Acta Informatica Pragensia, 2017, 6(2): 124–137
DOI: 10.18267/j.aip.105
Peer-reviewed paper

HACS: A Hybrid Framework for Continuous Flexible and Controlled Architecting

Bentlemsan Khadidja[*], Bennouar Djamel[†], Tamzalit Dalila[‡], Hidouci Khaled Walid[*],

Abstract

Systems like e-voting, e-banking or e-health must offer flexibility to continuously meet technical and legal changing requirements and must at the same time guarantee robustness to respect their security and sensitivity. Component Based Software Engineering (CBSE) and Service Oriented Software Engineering (SOSE) with their modular design represent the most suitable paradigms for those systems. They have strong complementary advantages, despite their similarities, their heterogeneity still hinders systems to benefit from both of them. In this paper, we propose a hybrid framework HACS (Hybrid Approach between Component and Service). HACS proposes to define sensitive systems as a hybrid architecture where the critical parts are controlled according to CBSE coupled to the flexibility and dynamism of SOSE. To address heterogeneity and make possible the substitution between hybrid components, HACS uses a common syntax with semantic annotations based on SAWSDL related to two ontologies; HACS ontology and domain ontology. We illustrate HACS all along the paper through an e-voting case study.

Keywords: CBSE, SOSE, Hybrid Architecture, HACS, E-Voting; Continuous Architecture.

1 Introduction

E-voting is a technological and effective way to modernize the voting process and administer the election (Gibson, Lallet and Raffy, 2008) by enabling voters to cast a secure ballot over the Internet. The main advantage is to simplify the voting process, to increase voters' accessibility, to reduce the error rate and to speed up the report of definite results. E-voting is considered as a high security sensitive system since it plays a decisive role in democratic organizations. Errors are not forgivable, it must offer a free vote and the exact election results. Electronic voting systems worthy of the name (Chondros et al., 2014) must meet the following properties: (1) *Universality*: all eligible voters have the right and ability to cast their votes using the e-voting system. (2) *Equality*: equal access to all eligible voters, they all have the same number of votes, usually one ballot each. (3) *Anonymity*: each voter has the right to cast his vote secretly and no

[*] Laboratoire de la Communication dans les Systèmes Informatiques, Ecole nationale Supérieure d'Informatique (ESI), BP 68M Oued Smar, 16309, El Harrach, 16309, Alegria
✉ k_bentlemsan@esi.dz, w_hidouci@esi.dz

[†] Department of Informatics, Université de Bouira, Rue Drissi Yahia, Bouira 10000, Algeria
✉ djamal.bennouar@univ-bouira.dz

[‡] Laboratory of Digital Sciences of Nantes, Department of Informatics, University of Nantes, Nantes 44300, France
✉ dalila.tamzalit@univ-nantes.fr

one should be able to relate voters to their vote. (4) *Privacy*: impossibly to a party to extract any information about the voter's ballot, or votes not cast by legitimate voters. (5) *Verifiability*: which is of two types: individual verifiability that represents the ability for voters to check that their votes were correctly recorded and that all the votes were processed and counted correctly. (6) *Trust*: eligible voters must trust the system and believe that e-voting principals were met. (7) *Robustness*: be resilient to the faulty behaviour of a system; partial component system malfunction occurs or when it is subject to external malicious attacks.

We propose in this paper a new framework called HACS combing CBSE and SOSE concepts and principals throw e-voting case study. The rest of this paper is organized as follows: in section 2, motivation and related works will be exposed. In section 3, the fundamental concepts of our approach (HACS) will be presented. In section 4 the substitution process will be explained and e-voting case study will be illustrated. Finally, section 5 will conclude the paper.

2 Motivation and related works

There are several e-voting systems proposed in the literature, the software architecture best practices make CBSE or SOSE the most suitable paradigms. Component-based supporters consider modular design, reusability and controllability as the key features to meet e-voting requirements in a homogeneous environment; as an example, Helios (Adida, 2008) based on homomorphic encryption provides universal verifiability and voter privacy. ZEUS (Tsoukalas et al., 2013) extends HELIOS in an open source way to address usability and tallying through a separate computing system. Mosaic (Abdellatif and Adouani, 2014) allows fine-grained control of each task and secure communication between the different system components. Its modularity is used for runtime adaptation and scalability.

Most SOSE voting systems are based on web services. Authors believe that they are the ideal technology to design robust e-voting systems when the internet is the communication platform because of their interoperability. Zurich (Beroggi, 2008) is a robust service-based system, its source code is not available; authors are convinced that attackers with such access could change voting and auditing records. DWSBEV (Omidi and Azgomi, 2009) proposes an architecture based on dependable web services that has been evaluated using stochastic Petri nets and provides security by replicated systems. SOREV (Cooke and Anane, 2012) is a robust FOO92 e-voting where robustness is considered from two perspectives protocol level and system level. SOREV provides privacy, verifiability and integrity.

