A Bio-Inspired Self-configuring Observer/ Controller for Organic Computing Systems

Ali Tarihi
Department of Computer Engineering and Science, Shahid Beheshti University, Tehran, Iran
a_tarihi@sbu.ac.ir
Hassan Haghighi*
Department of Computer Engineering and Science, Shahid Beheshti University, Tehran, Iran
h_haghighi@sbu.ac.ir
Fereidoon Shams Aliee
Department of Computer Engineering and Science, Shahid Beheshti University, Tehran, Iran
f_shams@sbu.ac.ir

Received: 04/Jul/2015 Revised: 29/Jun/2016 Accepted: 25/Jul/2016

Abstract

The increase in the complexity of computer systems has led to a vision of systems that can react and adapt to changes. Organic computing is a bio-inspired computing paradigm that applies ideas from nature as solutions to such concerns. This bio-inspiration leads to the emergence of life-like properties, called self-* in general which suits them well for pervasive computing. Achievement of these properties in organic computing systems is closely related to a proposed general feedback architecture, called the observer/controller architecture, which supports the mentioned properties through interacting with the system components and keeping their behavior under control. As one of these properties, self-configuration is desirable in the application of organic computing systems as it enables by enabling the adaptation to environmental changes. However, the adaptation in the level of architecture itself has not yet been studied in the literature of organic computing systems. This limits the achievable level of adaptation. In this paper, a self-configuring observer/controller architecture is presented that takes the self-configuration to the architecture level. It enables the system to choose the proper architecture from a variety of possible observer/controller variants available for a specific environment. The validity of the proposed architecture is formally demonstrated. We also show the applicability of this architecture through a known case study.

Keywords: Organic Computing; Observer/ Controller Architecture; Self-* Properties; Self-Configuration; Formal Verification.

1. Introduction

The arising complexity in computer systems has led to the introduction of new paradigms such as Autonomic Computing [1], Organic Computing (OC) and Pervasive Computing [2] that cope with complexity. Organic Computing is centered around cooperating entities which are sometimes called agents [2]; each of which has a set of capabilities. These capabilities are mostly sensors and actuators that enable the agents to interact with their environment and perform what is expected from them. Agents are also capable of communicating with each other and ultimately contribute to the creation of a single collective OC system. Because of the complexity in OC systems, an explicit design cannot be given for each possible situation. Therefore, a degree of freedom in decision making is given to the agents, so that the system can be managed collectively based on the local decisions [3]. This leads to the emergence of properties, like self-healing, self-configuration, and self-optimization at the system level that are called self-* in general [2].

The main drawback of obtaining self-* properties in this manner is the possible emergence of unwanted behaviors due to the lack of system-wide vision in the local decisions. Coping with this problem implies using a control mechanism. To achieve this goal, the Observer/Controller architecture or o/c (for short) has been proposed for OC systems [3]. The observer as the name suggests has to observe the system passively and reports to the controller for proper actions. The part of the system that is under observation is usually called System under Observation and Control (SuOC) [2]. The generic o/c architecture [3] is the most known and cited o/c architecture, and many of the existing researches in OC refine the generic o/c architecture for their own purposes; for example, see [4]-[6]. The generic o/c architecture is studied deeper in Section 2.

Self-configuration, which is related to the ability of the system to reconfigure itself dynamically [7], is among self-* properties that the o/c architecture tries to control. It is defined as “the set of all system and environmental attributes that can be modified by control actions” [8]; these attributes are divided into two categories: internal and external [8]. The former attributes that controlled by the system while the latter are controlled by the user or an external entity.

* Corresponding Author

In OC, self-configuration is mainly achieved in the SuOC level, meaning that the SuOC is reconfigured accordingly by the o/c component. In this way, the benefit of the self-configuration property is not present in the component level, or in other words, OC systems are committed to have a fixed o/c component governing the SuOC. Therefore, any rearrangement or change in the o/c component is prevented, which will be a major drawback to environments where multiple o/c components configurations are applicable. This issue motivated us to enable the self-configuration property at the o/c component level and achieve a first step toward a self-* enabled o/c component for the o/c architecture.

Hence, the main contribution of this paper is focused on promoting the self-configuration property to the o/c component level. In order to achieve this goal, we propose a bio-inspired self-configuring o/c architecture that configures itself according to the operational parameters. The bio-inspiration in our work comes from the notion of cell differentiation process [9]. We also use the feature model concept from the software architecture, or more precisely, the software product line so as to capture o/c component configurations. In addition, we present and evaluate our ideas using formal methods.

For this purpose Section 2 is dedicated to the background concepts, especially the biological ones while Section 3, the related work is reviewed, and then the proposed o/c architecture is presented in Section 4. In order to validate the proposed architecture, it is specified and verified using formal methods in Section 5. Section 6 shows the applicability of the proposed o/c architecture through a known case study, and finally, the last section is devoted to the conclusion and some directions for future work.

2. Background

2.1 The Generic o/c Architecture

The generic o/c architecture [3] consists of a set of components shown in Fig. 1. The observer component in this architecture is composed of several sub-components that monitor and use data from the SuOC for analysis and prediction; the results are aggregated and then used by the controller.

The controller component is in charge of executing the decisions made by its learning components for the SuOC. Three sources of data are given to the “aggregator”, and then the aggregated data is used by the “mapping” (rule base) and “rule performance evaluation” subcomponents of the controller. This component has both online and offline learning subcomponents. The “rule performance evaluation” subcomponent is for online learning, which updates the existing rules as needed, whereas the “rule adaptation” and “simulation model” subcomponents are due to offline learning, they create new rules and delete the old ones. The “objective function” represents the user interactions that affect the control of the system. Finally, an “observation model”, which is applied by all the observers in the OC system, is selected by the controller to indicate the observable attributes and the proper analysis method and parameters for observation (e.g., the sampling rate).

The generic o/c architecture has three variants [3]: The centralized variant that consists of a single o/c component and a single SuOC, while decentralized variant has many SuOCs, each with a dedicated o/c component. The third variant is the multi-level o/c architecture, in which one of the o/c components is in the highest level, while the underlying SuOC consists of a collection of smaller SuOCs. These smaller SuOCs in turn can have their own SuOCs, resulting in a fractal like structure.

