running system.

In the introduction, the authors explain the needs for adaptation and the primary elements of a self-adaptive system. In section 2, the authors attempt to explain the basics of Rainbow framework and the relation between its units. Section 3, explains how both static and dynamic aspects of an architectural style is defined by a set of entities. Then, it briefly discusses adaptation style. Demonstrating a case study of a web-based client-server system in section 4 helps readers to realize the role of invariants in adaptation strategy.
While the original Rainbow is limited to perform adaptation based on a
single objective like performance or cost, section 5 mentions to a work which Rainbow is extended to deal with more than one requirement at the same time like performance and cost. This section was not clear to me and the author may elaborate more on this section, as there is still space for the paper. We reworded Section 5 to be clearer for the reader.

The previous limitation is repeated again in section 6, meanwhile the limitation is addressed using a different approach. I would suggest to combine this section with previous one to provide a more unified section. We grouped the last three sections under a common section title to show that these three sections have a different type of extension to Rainbow. Finally, section 7 presents a very recent work, named Architecture Based Self Protection, which is a self-protecting software system to mitigate security threats.
Dynamic software updating: a survey

Massimo Quadrana  
DEIB  
Politecnico di Milano  
Milan, Italy  
massimo.quadrana@polimi.it

Francesco Venco  
DEIB  
Politecnico di Milano  
Milan, Italy  
francesco.venco@polimi.it

Abstract—Nowadays software requirements evolve quickly, and software should be update accordingly. This burdens on the shoulders of software designers and developers, who have to face with the difficulties in designing easy-to-update software and in developing the procedures to perform such updates. Dynamic software updating addresses software updating while maintaining state-consistency with no service interruption. We present here a small survey over the best acknowledged state-of-the-art techniques in this field. We start with the description of a general framework for centralized, single process, procedural software updating and we continue by exploring the issues in object-oriented program updating.

Index Terms—Dynamic software updating, object oriented programming, multithreading

I. INTRODUCTION

Software updating is a very tricky procedure that every software engineer has to face with during the lifetime of a software system or service. The traditional stop-update-restart paradigm often performs poorly in many conditions, and it may be completely unfeasible for some services that cannot be stopped anytime. Dynamic software updating is necessary to overcome these issues. However, it often requires huge effort from developers in finding the right procedures to update software correctly without corrupting the current state of the service that is still running. Moreover, each different developing paradigm (e.g., procedural, object-oriented and multithreading programming) has its own peculiarities that seriously affects the update process.

We present here some of the state-of-the-art solutions that have been proposed in literature so far. We initially analyze the state-of-the-art dynamic software updating framework developed by Hicks and Nettles in [5]. They provide a flexible, robust, easy to use and low overhead framework to update centralized, single process, procedural software. We then pass to object-oriented software systems. In the end we open the problem of dynamically updating multithreaded software systems, which is acquiring great importance with the widespread diffusion of software optimized for multi-core architectures.

In Section 2 we present the Dynamic software updating framework. In Section 3 we present the dynamic update of Object-Oriented software. In Section 4 we introduce the problem of multithreaded software updating. In the end conclusions are presented.

II. DYNAMIC SOFTWARE UPDATING

Hicks and Nettles propose a general-purpose framework for dynamically updating centralized, single process, procedural software [5]. The main goal of their work was to provide a flexible, robust, easy to use and low overhead framework for software updating purposes. Many attempts to address the issues in this context were done previously to this paper. Those systems address either what can be updated, when the updates can occur or how the updates may occur [3, 4, 10]. However, none of them reached the level of completeness achieved by Hicks and Nettles’ work.

The updating framework is based on semi-automatically generated dynamic patches to be applied to the software, coded into the Typed Assembly Language (TAL) [2]. The timing of the dynamic updates was also studied by the authors.

This framework offers flexible software updates, since changes are admitted at the granularity of individual definitions (functions, types and data). Updates may be performed at any time, even when the code is active, coherently with the semantics of the program. The authors used an imperative, C-like language named Typed Assembly Language (TAL), a verifiable native code with important safety properties, including type safety, to guarantee robustness. The mechanism of semi-automatic patch generation reduces the burden on developers, whose intervention is required only in non-trivial cases (tool automation is undecidable).

