Software and Systems Modeling
https://doi.org/10.1007/s10270-023-01131-3

SPECIAL SECTION PAPER

Modeling and enforcing access control policies in conversational user interfaces

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

Received: 15 November 2022 / Accepted: 2 October 2023

Abstract
Conversational user interfaces (CUIs), such as chatbots, are becoming a common component of many software systems. Although they are evolving in many directions (such as advanced language processing features, thanks to new AI-based developments), less attention has been paid to access control and other security concerns associated with CUIs, which may pose a clear risk to the systems they interface with. In this paper, we apply model-driven techniques to model and enforce access-control policies in CUIs. In particular, we present a fully fledged framework to integrate the role-based access-control (RBAC) protocol into CUIs by: (1) modeling a set of access-control rules to specify permissions over the bot resources using a domain-specific language that tailors core RBAC concepts to the CUI domain; and (2) describing a mechanism to show the feasibility of automatically generating the infrastructure to evaluate and enforce the modeled access control policies at runtime.

Keywords Model-driven engineering · Conversational user interfaces · CUIs · Access-control · RBAC

1 Introduction

Nowadays, user interfaces that allow fluid and natural communication between humans and machines are gaining popularity [28]. Many of these interfaces, commonly referred to as Conversational User Interfaces (CUIs), are becoming complex software artifacts themselves, for instance, through AI-enhanced software components that enable even more natural interactions with the possibility to use advanced Natural Language Processing (NLP) components embedded in chatbots or voicebots.

CUIs are being increasingly adopted in several domains such as ecommerce, customer service[1] (as a direct communication channel between the company and end-users) [34], eHealth (to automatize healthcare [3]) or to support internal enterprise processes, among others.[2] However, many of these domains are susceptible to trigger access-control risks. For instance, the following scenarios may cause several risky situations: (1) A bot for a Human Resource Intranet could disclose private data, such as salaries, to an unauthorized person; (2) A CUI embedded into an eLearning system could provide the same information for both teachers and students, although they have different roles and different needs; or (3) A CUI acting as the interface to a paying service could provide the same enriched answers to the non-paying users and the paying ones. To avoid the above situations, we need to ensure the

Communicated by Iris Reinhartz-Berger and Dominik Bork.

This work has been partially funded by the Spanish government (LOCOSS project - PID2020-114615RB-I00 and BODI project - PDC2021-121404-I00) and the Luxembourg National Research Fund (FNR) PEARL program, grant agreement 16544475.

✉ Elena Planas
eplanash@uoc.edu

Salvador Martínez
salvador.martinez@imt-atlantique.fr

Marco Brambilla
marco.brambilla@polimi.it

Jordi Cabot
jordi.cabot@list.lu

1 Universitat Oberta de Catalunya, 08018 Barcelona, Spain

2 IMT Atlantique, Brest, France

3 Politecnico di Milano, 20133 Milan, Italy

4 Luxembourg Institute of Science and Technology, 4362 Esch-sur-Alzette, Luxembourg

[1] According to Gartner Inc, by 2027, chatbots will become the primary customer service channel for roughly a quarter of organizations.

[2] The chatbot market size is growing at a compound annual growth rate (CAGR) of over 20% according to several sources.

Published online: 22 November 2023

CUI is able, for instance, to disable potential queries (scenario 1) for certain user profiles, to provide different answers for the same query depending on the user role (scenario 2) or to provide different information precision levels for the same queries depending on the user-specific privileges (scenario 3).

Several works [16, 30, 42] emphasize the importance of considering security, particularly access control in the definition of CUIs, as highlighted in the scenarios mentioned above. While some approaches exist to enhance security in general UML models or within specific contexts, such as web interfaces, we are lacking concrete solutions for enriching CUIs with robust access control mechanisms. Achieving a comprehensive intent-based control system that could be integrated with other components as part of a multiexperience user interface, which could be built together by using a Multiexperience Development Platform (MXDP), is still a remaining challenge.

To cover this gap, this work proposes to enrich CUI definitions with access-control primitives to enable the definition and enforcement of security policies. 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 model-based generation of such secure CUIs on top of different bot development platforms. As discussed in the related work, this security extension was regarded as needed in other domains such as web services [2], XML documents [13] or Internet of Things [36]. This work enables the same type of support for the new type of interface that CUIs represent.