The CBSE vision of e-voting systems supports the most important e-voting requirements (privacy, verifiability). Its biggest issues are the lack of interoperability and rigidity that directly affects system robustness. The SOSE vision represents the optimal paradigm to ensure robustness. Web services have the ability to create composite services dynamically through automatic and dynamic composition techniques in a heterogeneous environment. They make e-voting systems flexible and fully scalable. According to (Cooke and Anane, 2012) the absence of the state in SOAP/HTTP makes web services more resilient to failure. The alternating connections of web services and the regular flushing of the state that they initiate, make them very suitable for e-voting systems. Unfortunately, there are some issues that, to our knowledge, are not yet addressed:

(1) **Anonymity**: Web services guarantee a robust, flexible and fully scalable architecture. The confidential nature of e-voting must be respected, but there is no evidence that the service provider can't break the voter's anonymity and keeps track of votes.

(2) **Availability**: web services are created and updated on the fly, some web services required by the system may not exist at a precise moment or may be unavailable (timeout) which may lead to serious execution issues.

(3) **Trust***:* Web services are represented by black boxes. We cannot confirm that web services properly execute the required functional code. For crucial parts like security protocols, it may present a favourable point to attack.

(4) **Election rights**: election principles and rights change from a system to another. They present system-specific and confidential data. We think that web services can't be the appropriate implementation technology.

(5) **Controllability**: In SOSE, web services are exposed as black boxes and their functional code cannot be modified. In contrast of CBSE where components can be designed using white boxes or grey boxes allowing software architect to make some changes to keep fine control especially for critical parts of the system.

We propose to design the e-voting system as a hybrid architecture and take full advantages of CBSE and SOSE (Breivold and Larsson, 2007); this architecture must offer the flexibility of SOSE to meet changing requirements and at the same time guarantee controllability of CBSE to respect e-voting security and sensitivity. For that reason, we design all components as architectural elements of SOSE except for confidential and sensitive parts that have to benefit from CBSE controllability, availability and trust. We present in the next section our contribution called HACS.

3 HACS Framework

HACS proposes to meet together advantages of SOSE and CBSE through a hybrid framework. Its main objective is to continuously ensure robustness in security sensitive systems at both design-time and runtime. To achieve this, HACS supports flexibility and dynamism of SOSE and controllability of CBSE.

Starting from the fact that the service in SOSE is a specific component (Erl, 2005). HACS leans on component-based software architecture. Its components can be implemented as architectural elements of CBSE or SOSE. In the rest of paper, to represent SOSE components in HACS we use SOAP-based web services because of their biggest success and growing use. HACS proposes a meta-model defining its architectural elements and a process describing how the substitution is performed at runtime. To represent CBSE, HACS uses proprietary components.

3.1 HACS Meta Model

HACS embodies several concepts, following the component and connector architectural style(Medvidovic and Taylor, 2000) to handle in a uniform way proprietary components and web services. It introduces new concepts as usability value, brother component and neighbour component, Fig.1 exposes the meta-model of HACS and highlights its concepts presented hereafter:

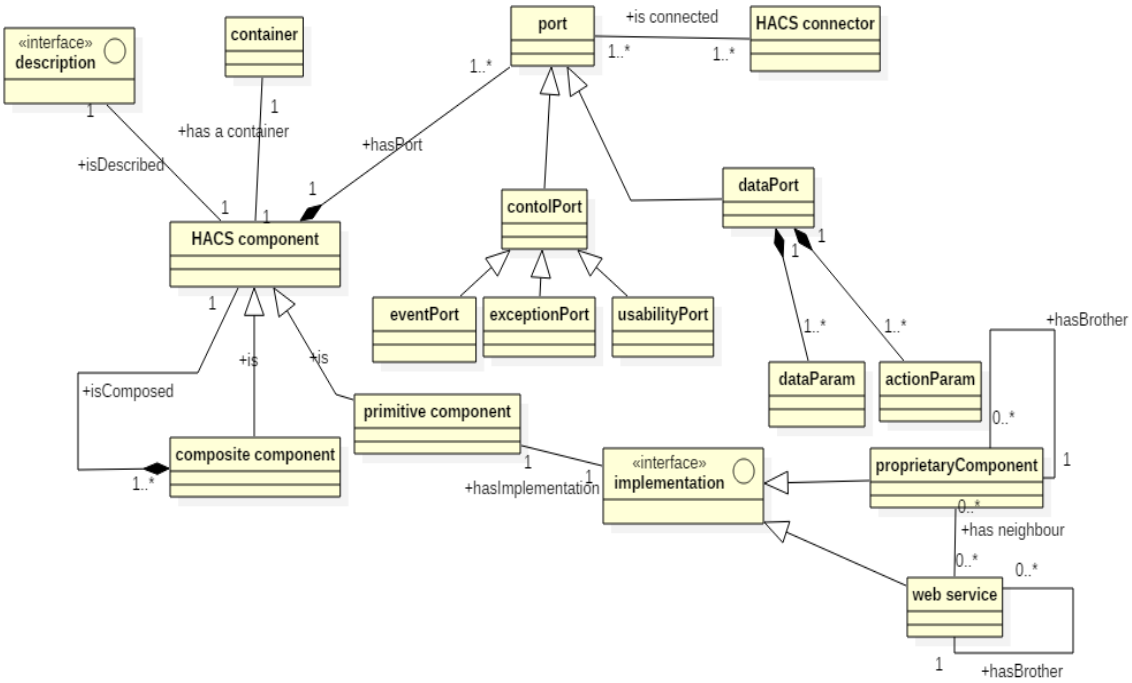

Fig. 1. *HACS Meta Model. Source: Authors.*

- **HACS Component:** is the basic architectural element in HACS, we distinguish two main categories of components; primitive and composite, both share the same external view (Fig. 2). The external view of HACS components is composed of a container and ports. Each HACS component has a usability value.
- **Container:** separates the component from external world and deals with the architectural mismatch between components.
- **Ports:** The expressive power of a component model depends mainly on the quality of its ports (Bennouar, Khammaci and Henni, 2010). In HACS ports are the interaction points between HACS components, they are classified into two types; data ports and control ports.
- **Data ports:** are composed of a set of data Parameters (in, out, in-out) to transmit data and action Parameters: to exchange the service flow between components.
- **Control ports:** are specific ports divided into three types:
  (1) The exception port is to throw an exception, (2) the event port is to expose conditions required to execute a component (3) The usability port contains a float value to express the usability value of a HACS component.

