Towards Access Control Models for Conversational User Interfaces*

Elena Planas[1], Salvador Martínez[2], Marco Brambilla[3], and Jordi Cabot[1,4]

[1] Universitat Oberta de Catalunya, Spain eplanash@uoc.edu,
[2] IMT Atlantique, France salvador.martinez@imt-atlantique.fr,
[3] Politecnico di Milano, Italy marco.brambilla@polimi.it,
[4] ICREA, Spain jordi.cabot@icrea.cat

Abstract. Conversational User Interfaces (CUIs), such as chatbots, are becoming a common component of many software systems and they are evolving in many directions (including advanced features, often powered by AI-based components). However, less attention has been paid to their security aspects, such as access-control, which may pose a clear risk. In this paper, we apply Model-Driven techniques to define more secure CUIs. In particular, we propose a framework to integrate an Access-Control protocol into the CUI specification and implementation through a set of policy rules described using a Domain-Specific Language (DSL) integrated with the core CUI language.

1 Introduction

Nowadays, user interfaces that allow fluid and natural communication between humans and machines are gaining popularity [11]. Many of these interfaces, commonly referred as Conversational User Interfaces (CUIs), are becoming complex software artifacts themselves, for instance, through AI-enhanced software components that enable the adoption of Natural Language Processing (NLP) features.

CUIs are being increasingly adopted in various domains such as e-commerce, customer service, eHealth or to support internal enterprise processes, among others. Many of these scenarios are susceptible to arise security risks of both the user and the system. For instance, we may need to add security when we need:

- To disable potential queries depending on the user (e.g. a bot for a Human Resource Intranet must be careful not to disclose private data, such as salaries, unless the request comes from an authorized person).
- To execute different behaviours depending on the user. For instance, a CUI embedded into an e-learning system will provide different answers depending on the user who queries the marks (teacher or student).
- To provide different information precision for the same query depending on the user privileges. For instance, a weather or financial CUI may provide a more detailed answer to paying users.

* This work has been partially funded by the Spanish government (LOCOSS project - PID2020-114615RB-I00 and BODI project - PDC2021-121404-I00).

Several works [7, 12, 16] emphasize the importance of considering security, and especially access-control as highlighted in the above scenarios, in the CUI definition though no concrete solution is proposed.

In this line, this work proposes to enrich CUI definitions with access-control primitives to enable the definition of more secure CUIs. Our solution is based on the use of model-driven techniques to raise the abstraction level at which the CUIs (and the access-control extensions) are defined. This facilitates the generation of such secure CUIs on top of different development platforms. In particular, we extend our generic CUI language [15] with new access-control modeling primitives adapted to the CUI domain and show how this extended models can be enforced as part of a policy evaluation component. As an example, we discuss such implementation on top of the Xatkit open source framework [6].

The rest of the paper is structured as follows: Section 2 provides the background about CUIs and access-control; Section 3 describes the framework we propose to provide model-based access-control for CUIs; Section 4 summarizes the related work; and finally Section 5 concludes.

2 Background

Conversational User Interfaces (CUIs) aim to emulate a conversation with a real human. The most relevant examples of CUIs are the chat*bots* and voice*bots*. A bot wraps a CUI as a key component but complements it with a behavior specification that defines how the bot should react to a given user request. The conversation capabilities of a bot are usually designed as a set of *intents*, where each intent represents a possible user's goal. The bot awaits for its CUI front-end to match the user's input text (called *utterance*) with one of the intents the bot implements. The matching phase may rely on external Intent Recognition Providers (e.g. DialogFlow, Amazon Lex, Watson Assistant). When there is a match, the bot back-end executes the required behaviour, optionally calling external services; and finally, the bot produces a response that it is returned to the user. For non-trivial bots, the behaviour is modeled using a kind of state-machine expressing the valid interaction flows between the users and the bot.