Dynamic patches act as the main characters of the update process. They describe the dynamic changes between two versions of a program module. They differ from static patches because they must deal with the state of the program. The dynamic patch of some file f is defined as the pair (f', S), where f' is the new version of the file and S is an optional state transformer function, which is used to convert the state accumulated by f into a form usable by the new code f'.

Changes in the code can happen to code and data. There are cases where the code of the state transformer function is not trivial:
because patches are applied to new version, without service-lizes its execution after the when type definitions o-models of the old version, and are t any given time. In this case programmersections are introduced to handle the lack of information; alternatively one can be overcome by modifying the code and data structures to handle the lack of information; alternatively one can differentiate old ‘converted’, data-deficient structures from new versions, and treat the deficient structures specially.

2) It considers the global state, including the heap and static data segment, but not the stack. This allows old and new data/code to be active contemporaneously for a time, which can lead to incorrect and/or unpredictable behavior. Stub functions are introduced to handle multiple versions of the same code running concurrently.

3) Type definition changes are driven directly by programmers. An alternate approach would be system-directed, in which only one version of a type’s data can be present at any given time. In this case programmers define type conversion functions to transform old type into the new, and the system invokes these functions when needed. State transformations can occur on demand, but it requires a greater implementation burden (e.g. in handling exceptions) and may result too inflexible, especially for the time of the updates.

In summary, a dynamic patch is defined as the tuple $(f, S, stub set)$, where $f$ is the new code, $S$ is the state transformer function and stub set is a set of mappings from functions to their corresponding stubs. Figure 1 shows a sample flow of dynamic update process. We start from a stable version 0.2 of program, which has been compiled and run from its source code. Once a new stable version 0.3 of the program is available, patches are generated semi-automatically only from the changed source files in the new version. The version 0.3 of the program is compiled and run, while patches are dynamically applied to version 0.2 to correctly migrate the state from old to new version, without service interruption.

A. Dynamic linking-based updating

Hicks et al. implements dynamic patches through dynamic linking-based updating, that is, patches are dynamically linked into the existing program: the state is transformed locally, and then the transition to the new code occurs. This approach is alternate to state transfer-based approach, in which the new program is compiled and started up alongside the old one, the state is transferred from the old to the new program, which initializes its execution after the received state has been transformed. Dynamic linking offers more simple and flexible updates, mainly because it avoids to create complex state transfer functions and to recompile the entire program at any update, even small.

Once a patch has been dynamically linked into the program, existing function calls and data is redirected to stubs and new definitions in the patch through code relinking, which redirects all references of the old definition to refer to the new ones (Figure 2).

Type definitions are updated through type renaming: the compiler is used to define a new type that logically replaces the old one by syntactically renaming the occurrences of the old name with the new one. The old and the new definitions are both used during type-checking, and transformer function and/or stub functions can be used to convert old instances at update-time (Figure 3).

B. Update timing

Another key aspect is update timing. Two models of updating were considered: the invoke model, in which the update time is determined statically (at compile-time), and the interrupt model, which instead updates are performed dynamically while the program is running.

The usage of an interrupt-drive model of updating assume that the program could be updated at any moment during its execution: the program is interrupted at some point, the update takes place, and then the program is

Figure 1: The dynamic software updating process

**Code/data changes type:** because patches are applied to individual files, the existing referrers to changed items will access them to the old (now incorrect) type. Stub functions offer the same interface of the old version, and are interposed between old callers and new definitions to get the types right.

**Changes to Type definitions:** when type definitions change, and consequently the named types of the code, stub functions convert instances of the old type into instances of the new type and call correctly the new associated functions. This allows more relaxed constraints on update timing, and the behavior of updates is more predictable by programmers.

The state transformation process has some limitations:

1) It is based on the assumption that the new state can be determined from the existing state. However, new data structures may contain information not available in the current program. This state transformation problem can be overcome by modifying the code and data structures to handle the lack of information; alternatively one can differentiate old ‘converted’, data-deficient structures from new versions, and treat the deficient structures specially.

2) It considers the global state, including the heap and static data segment, but not the stack. This allows old and new data/code to be active contemporaneously for a time, which can lead to incorrect and/or unpredictable behavior. Stub functions are introduced to handle multiple versions of the same code running concurrently.