Our framework (see Fig. 1) is composed of two main components: (1) a design time component (right part of the figure) to enable the specification of the CUI authorization policy and decide on the evaluation strategies; and (2) a runtime component (left part of the figure) in charge of interacting with the user and, based on the specified policies, act accordingly either by allowing or denying the access to the resources depending on the user permissions.

This paper extends our previous paper [41] in several directions:

1. We extend the DSL (Domain Specific Language) from [41] with global permissions to simplify the definition of general policies affecting the bot as a whole and new types of permission constraints for a more fine-grained restriction of the granted permissions (e.g., for temporally-restricted access).
2. We propose a (textual) concrete syntax for the DSL and a modeling editor (with the expected auto-completion and syntax highlighting features, among others) to easily define (and check the well-formedness of) such policies.
3. We discuss a couple of implementation strategies to enforce the rules on top of bot frameworks, including the full architecture and infrastructure for a generative approach that relies on Casbin to integrate the runtime evaluation of policies on top of the Xatkit framework.
4. We extend the related work to compare our approach with other solutions.

The rest of the paper is structured as follows: Sect. 2 provides the background about CUIs and access-control; Sect. 3 introduces a bot we will use as a running example along the paper; Sect. 4 specifies the language we propose to add model-based access-control for CUIs design; Sect. 5 describes how to evaluate and enforce the policy rules for CUIs; Sect. 6 performs a comparative analysis and summarizes the related work; and finally Sect. 7 concludes with the highlights and further work.

2 Background

In this section, we review the basic concepts where this work relies on: Conversational User Interfaces (CUIs) (see Sect. 2.1) and Role-Based Access-Control (RBAC) (see Sect. 2.2).

2.1 Conversational user interfaces (CUIs)

Conversational user interfaces (CUIs) aim to emulate a conversation with a real human. The most relevant examples of CUIs are *chat*bots and *voice*bots.

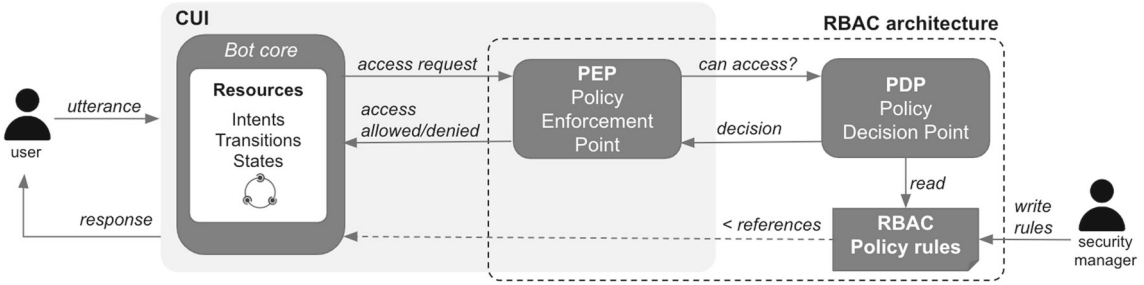

Fig. 1 Framework overview

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. Bots are classified in different types depending on the channel employed to communicate with the user. For instance, in *chat*bots the user interaction is through textual messages, in *voice*bots is through speech, while in *gesture*bots is through interactive images. Note that in all cases bots are the mechanism to implement a conversation, it just changes the medium where this conversation takes place.

The conversation capabilities of a bot are usually designed as a set of *intents*, where each intent represents a possible user's goal when interacting with the bot. 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,[3] Amazon Lex[4] or Watson Assistant).[5] As part of the match, one or more parameters (called also *entities* in the bot terminology) in the user utterance can also be recognized, in a process known as named entity recognition.

When there is a match, the bot back-end executes the required behavior, optionally calling external services; and finally, the bot produces a response that it is returned to the user.

For non-trivial bots, the behavior is modeled using a kind of state-machine expressing the valid interaction flows between the users and the bot.

2.2 Role-based access-control

Access-control [45] 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, but also any active entity which may interact with it) 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) [1], Discretionary Access-Control (DAC) [1], Attribute-Based Access-Control [21], and Role-based Access-Control (RBAC) [44].

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] https://cloud.google.com/dialogflow/.

[4] https://aws.amazon.com/lex/.