Access-control [18] is a mechanism aimed at assuring that the resources within a given software system are available only to authorized parties, thus granting *Confidentiality* and *Integrity* properties on resources. Basically, access-control consists of assigning *subjects* (e.g., system users) the *permission* to perform *actions* (e.g., read, write, connect) on *resources* (e.g., files, services). Access-control policies are a pervasive mechanism in current information systems, and may be specified according to many different models and languages, such as Mandatory Access-Control (MAC) [2], Discretionary Access-Control (DAC) [2], Attribute-Based Access-Control [9], and Role-based Access-Control (RBAC) [17]. In this work we focus on RBAC, where permissions are not directly assigned to users (which would be time-consuming and error-prone in large systems with many users), but granted to roles. Then, users are assigned to one or more roles, thus acquiring the respective permissions. To ease the administration of RBAC security policies, roles may be organized in hierarchies where permissions are inherited and possibly added to the more specific roles.

3 Access-control framework for CUIs

Fig. 1 summarizes our framework to integrate access-control on CUIs, consisting of: (1) a **design time component** (*RBAC Policy rules* in the figure) to enable the specification of the bot authorization policy (see Sec. 3.1); and (2) a **runtime component** (*PEP* and *PDP* in the figure) in charge of evaluating and enforcing that policy upon the resource's access from users (see Sec. 3.2).

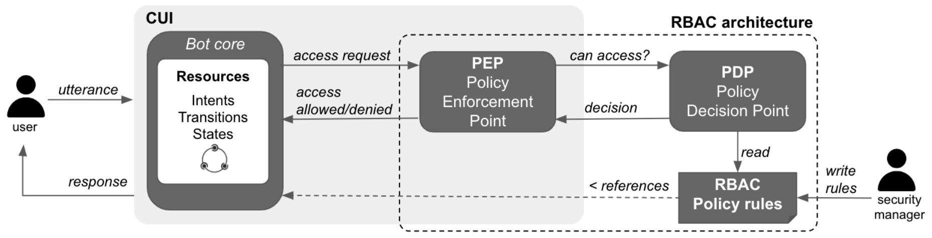

Fig. 1. Framework overview.

3.1 Policy specification

The authorization policy is expressed via a policy language. To this end, in this paper we propose to extend a generic CUI language [15] with new modeling primitives adding access-control semantics to CUIs. As any DSL, this extended *access-control-CUI* DSL is defined through two main components [10]: (i) an *abstract syntax* (metamodel) which specifies the language concepts and their relationships, and (ii) a *concrete syntax* which provides a specific (textual or graphical) representation to specify models conforming to the abstract syntax.

Fig. 2 depicts our proposal for the language metamodel, combining all the RBAC basic concepts with CUIs specific elements. In the following, we detail its main concepts.

CUI metamodel. The CUI-specific metamodel part (coloured in grey in Fig. 2) is a simplified version from the metamodel previously defined by the authors in [15] and describes the set of concepts used for modeling the intent definitions of a bot and its execution logic. The main elements of this metamodel are:

Intents. The metaclass *Intent* represents the possible user's goals when interacting with the CUI. Intents, which are a specific type of *Event* (as bot