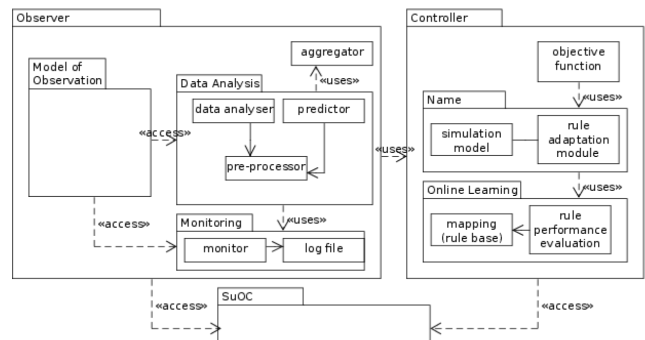

Fig. 1. The generic o/c architecture [3]

2.2 Cell Differentiation

For introducing the notion of cell differentiation, it is helpful to have a few words about the functionality of cells and the difference between them.

Multicellular organisms (or metazoons) need different types of cells (e.g., blood cells and neurons) so as to survive. Each cell type has a variety of functions to perform, some are common to all, while others are special to that type of cell. From a Biochemical point of view, proteins contribute to any biological function, and therefore, the difference between the cells comes from the difference in the proteins they have. For example, red blood cells have the special function of transferring oxygen in the blood because of hemoglobin, a protein that they have.

Regarding the mentioned concepts, interesting questions arise: 1) where proteins come from and 2) what makes each cell produce a special subset of proteins? The precise answer to this question is a major research topic in modern biology. But, we intend to present a brief answer from the biology literature that is both related and useful for the bio-inspiration mechanism used in this paper. All proteins inside a cell are encoded in a large biochemical molecule named DNA, which has many sections called genes that are used in a process called “transcription” [9]. In this process, the cell produces proteins from the DNA (the answer to the first question).

When a gene is used in creating proteins, it is said to be expressed. The term ``repressed'' is employed when a gene is not used for some reason such as some chemicals [9]. In other words, the expression/repression of genes controls the function of cells via proteins. This means, that the difference in the proteins produced by the cells

comes from the expression/repression of their genes (the answer to the second question).

With this introduction, cell differentiation can be defined as follows. All of the multicellular organisms begin in an embryonic state (before the birth) from a single cell called zygote, and all the cells evolve from it.

With each generation, some genes are expressed/repressed, and ultimately, specialized cells are evolved. This process which is most active before the birth is known as cell differentiation, which has critical role in the life of multicellular organisms. There are some decisive factors that affect cell differentiation [9], especially the gene expression/repression, that results in different functionalities in the cell. Cell differentiation also depends on some chemicals, like growth factors and inducers, which can cause or prevent cell differentiation [9]. Another factor that affects cell differentiation is the micro-environment (also called niche) which surrounds the cells. For instance, keratinocytes (skin cells) are affected by the micro-environment, and in this way, specialize and form the skin [9].

This is only a brief introduction to cell differentiation, the interested reader is referred to [9] for more information.

2.3 Feature Model

The feature model comes from Feature-Oriented Domain Analysis [10] "describe a hierarchy of properties of domain concepts" [11]. This model helps to determine which combinations of features can be selected for domain concepts. If we consider the domain of wrist watches as an example, some of the general statements that can be given are: The watch can be either digital or mechanical, displaying the time by digits or hands. , and in some showing the date.

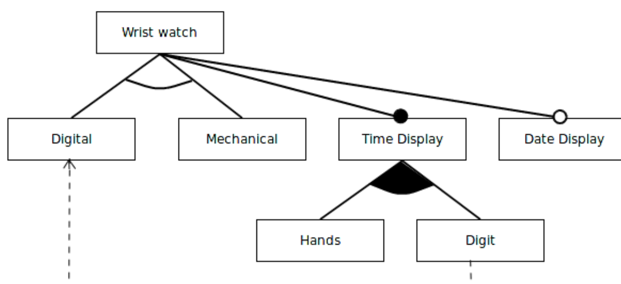

Fig. 2. A simplified feature diagram of the wrist watch example [10].

Feature diagram is the graphical representation of a feature model. Fig. 2 is a simplified feature diagram of the wrist watch example. The full dots indicate the "mandatory" features (like Time Display) that must be present in any domain concept regarding this feature model, while the empty dots indicate an "optional" property (like Date Display). The arc between Digital and Mechanical denotes "alternative" relationship (i.e., only one of these two features must be selected). There is another "or-relation" (for example, between Hands and Digit) that indicates any number of features that can exist together (e.g., a digital watch can have either digital hands or digits). Other two common relationships are "require" and "exclude" relationships [12]. The "require" relationship between Digit and Digital represented by a dashed arrow indicates that Digit display cannot be selected without a digital wrist watch. For example, digit display is only available to digital wrist watches. The "exclude" relationship is used to indicate that two features cannot exist together. It is usually displayed by double headed dashed arrows in feature diagrams. Having these relationships, feature models have many uses. In software product lines they are used for defining products and configurations [11]. In Section 4, we use the feature models for the configuration definition.

3. Related Work

Brinkschulte et al. [13] proposed an OC operational mechanism called Artificial Hormone System for task distribution among heterogeneous processing elements based on three types of hormones, namely, eager value, suppressor, and accelerator. The eager value determined the appropriateness of a task to be executed on a processing element. The suppressor and accelerator had two opposite effects on the process elements. The former increased the chance for taking tasks, while the latter tried to repress the execution of tasks. The Artificial Hormone System achieve

d many self-* properties by employing various sub-types of these three hormones [2] that participate in a hormone based control loop [2], in which each process element declares the appropriateness of a given task execution. Hormones from other process elements affected appropriateness value declared by the process elements. The overall effects of hormones on the control loop decided which process element would execute the task. The OC system achieved self-configuration by finding a suitable initial configuration for tasks based on the function of these hormones.