3) Type definition changes are driven directly by programmers. An alternate approach would be system-directed, in which only one version of a type’s data can be present at any given time. In this case programmers define type conversion functions to transform old type into the new, and the system invokes these functions when needed. State transformations can occur on demand, but it requires a greater implementation burden (e.g. in handling exceptions) and may result too inflexible, especially for the time of the updates.
resumed. However, the update can be delayed until certain conditions are satisfied (timing conditions), adding flexibility to the update process, since only the program state at the time of update needs to be considered. However, it lacks of predictability, since it is not guaranteed that timing conditions will be eventually satisfied (correct timing is known to be undecidable).

The invoke model simplifies the problem by specifying update points statically rather than dynamically. The program is coded to perform its own updating by invoking a special update procedure. This ensures the predictability of update times, which can be determined exactly, while it adds the complexity involved in choosing the correct update times.

Determining the correct update timing to ensure both responsiveness and correctness was not explored in depth by the authors. Subsequently other works explored this issue, and the concepts of quiescence and tranquility arose [12].

III. OBJECT-ORIENTED SOFTWARE UPDATING

A different type of problem not addressed in [3] is how to deal with classical object-oriented programming. In such paradigm several new difficulties arise. In fact it is necessary not only to change the value of specific fields at specific points of a program life, but also to translate the state of objects, that can take into account multiple internal variables connected by some logic.

The literature in this field is vast, however it must be noted that many works concentrate on Java. This is probably due to several reasons. First, Java is a very popular and spread programming language, but offer little to none support to dynamic update. Moreover, the Java Virtual Machine offers a distinct advantage respect to other languages such for example C++.

In fact one possible approach, presented by Wurthinger et al. [13] consists in a customized Java Virtual Machine. The proposed system can be used within any Java IDE that uses the standard Java Debug Wire Protocol. In fact the evolution of the code is triggered by commands of JDWP. When such change is requested, for each class a number identifying the class and an array with the new class bytes is transmitted. The redefinition can be done then in parallel with the normal program execution. Without entering into details, the core part of the algorithm is carried by the redefinition of the garbage collection algorithm. Then pointers originally referencing old classes are swapped to instances of the new versions. There are however several limitations. In fact, while swapping method bodies and adding methods, fields and supertypes is always possible, removing a method, a field or a supertype can incur in exceptions.

An alternative idea is represented by customized class loaders [6]. In this case, the program capabilities are extended at the application level: loaders are modified to permit to load updated versions of already loaded classes or components.

Frequently, the enhancing of the run-time update capabilities of Java is obtained using wrappers. Wrappers aim at wrapping old program parts to update them [8, 11]. In order to do so, every caller of a changed program parts must be changed. The application of wrapping can be either statically predefined before program start or triggered at run-time using method body redefinitions based on Java HotSwap [9].

While all these approaches are interesting, they do not generally specifically target the problem on how to manage the state transformation of objects after code update.

A very interesting work in this sense is the one recently presented by Magill at al. [7]. While the work presented is still implemented in Java, the framework outlined is general. The approach for object state transformation explained in the paper is called Targeted Object Synthesis (TOS).

TOS works in two phases: matching and synthesis. First it is necessary to select a class C that has been changed, and both the old and new versions of the program are run with the same input. At every update point encountered during the execution, the TOS records a snapshot of the heap. The objective is to obtain a set of pairs composed by the status of the old and new version of the objects, in order to make a comparison and deduce the operations to update the program. Thus the program must be deterministic (or at least deterministic in the part regarding the class C), in order to obtain the same number of instances of both the old and new version of the class C.

After this set has been built, the matching takes as input the snapshots produced by the test runs. The snapshots are pruned to include only C objects and objects which they refer, directly or transitively. Then, the goal of the matching phase it to produce a one-to-one mapping between objects in each pair of corresponding snapshots. This operation is achieved by identifying a class’s key fields.

Key fields of a class have two properties:

1) No pair of C objects in the same snapshot have the same values for all the candidate key fields

2) For each object in a snapshot of the old program, there is exactly one object in the corresponding snapshot of the new program that has the same values for the key fields.

A simple example could be the user name field of a “User” object in some system.