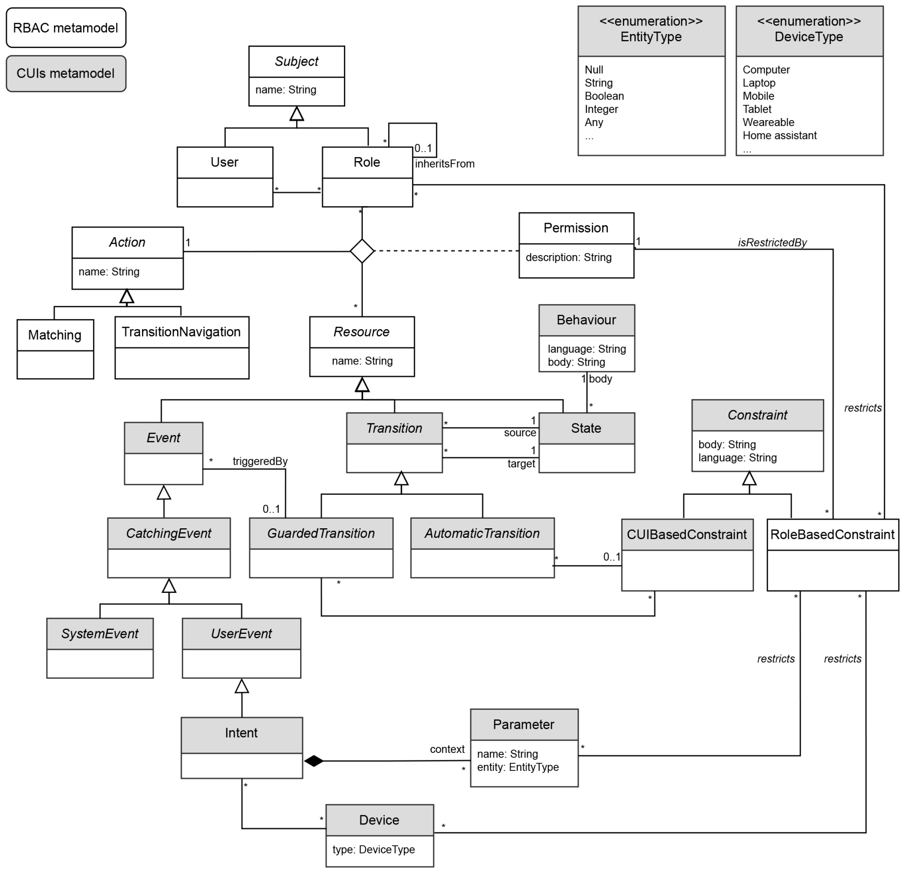

Fig. 2. Access-control CUIs metamodel.

interactions can also be triggered by external events), can optionally have *Parameter*s which allow defining specific characteristics of the *Intent*. On the other hand, intents can be triggered using several devices.

States. Following the state-machine formalism, this metaclass models a particular behavioral state in which the bot stays until a new intent triggers a transition to another state.

Transitions. The metaclass *Transition* represents the potential bot changes from one state to another. We distinguish two types of *Transition*s: *AutomaticTransitions* (triggered automatically) and *GuardedTransitions* (triggered when a specific guard holds). A *GuardedTransition* may be triggered by one or more *Event*s and include a *Constraint* to be satisfied for the transition to occur. This allows a fine-grained control over the firing of the *Transition*.

RBAC metamodel. The RBAC metamodel part is an extended version of the RBAC standard mentioned in Section 2 to adapt it to CUIs. This is done through the definition of a set of permissions which specify which roles are allowed to perform a specific action (a match to an intent or a transition navigation to a state) on a resource (intent, transition, or state). Its main elements are:

Resources. The metaclass *Resource* represents the objects that can be accessed within the CUI and that we may want to protect. In the context of CUIs, resources are basically of three types: *Intent*s, *Transition*s, and *State*s. Protecting intents will allow hiding part of the CUI's intents to specific roles. This may be necessary, for instance, to prevent specific users from accessing some intents. On the other hand, protecting transitions and states will allow, once an intent has been matched, to execute different behaviors depending on the role who triggered the intent. This may be useful, for instance, to provide different answers for an intent depending on the role of the user.

Subjects. The metaclass *Subject* represents the active entities which interact with the CUI. Following a RBAC approach, we define two kinds of subjects: *User*s and *Role*s, where users get roles assigned and role inheritance is supported.

Actions. The metaclass *Action* represents the access to the resources that may be performed by the subjects of the CUI. In this context, we consider the possible actions performed by subjects are *Matching*s (to an intent) and *TransitionNavigation*s (to a state of the state machine). The latter enables a more fine-grained control to the potential user interaction when needed.

Permissions. The metaclass *Permission* represents the right to perform a given *action* (a match or a transition navigation) on a given *resource* (an intent, transition or state) granted to a specific *role* (corresponding to a CUI user).

Constraints. The metaclass *Constraint* restricts the permission to execute the corresponding action only when certain conditions hold. The metaclass *RoleBasedConstraint*, which extends the original RBAC standard model combining a concept from the ABAC model, represents specific context-based constraints (such as geographic location or the used device) to restrict the permissions.

Concrete syntax. In order to complete the definition of our DSL, we could provide a textual concrete syntax, a graphical one or a combination. We show an example of a textual syntax in Section 3.3.