Roth et al. [14] suggest an OC middleware consisting of an organic manager and a set of ordinary services (like a database service) that communicated via the middleware running on distributed nodes for ubiquitous and pervasive computing. The goal of the middleware was to enable self-* properties (including self-configuration) for ordinary services. In this regard, the organic manager monitored the middleware and incorporated some self-* services, each of which was responsible for one self-* property. Using these self-* services required specific information provided by each ordinary service. However, since self-* services are independent, they might make conflicting decisions. Using the approach of Satzger et al. [5], a high-level planner component was added to the middleware in order to resolve the possible conflicts.

The o/c component of the middleware was inspired from the MAPE cycle of IBM autonomic computing [1] consisting of "Monitor", "Analysis", "Plan" and "Execute" stages. In the monitor stage, an information pool manager component managed the information pools containing the

information needed for the control mechanism. The analyze stage had an event manager and a fact base components; when an event occurred, in order to use the event for planning, the event was transformed into facts. The plan stage, consisted of both a low level planner and a high level planner components. A plan was devised and then executed so as to solve any detected problem using these two planners. The low level planner component had a reflex manager component that managed the low level reflexes subcomponent. The reflexes subcomponent acted like a cache for previous system rules. Having this cache, if a previous decision was applicable to the current state, it would be applied. The high-level planner finds solutions to situations that are not solved by the low level planner. The high-level planner is managed by the high-level manager that converts the facts into a high-level language to solve by the planner. Finally, the actuator executes the plan given by the plan stage. Using this structure, the self-configuration service in this middleware determines the required resources for ordinary services and "triggers an auction" [14] so as to find the best node for that service.

Nafz et al. [6] proposed the Restore Invariant Approach (RIA) controller in which a set of reconfiguration algorithms processed a set of resources and agents having the required capabilities. The OC system tried to keep a set of invariants, regarding these invariants, result checker component examined the results of the used reconfiguration algorithms before the actual reconfiguration. Reconfiguration algorithms component was responsible for achieving self-configuration and was used in determining which capability must be active on which agent (as the initial configuration). These algorithms were also used for reconfiguring the agents whenever the invariant was violated.

The ORCA project was aimed at "transferring self-* properties to robotic systems" [15]. In this project a multi-level o/c architecture with decentralized modules was proposed. One type of these modules included Organic Control Units that monitored and controlled other modules and configured them for operation. The lower-level organic control units were themselves monitored by higher-level organic control units leading to a multi-level self-configuration mechanism.

In summary, it can be said that all of the mentioned works only covered self-configuration in the SuOC level and do not extend it to the o/c component; hence, it lost the advantages of self-configuration in this level by having a fixed o/c component. The fixed architecture prevents any rearrangement or change in the o/c components, which will be a major drawback to environments where multiple o/c component configurations are applicable.

4. Proposed Architecture

Our approach to enabling bio-inspired self-configuring o/c (sco/c) is architectural. We try in this section, which is divided into several subsections, to explain the rationale behind our architectural decisions. First, we explain the influence of the bio-inspiration from the cell differentiation on our architectural decisions as principles extracted from cell differentiation. Then, an illustrative example is introduced that will be used throughout the paper for demonstrating our proposed architecture. The third subsection presents an architectural meta-model that incorporates our core ideas.

4.1 Bio-inspiration for Self-Configuration

The cell differentiation process can be considered as an advanced form of self-configuration in which each cell self-configures its functionality accordingly. To be able to apply the benefits of cell-differentiation, we need to have building blocks analogous to the cells. This leads us to the agents and the first core principle in sco/c architecture.

Principle 1. In sco/c architecture, the system is considered as a collection of communicating agents.

Though this principle is not novel, it is required as a base for the application of the other bio-inspired principles. Based upon Principle 1, we can adopt the concept of genes. The difference between the cells is related to the expressed/repressed genes. This must be shown in the sco/c architecture, too. Subsequently, we must be able to express the system in terms of genes, which their active/inactive state affects the behavior of the agents and ultimately the system.

Just like the multicellular organisms, where everything is expressed through the genes, we need an alternative concept so as to capture the sco/c architecture. We propose the use of the "capability" concept that has also been used in organic computing [2] as well as multi-agent systems.

Principle 2. For all the agents, every function must be definable in terms of capabilities. Every functionality is available if and only if the corresponding capability is activated. Likewise, the deactivation of any capability will result in the lack of corresponding functionality.

This is analogous to gene expression/repression in cells. This principle shows what the agents do. In relation to this principle the question of that what should be done about the "capabilities" of the o/c component may arise, which can be answered in various ways. Regarding the bio-inspiration, it can be noticed that all the functionalities of a living organism, even the control mechanisms, are coded into the genes. Since we have chosen capabilities as counterpart of the genes, the control mechanism of the system must be represented in terms of capabilities.

This is a key principle in sco/c architecture that results in a uniform view of the system that makes the agents more like cells in multicellular organisms. This means everything, including the control mechanism is represented using one concept. This principle blurs the distinction between SuOC and the o/c component compared to other o/c architectures. So as to simplify the architecture description, we distinguish the capabilities representing the o/c component from the rest of the capabilities.

We promote the concept of agent capabilities by introducing another set of capabilities called Organic Computing capabilities.

Principle 3. The Organic Computing capabilities or OC capabilities are related to the o/c component. They participate in the observation and control of the OC system. The set of OC capabilities includes the sub-components of the o/c component and follow Principle 2 in terms of activation and deactivation.

In order to distinguish between the OC capabilities and the capabilities that have nothing to do with the control mechanism, we will refer to the latter as normal capabilities. In other words, agents use normal capabilities in performing their normal tasks. This includes the agent sensors and actuators for interacting with their environment.

For example, when the RIA controller is identified as the suitable o/c component, the invariant monitor, reconfiguration algorithms and result checker are the needed OC capabilities. In addition, the "reconfiguration algorithms" capability needs the "invariant monitor" capability, while in turn it is needed for the "result checker" capability.

Principle 4. In order to form the control mechanism, the required relationships between the OC capabilities must be established.

For example, an OC capability like "data analyzer" from the generic o/c architecture (Section 2), so as to operate, needs to be somehow connected to a monitoring OC capability. In this way, a set of relationships between the OC capabilities is formed. It can be said that o/c architecture can be realized via cooperation of agents using OC capabilities with regard to their relationships. So far, the presented principles can create the foundation needed for sco/c architecture. The self-configuration property of the sco/c architecture is also influenced by bio-inspiration as follows. In the beginning stages of cell differentiation, only zygote exists with no differentiation. After that, some genes are expressed in the following generations, and thus, specialized cells appear.