After matching is complete, a list of pairs of objects of
class C is obtained.

At this point the synthesis takes place. For each pair, the algorithm synthesizes a set of candidate functions that maps the state of the old object to the new one. At the end, the algorithm intersects the sets of candidate functions to produce a function consistent with all examples.

More precisely, the synthesis algorithm proceeds one field at a time. For each filed, it synthesizes a conditional update that is capable of producing all the values for all the values for that field seen in the example pairs.

The authors claim to have successfully updated different complex programs using the system. However, the functions used are simple and are not able to cover all the possible cases, and the approach based on observations of the program heap at different points can be potentially biased, because they are chosen by the users.

In any case, even if TOS fails in building the correct transformer, the partial results can be very useful for programmers.

IV. MULTITHREADED SOFTWARE UPDATING

Another problem not covered by our reference paper regards multithreaded applications. This is out of the scope of this dissertation, so we just give here just a starting point for the readers interested in the related issues. In fact, the main problems in this context are two:
1) How to permit the concurrent execution of old and new code.
2) When to update the different part running in different threads.

A recent work has been presented by Chen et al. [1]. The general idea is to introduce a relaxed consistency model to support permissive changes involving both code and data to multithreaded application software without update timing constraint.

V. CONCLUSIONS

Dynamic software updating is a very useful tool for maintaining software up-to-date without interrupting its execution. Issues related to variable, function and object definitions have been addressed in many papers in literature. However, the entire process is clearly affected by the complexity of the runtime environment. As complexity grows, e.g. in the case of recursive functions, or in multithreaded environments where there is no guarantee on the number of threads that are running our program, effectively updating software is troublesome. We think that there are many fields which can get benefit from new and advanced techniques in dynamic software updating, not least software-as-a-service and high-performance computing systems.

REFERENCES


Impact of Compositional Reasoning on Incremental Verification

Alessandro Maria Rizzi 1, Christos Tsigkanos 2

DEEPSE group - DEIB, Politecnico di Milano
Milano, Italy

1alessandromaria.rizzi@polimi.it
2christos.tsigkanos@polimi.it

Abstract—In the context of compositional reasoning, an overview of techniques which exploit the assume-guarantee paradigm to reduce the complexity of the verification problem is presented. This short essay is structured as follows: in the first section we provide a general introduction to the field and a plethora of different verification approaches related to assume-guarantee. Then we analyse more deeply the ones which regard the application of the L’ learning algorithm which tries to cope with incrementality through a formulation of the component substitutability problem. In particular, first we present a generic overview on these techniques, which is then followed by a detailed analysis of different approaches in the current state of the art.

I. INTRODUCTION

Software systems are usually composed of multiple processes running in parallel; verifying specifications of such complex systems is non-trivial and can be computationally prohibitive. Compositional reasoning aims to shift the focus from a global system level to reason about local, component properties, whose conjunction implies the overall specification.

Given a model of a system and a required specification, model checking can be used to automatically detect behavioural anomalies, such as violations of safety properties. The major barrier to this proving mechanism, is the well-known state-explosion problem. Compositional reasoning, advocates a “divide and conquer” approach, where components are verified separately. Since components may exhibit or violate properties in specific environments, a frame of thought considering these contexts is highly applicable.

The idea behind assume-guarantee (AG) was introduced in [19] in a fairly simple way. The classic Hoare style proof system1 [18] is extended by introducing constraints on the context in which the program will be executing, limiting the way in which it can be changed by other programs and also by specifying in which ways the program itself can modify its context. In this work, the main difference between assume-guarantee and Hoare clauses lies in the fact that Hoare clauses have to be met before and after the program execution; in assume-guarantee instead, conditions which must hold during the whole process execution are specified, in order to capture the interaction between different processes. In the formalization of the assume-guarantee paradigm [24], a formula is expressed by a triple \((\varphi)M(\psi)\); \(\varphi\) and \(\psi\) are formulae in a temporal logic and \(M\) is a program. Intuitively, \(\varphi\) characterizes the environment in which \(M\) is expected to operate, and \(\psi\) a property that \(M\) guarantees to be satisfied.