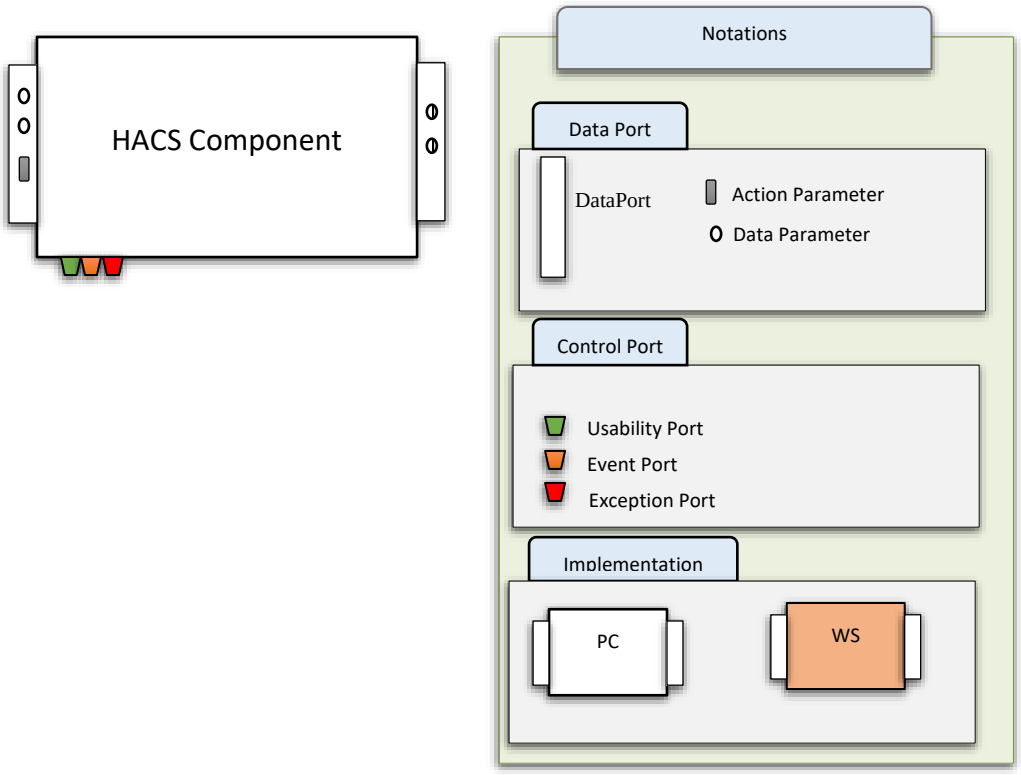

Fig. 2. *HACS concepts notation. Source: Authors.*

- **Usability value:** is a normalized weighted sum calculated from a set of non-functional properties (Zou et al., 2014). Suppose that Q(*c*) is a finite set representing the commonly used quality criteria or non-functional attributes for the HACS component *c*.
  We note Q(*c*) = {$Q_1(c)$, $Q_2(c)$, $Q_3(c)$, $Q_4(c)$, $Q_5(c)$} where:
  - **Availability** $Q_1(c)$: the probability that *c* is accessible.
  - **Reputation** $Q_2(c)$: the measure of trustworthiness of *c*.
  - **Reliability** $Q_3(c)$: the probability of success of *c*.
  - **Cost** $Q_4(c)$: the execution cost of *c*.
  - **Response time** $Q_5(c)$: the time interval between the request and response message of *c*.

  The usability value serves for determining the optimal value representing the quality criteria of a HACS component *c* by maximizing positive non-functional attributes X, such as reputation, availability and reliability and minimizing negative non-functional attributes Y, such as the execution cost and response time.
  The usability value of a primitive component *c* is defined:

$$UV(c) = \sum_{Qi \epsilon X} \frac{Q_i^{max} - Qi\,(c)}{Q_i^{max} - Q_i^{min}} + \sum_{Qi \epsilon Y} \frac{Qi\,(c) - Q_i^{min}}{Q_i^{max} - Q_i^{min}} \qquad Q_i^{max} \neq Q_i^{min} \qquad \textbf{(1)}$$

- **Primitive component:** a primitive component can be presented by a black box, white box or grey box. It can be implemented as a proprietary component for confidential and critical parts or as a web service otherwise. Primitive components can have brother components and neighbour components defined at design time by the software architect and updated at runtime.