Principle 5. In the beginning, no OC capability is "active"

The control mechanism is the first thing to be realized. Since the control mechanism is realized by OC capabilities, OC capabilities must be activated in such a way that the relationships between them are preserved.

This principle ensures that the system only operates when there is a control mechanism formed using OC capabilities (Principle 4). This principle prevents the system from operating without a control mechanism.

Principle 6. Micro-environment and the chemicals present in it are required for the cell differentiation process.

The micro-environment is achieved using the concept of neighborhood that is common in multi-agent systems meaning that when an OC capability is active in a neighborhood, it can prevent the other agents from activating it. Also, when a needed OC capability is absent from a neighborhood, it must be activated. For the chemicals (for example, inducers and growth factors), messaging will be used. Similarly, when an OC capability residing in a different agent is needed by another OC capability (having "require" relationship) messaging is used.

Principle 7. Each cell differentiates using its genes. Gene expression/repression play a key role in deciding what gene should be expressed/repressed.

This principle implies that local control of gene expression/repression is needed. Each agent must know the relationship between the OC capabilities and when it should activate them, and it must be able to activate/deactivate them when needed.

Based on these principles, the sco/c architecture can be presented, but first, an illustrative example is presented in the next subsection in order to help understand the application of the bio-inspired principles in the sco/c architecture.

4.2 Illustrative Example

The example is a self-organizing resource-flow system [5], [6] and [16] in which a number of resources are processed by independent agents. The process of each resource consists of a set of tasks performed on each resource by the agents. Each agent has a collection of tools, each of which can perform a specific task. These tools might fail, rendering the agent unable to perform one or more of its tasks. The goal is to reconfigure the agents in a way that the processing of resources can still continue. The reconfiguration mechanism changes the assignment of tools to the agents, or in other words, changes the tasks they perform. It must be noted that at some point no reconfiguration can be done so as to keep the process going on. For instance, when all the instances of a tool is broken, no agent can perform the task related to that tool anymore. This will leave no possible reconfiguration. The number of tasks for the resources is not restricted to any specific number, but in [5] , [6] and [16] three tasks for each resource were considered for identical agents, which were drilling a hole in the resource (it a work piece), inserting a screw in it and tightening the screw.

In order to keep the illustrative example simple and tangible as possible, we will use this particular instance of self-organizing resource-flow system (as defined in Satzger et al. [5] and Nafz et al. [16]) as the illustrative example.

4.3 Architecture Meta-Model

Fig. 3 shows the sco/c architecture meta-mode that supports and incorporates the bio-inspired principles mentioned before. The reason for proposing this meta-model is to point out the sco/c architecture works for systems that follow this meta-model and have its main elements.

A closer look at the meta-model shows the influence of principles 1, 2 and 3 clearly, since the meta-model is based on interacting agents with normal capabilities and general OC capabilities. Communication between agents realizes Principle 6 (i.e., micro-environment and chemicals in it).

There are two additional OC capabilities in the meta-model, named regulation and expression. The introduction of these two mandatory OC capabilities helps to realize the needed local control (Principle 7) and contribute to the self-configuration in the whole system. These two OC capabilities are defined more precisely as follows:

- The regulation capability must identify the proper o/c component configuration by activating the needed OC capabilities and deactivating the unnecessary ones. This function is similar to what happens inside each cell.
- The expression capability resolves the dependencies between OC capabilities that are identified by regulation. The expression capabilities of various agents collaborate with each other when needed.

Returning to our illustrative example, the robots are independent identical entities that can be safely considered as agents. Their capabilities are drilling, insertion and tightening. In this way, principles 1 and 2 are satisfied. The OC capabilities and activation/ deactivation of these capabilities in the example will be introduced later.

4.4 Self-Configuration for the SCO/C Architecture

The demonstration of self-configuration in the sco/c architecture requires the description of the usual behavior of the system. The scenario for sco/c can be described in short as follows. The system begins in an embryonic state in which no OC capability is active (Principle 5).

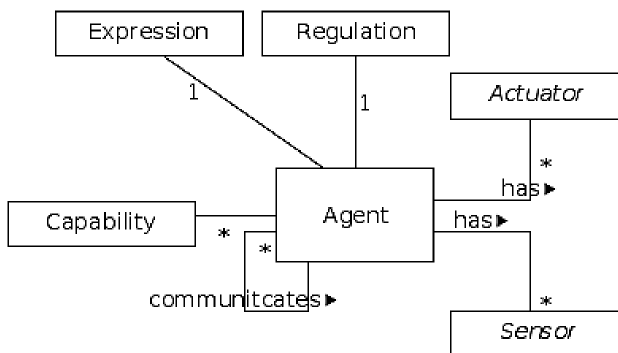

Fig. 3. The sco/c architectural meta-model

Firstly, both the regulation and expression are activated, so the local control is realized. The regulation capability identifies the OC capabilities needed to be activated. The expression capability resolves the dependencies. After that, the OC capability/capabilities that must be activated in each agent is indicated by a distributed algorithm. Finally, the desired OC capabilities are activated by the regulation capability.

Until the end of this section, the above mentioned scenario is presented with more details. The identification of the needed OC capabilities in the current sco/c architecture is in form of rules supplied by the architect in the design time (wrong rules will lead to undesired outcomes). Therefore, the validity of the mechanism is totally dependent on the mindset of the architect since the control mechanism in the sco/c architecture cannot understand the semantics of such rules. These rules have the generic form of “if-then” meaning that if a condition is matched, some capabilities are considered to be needed (i.e., an o/c architecture configuration).

The feature model (Section 2-3) is a good candidate for capturing the OC capabilities and their relationships in the form of a hierarchy. The possible configuration for the o/c component can be given through the feature diagram.