3.2 Policy evaluation and enforcement

Given an RBAC policy, our framework needs to combine a number of runtime components to enforce it. The recommendation in the implementation of modern policy frameworks is separating the infrastructure logic from the application logic by using a reference monitor architecture [1]. This architecture consists in two basic components: a **Policy Enforcement Point** (PEP) and a **Policy Decision Point** (PDP). Every access action requested by an user is intercepted by the PEP that, in turn, forwards it to the PDP to yield an access decision.

Our framework follows this architecture. As Fig. 1 shows, access requests to the bot resources (intents, transitions and states) are intercepted. These requests are then forwarded to the PDP, which reads the policy rules to resolve the access. The access decision yielded by the PDP is returned to the bot through the PEP.

3.3 Proof of concept

In order to show the feasibility of our approach, we discuss in this section a prototype implementation of our framework on top of the Xatkit framework [6] and illustrate it with a simple weather chatbot example[5].

We have first added the textual concrete syntax extensions needed to model the new CUI metaclasses as part of a Xatkit specification. Listing 1.1 shows a snippet of the authorization policy specification which describes that the intent *Get Historical Weather* can be only matched by registered users. To simplify the definition of more complex CUIs we could provide default permission configurations (i.e. enabling or disabling access unless explicitly stated otherwise) and define `GRANT ALL` level permissions (i.e. authorization to match all available intents). Proper parsing of these textual *syntactic sugar shortcuts* would be translated into equivalent metamodel instantiations.

```
Permission p1 (
    Role unregisteredUser
    Resource GetHistoricalWeatherIntent
    Action matching
) -> Deny
Permission p2 (
    Role registeredUser
    Resource GetHistoricalWeatherIntent
    Action matching
) -> Allow
```

Listing 1.1. Policy example for a weather chatbot.

We have then implemented the policy evaluation and enforcement. There are several possible strategies to this end, also depending on whether the chatbot designer has internal access to the chatbot engine.

When modifying the execution logic of the chatbot engine is possible, we could embed the security checks as part of the engine itself. These checks would be added as standard elements of the chatbot execution logic and be implicitly verified upon every single intent matching or transition navigation request. But in most scenarios, chatbot designers will not have this option as most chatbot platforms are not open source or are *hidden* behind an API offered to deploy the bot and interact with the engine. In these cases, access-control must be explicitly added to the individual chatbot logic. Authorization verification becomes now explicit but, on the other hand, it can be easily added on top of many more chatbot engines.

This is the strategy shown in Listing 1.2. In this example, we show how the transition from an initial *Awaiting input* state to the *Print historical weather* state will only be triggered when the user utterance matches the *Get Historical Weather* intent above and the user is authorized to match such intent.

[5] https://github.com/elenaplanas/xatkit-RBACBot

```
awaitingInput
 .when(intentIs(GetHistoricalWeatherIntent)
 .and(c -> policyRules.checkPermission(user.getRole().getName(),
 "matching","GetHistoricalWeatherIntent"))).moveTo(printHistoricalWeather)
```

Listing 1.2. Policy Enforcement Point (PEP) implementation.

Note that, even if access-control evaluation and enforcement is now explicit, it could still be automatically added to the concerned transitions. Given a security policy such as the one in Listing 1.1 and a plain chatbot definition, we could automatically instrument all relevant transitions with the proper access-control checks based on the policy definition.

4 Related Work

Several authors have expressed the need to secure chatbots, especially in critical domains such as banking [12] or health [16]. In the same line, [7] even proposes chatbot providers to attach an SLA to their chatbots, including security aspects. Indeed, as pointed out in [4, 5, 8, 19], chatbots are concerned by (and should be tested against) a number of security concerns. While these works highlight the need to integrate security aspects, they do not propose concrete and actionable solutions. Even for industrial tools (like DialogFlow, Amazon Lex or Watson Assistant) access-control is focused on the management of the permissions to collaborate in the bot definition. At most, you can also define who can execute the bot, with no further fine-grained permission levels.

This limitation is shared by proposals focusing on chatbot definition languages, such as [6, 14, 15], which do not include modeling primitives to define the access-control policies even if modeling of access-control policies is a subject with a long tradition in the MDE community [3], with some notable examples like SecureUML [13] which extends UML with an RBAC metamodel that serves as inspiration for our own proposal. To sum up, we believe ours is the first approach to integrate access-control as first-class citizen in a bot definition language.