- **Composite component:** is the combination of other HACS components either primitives or composites.
- **Brother component:** Two primitive components are said to be brothers, if they offer the same interface, the same functionality and are implemented in the same technology, this means if the "HACS component" is implemented as a proprietary component (respectively web service) all brothers must be implemented as proprietary components (respectively web services).
- **Neighbour component:** Two primitive components are said to be neighbour; if they offer the same interface, the same functionality and are implemented in different technologies, in other words, if the "HACS component" is implemented as a proprietary component (respectively web service) all neighbours must be implemented as web services (respectively proprietary components).
- **HACS connector:** acts as a mediator between HACS components, it is represented by simple interaction such as HTTP/RCP/SOAP that binds two ports. This is important to ensure loose coupling and the flexibility in HACS.
- **Description:** is the interface of a HACS component, its definition is explained in detail in the next section.

3.2 Description of HACS components

To address heterogeneity between proprietary components and web services, they must be described uniformly using the same syntactic elements. This will not only save significant time and reduce costs but allow possible substitution between proprietary components and web services.

As both of CBSE and SOSE are interface based, the description can be done using an existing Interface Description Language (IDL). Web services are already described by WSDL syntax(Chinnici et al., 2007), reusing WSDL to describe proprietary components is very beneficial compared to other IDLs, to avoid the re-description of web services and to help the software architect to quickly analyse the description with a well-known standard. Unfortunately, the interpretation of WSDL file is very ambiguous for the machine because of its lack of semantics. SAWSDL (Semantic Annotation for WSDL) is W3C recommendation extending WSDL by adding semantics to its elements and makes them interpretable by the machine using references to semantic models as ontologies (Kopecký et al., 2007). SAWSDL syntax can be extensible by adding new elements related to ontologies' concepts. We have inspired this idea from (Chabeb and Tata, 2008) to provide a mechanism that ease the automatic discovery, composition and invocation as explained in Fig. 3.