Fig. 4 shows a feature diagram for the illustrative example incorporating [5] and [16] as the two related works presented in Section 4-2 (the organic middleware [5] and the RIA controller [16]). The OCu represents the organic middleware controller. It must be mentioned that other o/c component configurations can be incorporated in the feature diagram, but we used the ones that suit our illustrative example the best. As can be seen, the o/c component mandates both observer and controller. The observer can have one of the two o/c components (invariant monitor and information pool manager). The “require” relationship indicates inter-tree relations between the OC capabilities. For example, the RIA controller can be realized using the “require” relationship between invariant monitoring and RIA controller. The final subcomponents of the RIA controller must be realized because of the mandatory relationship between the result checker and the reconfiguration algorithm.

After the identification of the OC capabilities by the regulation capability in each agent, all of the agents know that the o/c component configuration must be activated.

Next, each agent compares its OC capabilities with the needed OC capabilities for realizing the selected o/c component configuration. There might be multiple instances of each needed OC capabilities identified by the agents. In other words, many agents may have the needed OC capabilities. They are announced to the neighborhood and ultimately all the system. After that, a distributed election algorithm, such as the one introduced in [17], elects the desired OC capabilities. The algorithm denotes which OC capability in which agents must be activated. Therefore, any other OC capability except those indicated by the algorithm must be deactivated. The reason for deactivation is that there might be a previous o/c component configuration. If this deactivation does not happen, there might be another configuration active, and this might lead to unexpected results. This causes the sco/c architecture to be usable in variety of environments, i.e., if the regulation can determine the type of the o/c component configuration, it make the system operate automatically and without manual intervention. If the activation/deactivation process is completed, all the desired OC capabilities are activated, and a special configuration of the o/c architecture can be realized. After a successful configuration, the OC system starts to operate. It can be said that, in the sco/c architecture, there are two distinct self-configuration and operation stages. Self-configuration is involved with the realization of the control mechanism, while in the operation stage, the SuOC is reconfigured accordingly.

If we consider Fig. 4 as the feature diagram, the following argument can be presented for the RIA

controller and the organic middleware: The former uses a centralized variant of the o/c architecture, while the latter uses a decentralized one. The decisions are centralized in the former and easier to achieve, while in the latter, the decisions are made independently and then coordinated. So, a key architectural decision would be to choose and employ a proper new configuration from these two alternative o/c architectures. As one of the configurations, we can assume the computational power of the agents in the regulation rules supplied by the architect. The computational power is chosen because the organic middleware requires that its instances run on each agent and make decisions, therefore requires higher computational power, and consequently power usage. This power usage is a major concern when it comes to general applications of pervasive or ubiquitous computing. On the other hand, the RIA controller is centralized and has more lightweight components (or in other words, less computational power) than the organic middleware.

When the decision is made and one of the variants is chosen, the required components should be identified. For instance, if we want to select the RIA controller itself, all the agents should choose the respective components: invariant monitor, reconfiguration algorithms and result checker. In our illustrative example, since the agents (robots) are identical, all of them announce the three needed OC capabilities. The distributed election algorithm will eventually specify the appropriate allocation of the OC capabilities. The expression of each robot activates the selected OC capabilities and deactivates the others. After that, the system can begin its normal operation.

5. Formal Specification and Verification

In order to present the sco/c architecture more precisely and with less ambiguity, and verify its self-configuration property formally, the sco/c architecture is specified. We also used Linear Temporal Logic (LTL) [18] for expressing invariants needed for the sco/c architecture. Our approach for verification is using model checking capabilities of the Maude formal tool [18].

5.1 Specification

Our specification is focused on the self-configuration phase because it involves all of the contributions of this paper. Specifications 1 through 5 describe the regulation and expression capabilities and the governing conditions in Maude.

Specification 1. This specification formalizes Principle 7 for the regulation capability. It is a collection of rules supplied by the architect.

$$\text{Condition}: \mathbb{P}\text{Parameter} \rightarrow \text{Boolean} \quad (1)$$

$$\text{RegulationRule}: \text{Condition} \rightarrow \mathbb{P}\text{Capability} \quad (2)$$

$$\text{Regulation} == \mathbb{P}\text{RegulationRule} \quad (3)$$

$$\text{regulate}: \text{Condition} \times \text{Regulation} \rightarrow \mathbb{P}\text{Capability} \quad (4)$$

Parameter represents any information (such as the number of agents or agent distribution) that can be used for decision making and selecting the OC capabilities. Condition (Declaration 1) is a function that takes a set of parameters (as a specific condition) into account and returns a Boolean value representing the validity of that condition. For example it checks if $\text{ping time} > 10\text{ms}$ and $\text{bandwidth} > 64\text{K}$ as two parameters constituting a specific condition holds or not. Regulation (Declaration 3) is defined as a set of RegulationRule (Declaration 2) RegulationRule defines the elements of the regulation capability in the form of a rule that specifies a proper set of capabilities for each condition. The regulate function (Declaration 4) specifies the regulation function of the regulation capability. It takes a condition and the set of RegulationRule and returns the capabilities that are needed to be activated in that condition.

Specification 2. Similarly, Expression (Declaration 5) denotes the expression capability. It shows the relationships reltype between the OC capabilities according to the feature diagram. This specification formalizes Principles 3, 4 and 7.

$$\text{Expression}: \mathbb{P}\text{Capability} \rightarrow \mathbb{P}(\text{reltype} \times \text{Capability}) \\ \text{reltype} = \{\text{mandatory}, \text{optional}, \text{excludes or requires}\} \quad (5)$$

This specification formalizes the relationships discussed in Principle 4. Using reltype defined in this specification, the relationships between OC capabilities can be represented.

Apart from these specifications, additional ones are needed in order to specify operations and normal capabilities of the agents. Since we have focused on OC capabilities, the specification can be simplified by ignoring other operations and normal capabilities of the agents.