5 Conclusions

In this paper we have proposed a new model-driven framework for enhancing the security of CUIs by integrating and adapting the semantics of the Role Based Access-Control (RBAC) protocol to Conversational User Interfaces (CUIs). In particular, we have extended a generic CUI metamodel with RBAC primitives that enable the definition of fine-grained access control policies for all key CUI elements (such as intents, states and transitions). We also provided a preliminary proof of concept to demonstrate the feasibility of our approach.

As further work we plan to enrich the framework with other access-control models and improve the validation and tool support of the approach. Moreover, we see this work is a first step towards the modeling of other security-related aspects for CUIs, such as DDoS, privacy, encryption, and so on.

References

1. Information technology - Open Systems Interconnection - Security frameworks for open systems: Access control framework (ISO-10181-3/X.812) (1996)
2. 5200.28-STD, D.: Trusted Computer System Evaluation Criteria. Dod Computer Security Center (1985)
3. Basin, D., Clavel, M., Egea, M.: A decade of model-driven security. In: Proc. of the 16th ACM symposium on Access control models and technologies. pp. 1–10 (2011)
4. Bozic, J., Wotawa, F.: Security testing for chatbots. In: Testing Software and Systems (2018)
5. Cabot, J., Burgueño, L., Clarisó, R., Daniel, G., Perianez-Pascual, J., Rodríguez-Echeverría, R.: Testing challenges for nlp-intensive bots. In: 3rd IEEE/ACM Int. Workshop on Bots in Software Engineering. IEEE (2021)
6. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: A multimodal low-code chatbot development framework. IEEE Access **8** (2020)
7. Gondaliya, K., Butakov, S., Zavarsky, P.: SLA as a mechanism to manage risks related to chatbot services. In: 2020 IEEE 6th Int. Conference on Big Data Security on Cloud (BigDataSecurity) (2020)
8. Hasal, M., Nowaková, J., Ahmed Saghair, K., Abdulla, H., Snášel, V., Ogiela, L.: Chatbots: Security, privacy, data protection, and social aspects. Concurrency and Computation: Practice and Experience **33**(19) (2021)
9. Hu, V.C., Ferraiolo, D., Kuhn, R., Friedman, A.R., Lang, A.J., Cogdell, M.M., Schnitzer, A., Sandlin, K., Miller, R., Scarfone, K., et al.: Guide to attribute based access control (abac) definition and considerations (draft). NIST special publication **800**(162) (2013)
10. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Pearson Education (2008)
11. Klopfenstein, L.C., Delpriori, S., Malatini, S., Bogliolo, A.: The rise of bots: A survey of conversational interfaces, patterns, and paradigms. In: Conference on Designing Interactive Systems,. ACM (2017)
12. Lai, S.T., Leu, F.Y., Lin, J.W.: A banking chatbot security control procedure for protecting user data security and privacy. In: Advances on Broadband and Wireless Computing, Communication and Applications (2019)
13. Lodderstedt, T., Basin, D., Doser, J.: Secureuml: A uml-based modeling language for model-driven security. In: International Conference on the Unified Modeling Language. pp. 426–441. Springer (2002)
14. Pérez-Soler, S., Guerra, E., de Lara, J.: Model-driven chatbot development. In: Conceptual Modeling (2020)
15. Planas, E., Daniel, G., Brambilla, M., Cabot, J.: Towards a model-driven approach for multiexperience AI-based user interfaces. Soft. and Syst. Modeling **20**(4) (2021)
16. Roca, S., Sancho, J., García, J., Álvaro Alesanco: Microservice chatbot architecture for chronic patient support. Journal of Biomedical Informatics **102** (2020)
17. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST Model for Role-Based Access Control: Towards a Unified Standard. In: RBAC'00. ACM (2000)
18. Sandhu, R.S., Samarati, P.: Access Control: Principle and Practice. Communications Magazine, IEEE **32**(9) (1994)
19. Ye, W., Li, Q.: Chatbot security and privacy in the age of personal assistants. In: 2020 IEEE/ACM Symposium on Edge Computing (SEC) (2020)