\[
\begin{align*}
\langle \text{true} \rangle M \langle \varphi \rangle \\
\langle \varphi \rangle M' \langle \psi \rangle \\
\langle \text{true} \rangle M \parallel M' \langle \psi \rangle
\end{align*}
\]

In this fashion, it is asserted that if \(M\) satisfies \(\varphi\), the environment of \(M'\) satisfies \(\varphi\) and \(M'\) satisfies \(\psi\), then the composition of \(M\) and \(M'\) will satisfy \(\psi\). The advantage of assume-guarantee is highlighted, as the composite space of the two programs is not examined. Alternatively, \(\varphi\) is verified using \(M\), and subsequently \(\psi\) with \(M'\) and the assumption \(\varphi\).

A. Assume-Guarantee Approaches

The fundamental paradigm of assume-guarantee reasoning has enjoyed significant contributions from various fields of the formal verification community. Base concepts apart from model checking, include progress on temporal logics, predicate abstraction techniques and probabilistic extensions.

A central concept in state transition systems is simulation preorder. Simulation preorder is a relation between state transition systems associating systems which behave in the same way using the notion that one system simulates the other. In [16] the simulation preorder relation is extended to compositional reasoning. Building on this, various works on refinement have been conducted [17].

A formal model for composite systems of synchronous and asynchronous components supporting compositional and hierarchical reasoning is presented in [1]; upon this model the tool MOCHA [2] has been built, providing explicit support for refinement checking using assume-guarantee reasoning.

Several approaches have been conducted in specifying assumption and guarantees in temporal logics. In [16] a subset of CTL is analysed which preserves satisfaction under composition, also providing a suitable preorder relation. In [29] a study of the complexity of assume-guarantee model checking in the case of assumptions expressed in LTL and guarantees stated in CTL is presented. A comparison among different assume-guarantee specification methods using LTL for assumption and
LTL/CTL for specification with the model checkers SPIN and SMV can be found in [22].

In contrast to conventional type systems that specify component interfaces in terms of values and domains, formalisms utilizing interface automata intend to capture the temporal aspects of a component interaction, regarding information it utilizes as well as information it produces as output. These automata [9], building upon labelled transition systems, are used to model the interface of a component and then used to reason about component-based design by enabling specification, well-formedness, composition and compatibility checking.

These techniques attempt to minimize the global state transition graph by focusing only on communication among component processes, in contrast to considering events that do not relate to component communication [5]. In [12], a game-theoretic framework is used to develop these terms. The component compatibility question is considered as a game between a component and the environment, where output guarantees interplay with input assumptions.

Assume-guarantee has also been used for performing probabilistic verification. In [20], a compositional technique for probabilistic automata is presented, highlighting the transformation to a multi-objective model checking.

Finally some attempts deal with adapting the assume-guarantee paradigm to software testing [15]. In this case the design-time decomposition of different components is used for testing against local requirements.

An important topic regards the link between assume-guarantee and other formalisms for modelling concurrency, in particular separation logic. Separation logic [25] is an extension of Hoare logic for reasoning about programs using shared mutable data structures. On the relation between this formalism and assume-guarantee, many works have been carried out, e.g., [28]. An important result is [14], where it is shown that Concurrent Separation Logic can be considered as a particular case of AG reasoning for well-synchronized programs.

However, the main problems of the application of assume-guarantee approaches, concern the subdivision of the system in different modules and more importantly the assumption generation for each module, which means determining an appropriate $\varphi$ in formula 1 above. Although this task has been delegated to a domain expert in the first publications, many efforts have been done in subsequent works in order for it to be done in an automatic way.

This paper is focused in particular on this problem by exploring the attempts based on a learning algorithm to automatically generate such assumptions.

B. Application and the substitutability problem

As software systems evolve throughout the product lifecycle, modules get inevitably transformed, and verification is required to assure that all the expected requirements are satisfied. Such is the formulation of the component substitutability problem, which addresses the safe replacement of one or more parts of a system. A prime domain for application of compositional reasoning, avoiding re-verification of the whole system is considered a major objective. In a general sense, research is centred on modelling components and demonstrating compatibility between their behaviours, in a divide and conquer way.

As mentioned, the major hurdle associated with applying the assume-guarantee paradigm to reason about components, is generating assumptions. Initial approaches required non-trivial human input to generate assumptions, a manual, complicated procedure that generates much overhead to the overall process.