Specification 3. Based on the sco/c meta-model and principles 1, 2 and 3, the agent can now be defined (Declaration 6) regarding an instance of Regulation, an instance of Expression, a set of active OC capabilities and a set of inactive OC capabilities. The active OC capabilities represent the active/expressed OC capabilities, while the inactive OC capabilities represent the deactivated/repressed OC capabilities. These two sets of OC capabilities have no intersection. In other words, no capability can be both active and inactive at the same time; see Equation 7.

$$\text{Agent}: \text{Regulation} \times \text{Expression} \times \mathbb{P}\text{Capability} \times \mathbb{P}\text{Capability} \quad (6)$$

$$a = (R, E, AP, IP) \text{ where } a \in Agent \wedge R \in Regulation \wedge E \in Expression \wedge P \in \mathbb{P}Capability \wedge IP \in \mathbb{P}Capability \wedge AP \cap IP \quad (7)$$

A few auxiliary functions are needed for simplifying the specification. They are presented in declarations 8 to 11. The disableCap and enableCap functions represent the actions of enabling and disabling capabilities,

respectively. They take a set of capabilities and enable/disable them in an agent. Therefore, they return an agent with new capabilities. The filter function takes sets of pairs of relationship types (mandatory, optional, requires and excludes) and OC capabilities ($\text{reltype} \times \mathbb{P}\text{Capability}$) alongside a relationship type (the second argument of filter in Declaration 10) for filtering and returns the set of OC capabilities whose relationship type is the same as the type determined as the second argument of filter . For instance, this function can assist in extracting the OC capabilities that are mandatory or needed. Declaration 11 denotes a simple auxiliary function which returns all the capabilities (either active or inactive) of an agent.

$$\text{disableCap}: \mathbb{P}\text{Capability} \times \text{Agent} \rightarrow \text{Agent} \quad (8)$$

$$\text{enableCap}: \mathbb{P}\text{Capability} \times \text{Agent} \rightarrow \text{Agent} \quad (9)$$

$$\text{filter}: \mathbb{P}(\text{reltype} \times \text{Capability}) \times \text{reltype} \rightarrow \mathbb{P}\text{Capability} \quad (10)$$

$$\text{capabilitySet}: \text{Agent} \rightarrow \mathbb{P}\text{Capability} \quad (11)$$

Specification 4. The specification of the used election algorithm is in the form of a function (elect in Declaration 12) that returns the set of pairs of agents and the set of OC capabilities ($\mathbb{P}(\text{Agent} \times \mathbb{P}\text{Capability})$) denoting which OC capability or capabilities of each agent must be activated. Regarding Declaration 13, isElected is another auxiliary function related to the elect function that returns the elected OC capabilities ($\mathbb{P}\text{Capability}$) for an agent (Agent) through an election ($\mathbb{P}(\text{Agent} \times \mathbb{P}\text{Capability})$).

$$\text{elect}: \mathbb{P}(\text{Agent} \times \mathbb{P}\text{Capability}) \times \mathbb{P}\text{Capability} \rightarrow \mathbb{P}(\text{Agent} \times \mathbb{P}\text{Capability}) \quad (12)$$

$$\text{isElected}: \mathbb{P}(\text{Agent} \times \mathbb{P}\text{Capability}) \times \text{Agent} \rightarrow \mathbb{P}\text{Capability} \quad (13)$$

Having these functions in place, we are ready to define the self-configuring mechanism in form of function application. To do so, we need to declare the required variables (Declaration 14).

$$\text{c} \in \text{Condition} \\ \text{agents}: \mathbb{P}\text{Agent} == \{a_0, a_1, \dots a_n\} \quad (14)$$

Equations 15 and 16 show a few abbreviations and variable definitions to simplify Specification 5. involvedCaps are the mandatory, optional and required OC capabilities involved in the sco/c architecture. Equation 17 is of particular interest. It shows the candidates of agents (and their capabilities across the system) that can take part in the sco/c architecture according to the regulation and expression (see the definition of involvedCaps in Definition 16). These candidates resulted as follows: for each agent, its OC capabilities (members of $\text{capabilitySet}(a_i)$) that are involved in involvedCaps is obtained. The excluded capabilities are used later for disabling the unnecessary OC capabilities (see Equation 19). The resulting candidates and the set of the involved capabilities are finally given to the elect function which determines the required OC capability or capabilities for activation.

$$\text{allInvolvedCaps} == \text{Expression}(\text{regulate}, (\text{c}, \text{Regulation})) \quad (15)$$

$$\text{involvedCaps} == \text{filter}(\text{allInvolvedCaps}, \text{mandatory}) \cup \text{filter}(\text{allInvolvedCaps}, \text{optional}) \cup \text{filter}(\text{allInvolvedCaps}, \text{required}) \quad (16)$$

$$\text{candidates} == \bigcup_{i=0}^{n} (a_i, \text{involvedCaps} \cap \text{capabiltiySet}(a_i)) \quad (17)$$

$$\text{election} == \text{elect}(\text{candidates}, \text{involvedCaps}) \quad (18)$$

Specification 5. Finally, to specify the self-configuration mechanism, the system is specified in Equation 1, and the self-configuration mechanism is shown in form of a function application. Initially, for each agent, the excluded OC capabilities are disabled, and then, the elected OC capabilities of that agent are enabled. The formed OC system has thus all the OC capabilities required for the selected o/c architecture enabled by the regulation capability of the agents.

$$\text{system} == \{a_0, a_1, \dots a_n\} \\ \text{where } a_i \in \text{Agent} \wedge \\ a_i = \text{enableCap}(\text{isElected}(\text{election}, a_i), \\ \text{disableCap}(\text{filter}(\text{allInvolvedCaps}, \\ \text{excludes}), a_i)) \quad (19)$$

5.2 Verification

In order to verify the self-configuration property of the sco/c architecture, we use LTL model checking. What is important in terms of self-configuration is that the system will be eventually in a state of proper operation [4], which means that: First, an o/c component configuration has been selected. Second, the OC capabilities are successfully identified, and the agents that must activate them are specified via the election algorithm. Third, the activation/deactivation of OC capabilities is done.

Having all the above mentioned conditions, it can be said that “the system is in a valid o/c component configuration”. These phrases can be expressed in the form of an invariant (Formula 20) where system comes from Equation 19, and c comes from Declaration 14.

$$\diamond\, \text{conforms}(\text{system}, \text{Expression}(\text{regualte}(\text{c}, \text{Regulation})) \quad (20)$$

The conforms function is an auxiliary function that checks the validity of the OC system. It uses a condition variable (c) that the sco/c architecture has been selected.

Therefore, the OC capabilities are extracted from the OC system and used for comparison so as to see the conformation to the valid model returned by the regulate function.