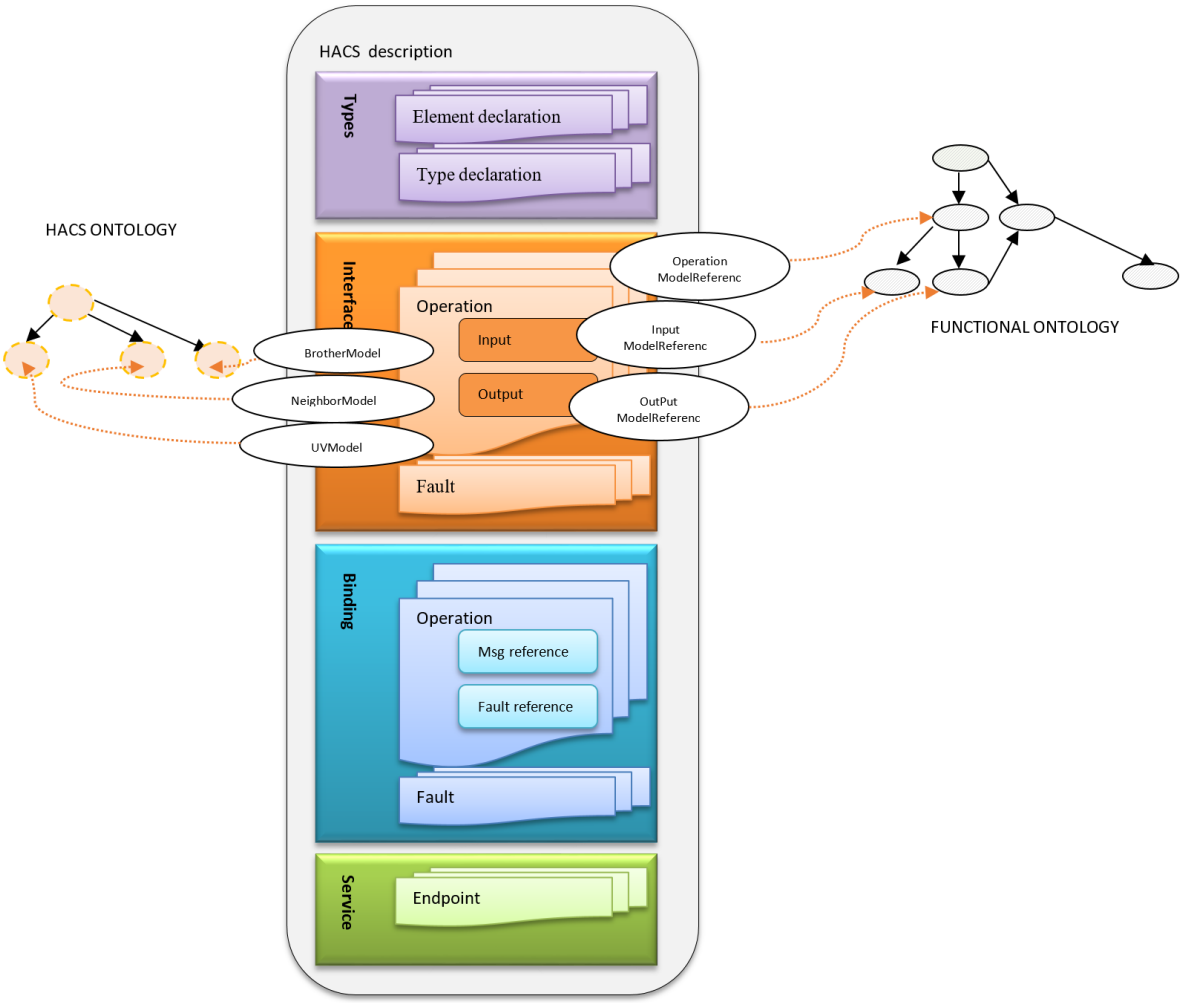

Fig. 3. *The HACS Description schema. Source: Authors.*

The semantic model of HACS uses SAWSDL annotations associated to two ontologies. The first is called HACS ontology, it contains concepts defining semantic of HACS components, their brothers, their neighbours and their non-functional attributes. The second ontology is called functional or domain ontology, it contains the domain-specific semantic describing the system, E-voting in our case study. Three extension attributes to annotate the operation element in HACS description (Fig. 3) are added to XML Schema element declarations and type definitions named brotherModel, neighborModel and uvModel.

```
<xs:attribute name="brotherModel"  type="listOfAnyURI" />
<xs:attribute name="neighborModel" type="listOfAnyURI" />
<xs:attribute name="uvModel"       type="listOfAnyURI" />
…
…
<xs:simpleType name="listOfAnyURI">
<xs:list itemType="xs:anyURI"/>
</xs:simpleType>
```

The input and output data parameters are expressed as a request message and a response message respectively. The main difference between the description of proprietary components and web services lies on data related to web service invocations: for instance, the binding and the port elements, they are represented by empty elements as details in the following listing:

```
<description ...> // description of the proprietary Component
          <types>
        ...// Using built-in data types and they are defined in XMLSchema.
          </types>

          <interface ...>
            <operation ... >
           ...// Here annotation of the proprietary Component
           ...// brother / neighbour / UV
              <input>
                ...// the proprietary Component Request
              </input>
              <output>
                ...// the proprietary Component Response
              </output>
              ...
            </operation>
          </interface>

          <binding />

          <service ...>
             ...// Location of the property component
            <port/> ...// Empty element
            ...
          </service>
</description>
```

3.3 The Runtime substitution process

The substitution process of HACS ensures a rapid and a continuous delivery by introducing brothers, neighbours and dynamic web services searches (see Fig. 4).

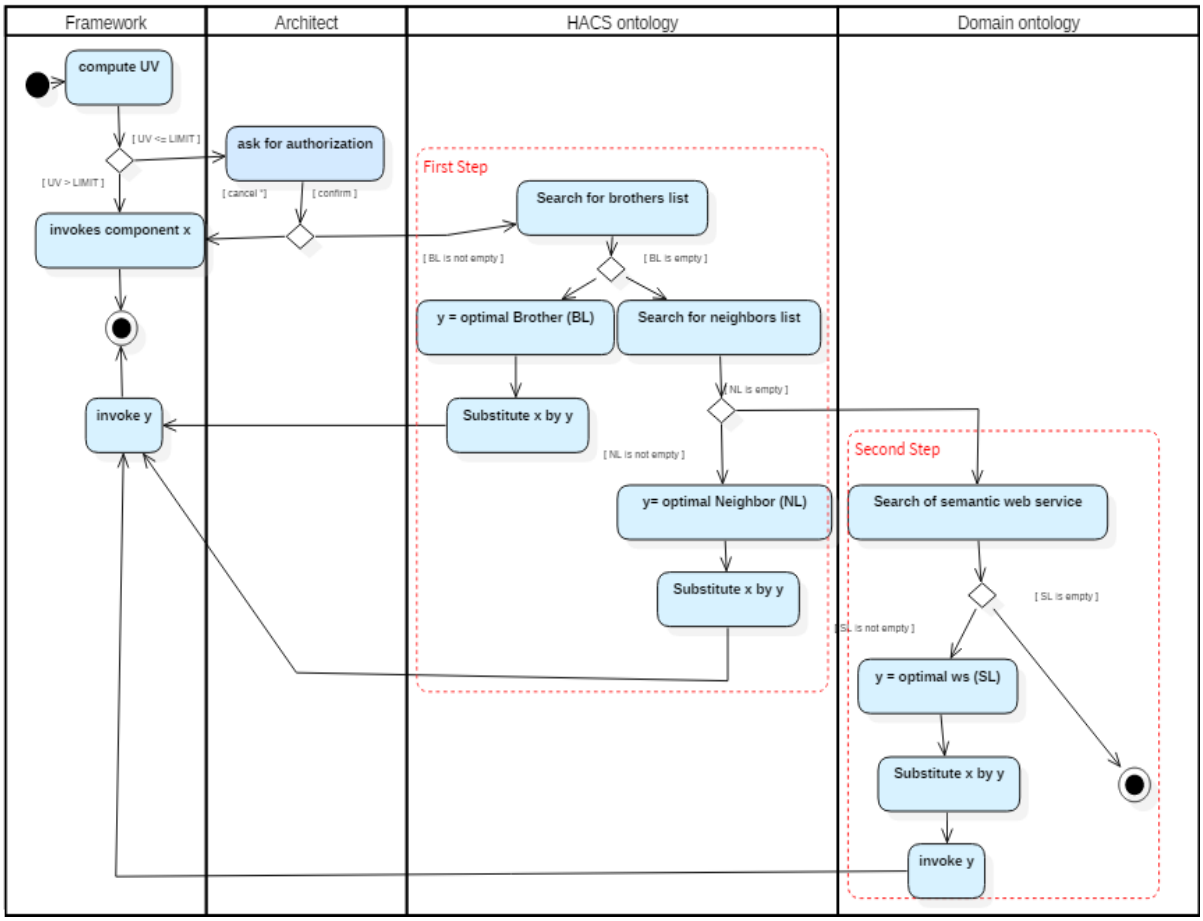

Fig. 4. *The substitution process. Source: Authors.*

To ensure system robustness, HACS framework continuously detects malfunction behaviour before invoking the HACS component by comparing UV to the limit value, if the failure is found, the substitution is launched and is done in two steps:

- **The first step:** HACS Framework based on SAWSDL description, extracts the brother components list (BL) from HACS ontology. The optimal brother with the highest UV is selected and the faulty component is directly substituted by its brother. If any brother is found, HACS searches the optimal neighbour from neighbours component list (NL) and proceeds the same way as brother's search. This first step is introduced in HACS in order to enhance rapid delivery. When no component in BL or NL is found, the second step is launched.
- **The second step:** SAWSDL annotations help to find similar web service from Domain ontology according to the following sequence:
    - *Discovery:* finds the list of similar web services (SL) that offer the same functionality as the faulty component from domain ontology.
    - *Selection:* once (SL) discovered, the optimal web service (WS*) having the highest usability value is selected from SL.
    - *Substitution*: the faulty component is replaced by WS* and deleted from HACS ontology, then WS* is added with its existing brothers as new concepts in HACS ontology to reduce the execution time of future substitutions. Brothers of the optimal web service are the remaining web services from the candidate list {SL-(WS*)}.

4 E-voting in HACS

To secure the voting process from various attacks, e-voting protocols have been proposed. One of the well-known and proved protocols is FOO'92 based on blind signature, it evolves voter, administrator and counter in three phases: registration, casting and tallying. the protocol schema is explained in details in (Fujioka, Okamoto and Ohta 1992). Our case study is based on FOO'92 Protocol. EVSH (E-Voting System in HACS) is seen as a hybrid application, the software architect defines at design time its architecture composed of two proprietary components to expose critical and very sensitive parts: "FOO92 component" and "the elections rights component", as well as web services components like "Authentication component", "Voting component", "Duplication component", the role of each component is explained below:

1. The FOO 92 component: is a composite HACS component to secure EVHS according to the FOO 92 schema composed of four primitive proprietary components (see Fig. 5):
   - FOO Voter: In registration phase, the voter prepares a ballot, encrypts it, signs it then sends it to the administrator. in casting phase, removes the blinding encryption layer revealing an encrypted ballot signed by the administrator then sends the resultant signed-encrypted ballot the counter and finally in tallying phase, verifies that their ballots are on the list and sends the counter the decryption keys.
   - FOO Administrator: verifies in registration phase that the signature belongs to an eligible voter who has not yet voted, then signs the valid ballot and returns it to the voter.
   - FOO Counter: checks in casting phase the signature on the encrypted ballot. If the ballot is valid, the counter puts it on a temporary list that is published after closing the vote. In tallying phase; FOO counter uses voter keys to decrypt the ballots and add the votes to the election tally.
   - FOO Taller: publishes the voting results so that voters can verify their votes.
2. Election rights component: defines the legal framework of the e-voting system.
3. Authentication component: gives access to eligible voters after verifying their name, id and fingerprint.
4. Voting component: specifies the voting user interface to cast votes.
5. Duplication component: replicates the voting results on multiple servers for safety.