To counter this, directions that have achieved significant traction in the research community model valid behaviours defined by regular languages. Initially, only finite state models had been used. Subsequently infinite state models have been considered; abstract finite models are obtained from concurrent programs by automated predicate abstraction techniques [7], in an iterative fashion known as counter-example guided abstraction refinement (or CEGAR) [10]. Subsequently, modified behaviours of components are considered as words added or removed from a regular language. This thought framework is highly foundational, as it allows rigorous and sound reasoning based on automata.

As introduced in [8], the component substitutability problem involves two separate types of verification strategies. Containment refers to the quality of a new component’s behaviour to encompass every behaviour of the old one. In this case, using models of the two components as input, the goal is to apply an algorithm to iteratively verify that all the behaviours of a component are included in the new version; alternatively, a counter example of absent behaviours is to be produced for feedback generation. A key feature of this substitutability solution, is the allowance of the existence of more behaviours in the new component than the previous one, in contrast to previous approaches such as interface automata [8]. However, since the new behaviours might violate global properties, compatibility to the rest of the system must subsequently be checked; this notion entails that a new component can be safely integrated into the system assembly.

II. INCREMENTAL ASSUME-GUARANTEE REASONING BASED ON LEARNING

A. The $L^\ast$ algorithm

As already mentioned, the major hurdle with applying assume-guarantee style reasoning is calculating assumptions.

Central to this view are techniques built upon the $L^\ast$ inference algorithm by Angluin [4] with later improvements by Rivest [26]. The algorithm, aims at learning an unknown language $L$ over some alphabet $\Sigma$; it does so by querying a minimally adequate teacher entity with two types of questions. Membership, where given a word $w \in \Sigma^\ast$, the teacher returns true if $w \in L$, and false otherwise. Subsequently, conjecture, where given a finite automaton $C$, the teacher returns true if the language accepted by $C$ is indeed $L$. Otherwise, a counter-example is given in the symmetric difference of $L(D)$ and $U$.

Thus, in a polynomial space and time the procedure iteratively generates the minimal automaton $M$, such that $L(M) = L$. 


Hence, the problem of calculating assumptions is approached in a novel way; assumptions are to be learned iteratively.

The first attempt of using L* algorithm in the verification field is presented in [11]. The model considered involves labelled transition systems for describing the communication interactions between parallel components. The basic idea is the automatic generation of the assumptions by the learning algorithm, which are represented by finite state automata. This is achieved in an iterative process where assumptions generated by L* are verified by traditional model checkers.

The purpose of this original work is not in the area of incremental verification, but just automatic verification: while assume-guarantee reasoning can help in complex verification, i.e., by coping with the state explosion problem, at that time in all the existing assume-guarantee approaches the assumptions had to be manually provided. A similar approach is presented in [3]. In this work it is used a symbolic implementation of the learning algorithm, which uses Binary Decision Diagrams (BDD) state-space exploration, with a modified implementation of the NuSMV model checker.

B. Incrementality

The introduction of the incremental perspective in assume-guarantee has been performed in [8], which is achieved in the setting of the previously stated substitutability problem. This approach is one of the first which exploits the advantage of assume-guarantee to provide incremental verification. Its main idea is the same of [11], which is enclosed in the concept of component compatibility. However this is not sufficient from an incremental point of view, since the new component may not provide all the interesting features of the previous one; this is addressed by the containment concept. The formalism of labelled Kripke structures is applied, and again regular languages techniques and tools are used; the containment problem becomes just inclusion between regular languages, whereas the compatibility problem is handled as in [11].

Although these works introduced automatic assumption generation and a kind of incremental verification, they are based on finite state formalisms. In [27], the approach is extended in order to deal with infinite state models. The key idea behind this is predicate abstraction. With this technique it is possible to reason on smaller systems by focusing on some interesting properties. This is done by selecting a set of valuations from a set of interesting predicates. Each predicate is a boolean property of the system and a valuation of a set of properties is a mapping between each property to a truth value. Thus, the state space can be reduced: each concrete state can be assigned to an abstract one which matches its evaluation of the property set. The transition in the abstract space can be derived in different ways, where the most common one is to add a transition from abstract states A_1 and A_2 if exists a pair of states C_1 and C_2. Then C_1 is mapped to A_1, C_2 to A_2 and there is a transition from C_1 to C_2. However, a given abstraction can be insufficient for a certain property. In order to overcome this, an iterative process like CEGAR is used.