It should be noted that the validity of sco/c depends on the rules defined in the regulation. With wrong rules (such as impossible configuration or unreachable conditions), sco/c does not work, and the system cannot be configured. These conditions include applying those rules that employ non-existing OC capabilities or when the needed capabilities cannot be found in the agent and

its neighborhood. Also, when none of the conditions can be evaluated to true, and sco/c cannot self-configure. The same can be said when more than one o/c architecture can be selected. In this condition, the agents will split into two or more groups each trying to achieve a specific o/c architecture. Depending on the OC capability distribution among each group, different outcomes can be expected (such as zero, one or more successful o/c architecture configuration). But, any outcome in this state cannot be accepted, and even, if it works, it will be accidental.

The verification was performed successfully for various possible scenarios using the Maude tool. The target scenarios were divided into two types. The focus of the first type was on situations in which a variant of the o/c architecture could be applied. The goal was to see whether the proper variant was selected and activated in such scenarios. The second type of scenarios included the ones in which no variant were applicable. Also, the verification was performed for impossible and unreachable configurations. In all of the scenarios, the specification of the sco/c architecture proved to be sound and correct.

6. Case Study

In this section we demonstrate the applicability of the sco/c architecture on the example illustrated in subsection 4.2 using our formal specification and verification approach. We will first specify the general form of the problem, meaning the number of tasks and robots is not limited to what has been specified in [5] and [16]. Next we will use this general form to verify the illustrative example.

6.1 General Description

Most of the specifications needed for this example have been already provided. Apart from our intention to present a case study for the application of the sco/c architecture, we also intend to specify normal operations (along with the sco/c architecture specification), resulting in a complete system specification. Specification 6 describes a generic robot in which Agent has already been introduced in Declaration 6. $\mathbb{P}\mathrm{WorkingCapability}$ is used for indicating a set of normal capabilities (sensors and actuators), and $\mathbb{P}\mathrm{Resource}$ for a set of resources that the robot is working on. This set can be $\emptyset$ when there is no resource.

Specification 6. Robot definition.

$$\mathrm{Robot} == (\mathrm{Agent} \times \mathbb{P}\mathrm{WorkingCapability} \times \mathbb{P}\mathrm{Resource}) \quad (21)$$

We defined a simple behavior for each robot based on [6] and then applied the general theme discussed above. The behavior of each robot in our specification consists of three general actions, i.e., acquire, process and release (Specification 7): the resource is acquired, processed and finally released. It is important to note that no action must be done unless the sco/c architecture is formed. This guarantees the formation of the self-configuration phase.

As discussed in the previous section, the $\mathrm{conforms}$ function has the responsibility of checking the conformance between the current formed architecture and the desired one. We use this function as a guard (whose result is used as the Boolean parameter in equations 22 through 24) for all of the actions in our case study.

$\mathbb{P}\mathrm{Resource}$ indicates the set of all available resources; with each acquire operation for a resource ($\mathrm{Resource}$ as the third argument for the $\mathrm{acquire}$ function) or when a resource is completely processed, it is removed from the resource set. Because the definition of Robot has an occurrence $\mathbb{P}\mathrm{Resource}$ that indicates the resource the robot is working on, when this set changes, the Robot needs to change. Therefore, the Robot is considered as both input and output in declaration 22 to 24. By the $\mathrm{process}$ function (Declaration 23), a task is performed on a resource or resources. Finally, when the process is done or a problem happens (e.g., one of the robot tools is broken), the resource is released by the robot (Declaration 24) and added to the resource list.

Specification 7. Simple behavior for the robots.

$$\mathrm{acquire}: \mathrm{Boolean} \times \mathrm{Resource} \times \mathbb{P}\mathrm{Resource} \rightarrow \mathrm{Robot} \times \mathbb{P}\mathrm{Resource} \quad (22)$$

$$\mathrm{process}: \mathrm{Boolean} \times \mathrm{Robot} \rightarrow \mathrm{Robot} \quad (23)$$

$$\mathrm{release}: \mathrm{Boolean} \times \mathrm{Robot} \times \mathbb{P}\mathrm{Resource} \rightarrow \mathrm{Robot} \times \mathbb{P}\mathrm{Resource} \quad (24)$$

After using these behaviors and completing the required definitions (i.e., specifying $\mathrm{WorkingCapability}$, $\mathrm{Regulation}$, $\mathrm{Expression}$, etc.), the LTL formula 20 is verifiable. The next subsection completes these items for the illustrative example and performs the verification.

6.2 Specification and Verification of the Illustrative Example

This subsection presents definitions specific to the illustrative example.

The first step is to define the regulation which consists of two rules ($\mathrm{regulation}_1$ and $\mathrm{regulation}_2$ given below). Based on the computing power of the robots (as the condition), one of the o/c variants can be selected: The OC middleware (mentioned in section 3) is more suitable for higher computational power since it needs an instance of the middleware running on each robot, making it suitable for the decentralized variant, while the RIA controller needs less computational power compared to that of the OC middleware.

$$\mathrm{computingPower}: \mathrm{Boolean} \quad (25)$$

$$\mathrm{regulation}_1 = (\mathrm{computingPower}, \{\mathrm{Information\ Pool\ Manager}, \mathrm{Fact\ Base}, \mathrm{Event\ Manager}, \mathrm{Reflex\ Manager}, \mathrm{High\ Level\ Planner\ Manager}, \mathrm{High\ Level\ Planner}, \mathrm{Actuator}, \mathrm{Low\ Level\ Reflexes}\}) \quad (26)$$

$$\mathrm{regulation}_2 = (\neg \mathrm{computingPower}, \{\mathrm{Invariant\ Monitor}, \quad (27)$$

$\text{Reconfiguration Algorithm}, \text{Result Checker}\})$
$$\text{Regulation} = \{\text{regualtion}_1, \text{regulation}_2\} \quad (28)$$

Definition 25 specifies computingPower as the required condition. This function returns true when the computational power is suitable for running the OC middleware; when the function returns false, it means the RIA controller should be used.