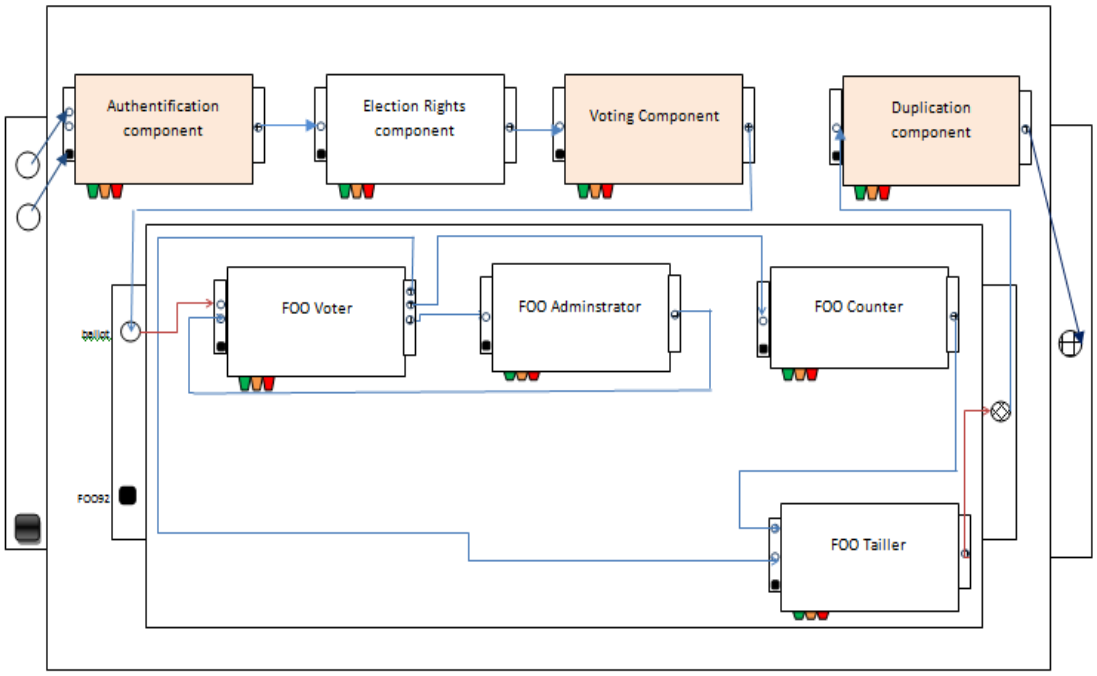

Fig. 5. *E-voting process in HACS. Source: Authors.*

The software architect defines an acceptable range of values for each quality criteria (Tab. 1). From which the limit value is calculated, in this example the Limit ="1.1", Then he assigns eventual brothers and neighbours of all primitive components in EVHS.

We suppose that UV (Authentication component) =1, it is therefore a faulty component because *UV < limit value*. The substitution process is launched to replace it with the most suitable component.

	Non Functional Attributes to be Minimized		**Non Functional Attributes to be Maximized**			
	Cost [1,200] *€*	**Response time** [1,1000] *ms*	**Availability** [0.6,1] %	**Reputation** [0.4,1] *%*	**Reliability** [0.6,1] *%*	***UV***
Auth1	180	750	0.8	0.6	0.8	1.68
Auth2	150	800	0.7	0.5	0.8	1.2

Tab. 1. *Usability values of Auth1 and Auth2. Source: Authors.*

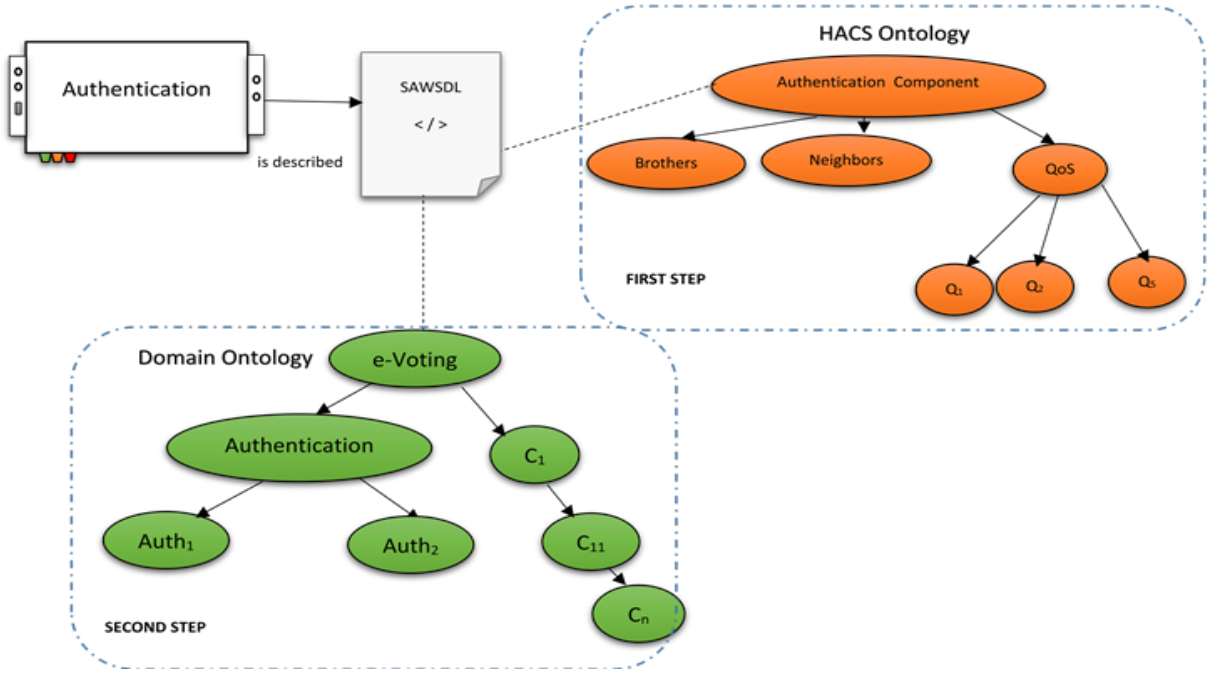

Fig. 6. *HACS semantic model. Source: Authors.*

The substitution process (see Fig. 6) go throw the following steps after extracting information from the SAWSDL description file of "Authentication component":

First step

1- Searching brothers from HACS ontology.

RESULT: No brother was found BL= {}.

2- Searching neighbours from HACS ontology.

RESULT: No neighbour was found NL= {}.

Second step

3- Searching candidate list from Domain ontology (Fig.6).

RESULT: SL = ($Auth_1$, $Auth_2$).

4- Selecting the optimal web service.

RESULT: WS* = $Auth_1$.

$Auth_1$ is the optimal UV (WS^*) = 1.68, "Authentication component" is substituted by $Auth_1$. $Auth_2$ is a web service offering the same functionality of $Auth_1$, so $Auth_2$ is a brother component of $Auth_1$. The HACS ontology is updated by adding $Auth_1$ with its brother $Auth_2$, then removing "Authentication component" from HACS ontology. If $Auth_1$ falls $Auth_2$ will be found in future invocation at the first step of the substitution process.