Predicate abstraction is used regarding substitutability in [27], to build finite models of the components under verification. In order to check containment between two modules C_i and C'_i, first of all two models M_i and M'_i are built using predicate abstraction such that C_i ⊆ M_i and M'_i ⊆ C'_i. Then it is checked whether M_i ⊆ M'_i holds. If it holds, then the containment is satisfied; otherwise a counterexample CE such that CE ∈ M_i and CE ∉ M'_i is produced. The counterexample has to be checked if it is real (CE ∈ C_i and CE ∉ C'_i), otherwise the model has to be refined.

Regarding the compatibility part, the same considerations of abstraction apply. This work [27] improves the technique used in [8] by considering the substitution of different components simultaneously. In addition, incrementality is sought by changing the L* algorithm with a dynamic variant. In practice, incrementality is achieved by reusing the table constructed by L* for verifying the compatibility of the previous component to the new one.

C. Conceptual extensions and current state of the art

An extension of the approach is presented in [6], by applying the reasoning technique to branching time modelling. In order to achieve this, the finite automata are substituted by a tree automata. Additionally, the L* algorithm is subsequently modified in the new L^T algorithm which can build the minimum deterministic tree automata in a similar way to how L* works. Basically L^T is the application of L* to tree automata which can be considered as a bi-dimensional extension of finite state automata.

Additional improvements have been provided in [23], where a technique for alphabet refinement has been developed: since in order to verify some properties it is not necessary to consider the finite automata’s whole alphabet, it is possible to improve performance by only considering a small subset. The algorithm for such refinement is quite similar to the one used with predicate abstraction. The standard verification is performed using only a subset of the alphabet; if the verification succeeds, then the algorithm stops, otherwise checking if the counterexample is real or spurious is done, in which case the alphabet used is refined and a new iteration is performed.

However this work is more important for applying the learning algorithm also to symmetrical reasoning. In order to introduce this recall that there are two distinctive ways of using an assume-guarantee reasoning. The first one, considered in all the learning-based approaches considered up to now, is composed of an asymmetric (or non-circular) rule in which the different components have different roles; in particular there is one component M_1 which does not assume anything; an other component relies on the guarantees provided by M_1.

In symmetric (or circular) reasoning, all the components can make some assumptions and guarantees at the same time. This kind of assume-guarantee reasoning has some drawbacks, since in general it cannot be both sound and complete [21]. For this reason the reasoning framework adopted in the work is enriched with additional constraints.
\[
\begin{align*}
\langle A_1 \rangle M_1 \langle P \rangle 
\quad & (4) \\
\cdots
\quad & (5) \\
\langle A_n \rangle M_n \langle P \rangle 
\quad & (6) \\
\mathcal{L}(\bar{A}_1 \parallel \cdots \parallel \bar{A}_n) \subseteq \mathcal{L}(P) 
\quad & (7) \\
\langle \text{true} \rangle M_1 \parallel \cdots \parallel M_n \langle P \rangle
\quad & (8)
\end{align*}
\]

In particular, the reader is urged to notice that all components share the same guarantee \( P \); the last rule means that the parallel composition of the complements of each assumption must be contained in the guarantee \( P \). The approach followed involves the separate application of \( L^* \) to each module for retrieving its assumption. These assumptions are used to check the final property. If it is satisfied, the procedure terminates successfully; otherwise the counterexample obtained is again checked and eventually leads to another iteration of the algorithm. Assume-guarantee reasoning techniques based on learning have been applied also outside the traditional verification field. For example, learning algorithms have been applied also in probabilistic assume-guarantee verification [13].

D. Conclusions

The fundamental paradigm of assume-guarantee reasoning has enjoyed significant contributions in various subdomains. In this essay, a dense overview of the field has been presented, highlighting the application of the \( L^* \) learning algorithm which tries to cope with incrementality for automatic assumption generation. Various extensions build upon the base formalisms; results are promising.

REFERENCES