Based on the above specification, the regulate function should be called with computingPower as the first parameter. Due to space limitation, the specification of expression capability has been ignored, but it can be easily extracted from Fig. 4. Declaration 29 shows the specification of Agent used in the robot definition for the illustrative example. As can be seen, the Agent components have been replaced by definitions from this section. Resource has been defined as sequence of WorkingCapability, meaning that each task is represented by the corresponding tool. When each task is performed, the first task in the sequence is removed from Resource. An empty Resource represents a resource on which all the required tasks have been successfully performed.

$$\text{Agent} = (\text{Regulation}, \text{Expression}, \text{regulate}(\text{computingPower}, \text{Regulation})), \text{Regulation} - \text{regulate}(\text{computingPower}, \text{Regulation})) \quad (29)$$
$$\text{WorkingCapability} = \{\text{drill}, \text{insert}, \text{tighten}\} \quad (30)$$
$$\text{Resource} = \text{seq}(\text{WorkingCapability}) \quad (31)$$

Now robots for the illustrative example can be defined using Declaration 32.

$$r_1, r_2, r_3 : \text{Agent} \quad (32)$$

As for the verification, the LTL formula 20 was verified using the Maude tool. Also, verification was performed after the self-configuration phase in order to verify the conforms function as the guard of declarations 22 through 24. The verification phase showed that when the condition of the o/c component configuration changed, the system stopped operation, configuration was chosen, and then the system resumed normal operation. In cases that were designed for impossible operation, as expected, the system stopped all operations.

7. Conclusions and Future Work

In this paper, the sco/c architecture which uses the idea of cell differentiation has been presented in order to achieve self-configuration in the o/c component level. In order to support this idea, an architectural meta-model that considers the OC system as a collection of agents with some capabilities has been proposed. Among these capabilities, there are OC capabilities representing the capabilities that can perform operations related to the o/c architecture. Also, these capabilities are responsible for the self-configuration in the level of architecture itself. The sco/c architecture uses some rules provided by the architect based on the parameters of the system or environment. This architecture is then configured, and finally, the system operates. However, we believe that the rules should be adapted accordingly by considering the prior executions. In biology this notion is called genetic memory [9]. As a future work, we are planning to use the mechanisms related to this notion in order to improve the bio-inspiration and to step closer to living systems. Also, we consider the use of a simple ontology in the OC systems so as to create a semantic base. This can help to create a knowledge-based self-awareness that can assist greatly in cases like the selection of the proper o/c component configuration in the sco/c architecture. This potentially can increase the interoperability between organic systems. This can be especially useful when two or more ubiquitous or pervasive systems are needed to cooperate.

References

[1] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” Computer, vol. 36, no. 1, pp. 41–50, 2003.
[2] C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds., Organic Computing - A Paradigm Shift for Complex Systems. Springer, 2011.
[3] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, and H. Schmeck, “Towards a generic observer/controller architecture for Organic Computing.,” GI Jahrestag. 1, vol. 93, pp. 112–119, 2006.
[4] A. Berns and S. Ghosh, “Dissecting self-* properties,” in Self-Adaptive and Self-Organizing Systems, 2009. SASO’09. Third IEEE International Conference on, 2009, pp. 10–19.
[5] B. Satzger, A. Pietzowski, W. Trumler, and T. Ungerer, “Using automated planning for trusted self-organising organic computing systems,” in Autonomic and Trusted Computing, Springer, 2008, pp. 60–72.
[6] H. Seebach, F. Nafz, J.-P. Steghöfer, and W. Reif, “How to Design and Implement Self-organising Resource-Flow Systems,” in Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 145–161.
[7] M. Salehie and L. Tahvildari, “Self-adaptive software: Landscape and research challenges.,” TAAS, vol. 4, no. 2, Jul. 2009.
[8] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter, “Adaptivity and self-organization in organic computing systems,” ACM Trans Auton Adapt Syst, vol. 5, no. 3, pp. 10:1–10:32, Sep. 2010.
[9] B. Alberts, D. Bray, J. Lewis, M. Raff, K. Roberts, and J. D. Watson, Molecular Biology of the Cell, 4th ed. Garland, 2002.
[10] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study,” Software Engineering Institute, Carnegie Mellon University, CMU/SEI-90-TR-21, 1990.

[11] M. Riebisch, “Towards a more precise definition of feature models,” Model. Var. Object-Oriented Prod. Lines, pp. 64–76, 2003.

[12] K. Lee and K. C. Kang, “Feature dependency analysis for product line component design,” in Software Reuse: Methods, Techniques, and Tools, Springer, 2004, pp. 69–85.

[13] U. Brinkschulte, M. Pacher, and A. Von Renteln, “Towards an artificial hormone system for self-organizing real-time task allocation,” in Software Technologies for Embedded and Ubiquitous Systems, Springer, 2007, pp. 339–347.

[14] M. Roth, J. Schmitt, R. Kiefhaber, F. Kluge, and T. Ungerer, “Organic Computing Middleware for Ubiquitous Environments.,” in Organic Computing, C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Springer, 2011, pp. 339–351.

[15] W. Brockmann, E. Maehle, K.-E. Grosspietsch, N. Rosemann, and B. Jakimovski, “ORCA: An organic robot control architecture,” in Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 385–398.

[16] F. Nafz, J.-P. Steghöfer, H. Seebach, and W. Reif, “Formal modeling and verification of self-* systems based on observer/controller-architectures,” in Assurances for Self-Adaptive Systems, Springer, 2013, pp. 80–111.

[17] V. C. Barbosa, An introduction to distributed algorithms. MIT Press, 1996.

[18] M. Clavel, F. Dur’an, S. Eker, P. Lincoln, N. M. Oliet, J. Meseguer, and C. Talcott, All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic. Springer, 2007.

Ali Tarihi received his BS and MS degrees in Software Engineering from Shahid Beheshti University, Tehran, Iran. He is currently a Ph.D student at the Computer Science and Engineering Faculty, Shahid Beheshti University, Tehran, Iran.

Hassan Haghighi received his BS, MS and PhD degrees in Software Engineering from Sharif University of Technology, Tehran, Iran. He is currently an assistant professor at the Computer Science and Engineering Faculty, Shahid Beheshti University, Tehran, Iran.

Fereidoon Shams Aliee received his BS and MS degrees from Shahid Beheshti University and Sharif University of Technology, respectively. He received his Ph.D in Software Engineering from Department of Computer Science, Manchester University, UK. He is currently an associate professor at the Computer Science and Engineering Faculty, Shahid Beheshti University, Tehran, Iran.