4.1 Discussion

Our contribution successfully meets the e-voting requirements and offers controllability and flexibility. Universality is guaranteed by "Authentication component". Anonymity, privacy and verifiability are ensured by "FOO'92 component". Equality is verified through "Election rights component". The use of proprietary components in critical parts and the duplication of results increase trust and finally the substitution process strengthen the system robustness at runtime.

5 Conclusion

In this paper, we presented HACS, a framework to build hybrid and continuous architecture mixing the power of CBSE and SOSE. We have shown through the EVSH how the requirements of e-voting are fulfilled. To deals with the heterogeneity in HACS we have proposed to use a SAWSDL common syntax related to two ontologies; domain ontology to describe the domain of the system and HACS ontology to describe non-functional concepts. Currently, HACS deals with an existing architecture defined at design time by the software architect and updated continuously at runtime. An interesting direction of our future work is to define architecture from the scratch. Furthermore, we have proposed to describe HACS components using IDLs, this reveals insufficiency to describe the structure of complex components, we think that an architecture description language will be more appreciated. Additionally, more careful work is needed to define the web service discovery algorithm by adopting recent propositions as AI planning graph.

References

Abdellatif, T. & Adouani, A. (2014). Mosaic: a secure and practical remote voting system. *International Journal of Autonomic Computing*, 2(1), 1-20. doi: 10.1504/IJAC.2014.059109

Adida, B. 2008. Helios: Web-based Open-Audit Voting. In *Proceedings of the 17th conference on Security symposium* (pp. 335-348). Berkeley: USENIX Association.

Bennouar, D., Khammaci, T. & Henni, A. (2010). A new approach for component's port modeling in software architecture. *Journal of Systems and Software*, 83(8), 1430-1442. doi: 10.1016/j.jss.2010.03.005

Beroggi, G. E. G. (2008). Secure and easy internet voting. *Computer*, 41(2), 52-56. doi: 10.1109/MC.2008.60

Breivold, H. P. & Larsson, M. (2007). Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. In *Proceedings of the 33rd EUROMICRO Conference on Software Engineering and Advanced Applications* (pp. 13-20). New York: IEEE. doi: 10.1109/EUROMICRO.2007.25

Chabeb, Y. & Tata, S. (2008). Yet another semantic annotation for WSDL. In *IADIS International Conference on WWW/Internet* (pp.437-441). Retrieved from https://hal.archives-ouvertes.fr/hal-01380984/document

Chinnici, R., Moreau, J.-J., Ryman, A. & Weerawarana, S. (2007). *Web services description language (wsdl) version 2.0 part 1: Core language*. Retrieved from https://www.w3.org/TR/wsdl20/wsdl20.pdf

Chondros, N., et al. (2014). Electronic Voting Systems – From Theory to Implementation. In *E-Democracy, Security, Privacy and Trust in a Digital World. e-Democracy 2013. Communications in Computer and Information Science, vol 441* (pp. 113-122). Cham: Springer. doi: 10.1007/978-3-319-11710-2_11

Cooke, R. & Anane, R. (2012). A service-oriented architecture for robust e-voting. *Service Oriented Computing and Applications*, 6(3), 249-266. doi: 10.1007/s11761-012-0108-0

Erl, T. (2005). *Service-oriented architecture: concepts, technology, and design*. India: Pearson Education.

Fujioka, A., Okamoto, T. & Ohta, K. (1992). A practical secret voting scheme for large scale elections. In *International Workshop on the Theory and Application of Cryptographic Techniques*, (pp. 244-251). Berlin: Springer. doi: 10.1007/3-540-57220-1_66

Gibson, J. P., Lallet, E. & Raffy, J.-L. (2008). Analysis of a Distributed e-Voting System Architecture against Quality of Service Requirements. In *The Third International Conference on Software Engineering Advances, ICSEA'08* (pp. 58-64). New York: IEEE. doi: 10.1109/ICSEA.2008.18

Kopecký, J., Vitvar, T., Bournez, C. & Farrell, J. (2007). SAWSDL: Semantic Annotations for WSDL and XML Schema. *IEEE Internet Computing*, 11(6), 60-67. doi: 10.1109/MIC.2007.134

Medvidovic, N. & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. IEEE Transactions on software engineering, 26(1), 70-93. doi: 10.1109/32.825767

Omidi, A. & Azgomi M. A. (2009). An architecture for e-voting systems based on dependable web services. In *International Conference on Innovations in Information Technology, IIT'09* (pp. 200-204). New York: IEEE. doi: 10.1109/IIT.2009.5413640

Tsoukalas, G., Papadimitriou, K., Louridas, P. & Tsanakas, P. (2013). From helios to zeus. In Presented as part of the 2013 Electronic Voting Technology Workshop and Workshop on Trustworthy Elections. Retrieved from http://esdep.web.auth.gr/wp-content/uploads/2014/06/from_helios_to_zeus.pdf

Zou, G., Lu, Q., Chen, Y., Huang, R., Xu, Y. & Xiang, Y. (2014). QoS-Aware Dynamic Composition of Web Services Using Numerical Temporal Planning. *IEEE Transactions on Services Computing*, 7(1), 18-31. doi: 10.1109/TSC.2012.27