[5] https://cloud.ibm.com/catalog/services/watson-assistant.

3 Running example

In this section, we introduce a simplified version of an ecommerce chatbot, which will be used as a running example in the remainder of this paper. Broadly, an ecommerce bot (see Fig. 2) interacts with the user to provide customer service, answer questions, recommend products, gather feedback, and track engagement, among many others. The simplified ecommerce chatbot we use in this paper addresses the user intentions described in Table 1.

The user intentions match the corresponding chatbot intents and prompt the bot to trigger a specific behavior in response. This execution logic is specified using the UML state-machine formalism [35], which allows expressing the valid (conversational or event-driven) interaction flows between the user and the bot. The state-machine of our ecommerce chatbot (see Fig. 3) includes an initial state (*Greet user*), where the bot greets the user, after which automatically navigates to the main menu awaiting for a user request. In

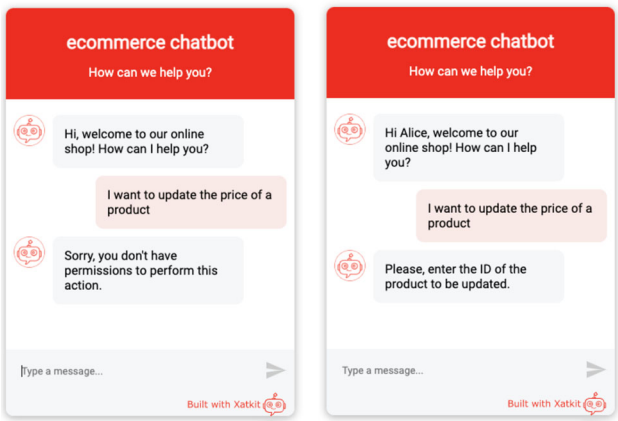

Fig. 2 ecommerce chatbot interaction with an anonymous user (left) and an employee (right)

Table 1 Intents of our ecommerce chatbot

Intent	Description
1. Find product	Searches for a product within the catalogue
2. Get product details	Returns information about a product
3. Buy product	Records an order
4. Update shop catalogue	Updates the product catalogue

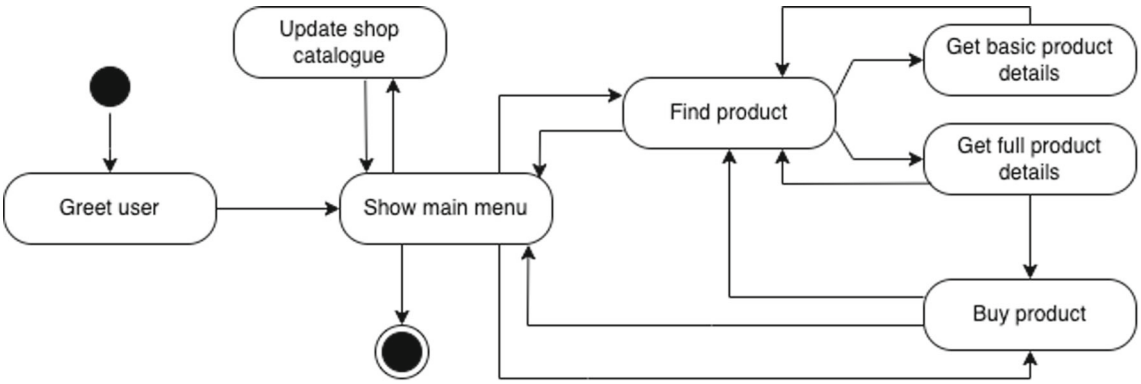

Fig. 3 ecommerce bot state machine

Table 2 ecommerce bot policy rules

Role	Type	Permission
anonymous	Intent	Find product
		Get product details
	State	Greet user
		Show main menu
		Find product
		Get basic product details
	Transition	*from* Greet user *to* Show main menu
		from Show main menu *to* Find product
		from Find product *to* Get basic product details
		from Find product *to* Show main menu
		from Get basic product details *to* Find product
registered	All	GRANT ALL Permissions (ecommerceBot)
		EXCEPT FOR
		Intent Update shop catalog
		State Get basic product details
employee	All	GRANT ALL Permissions(ecommerceBot)

case the user request matches with the *find a product* intent, a transition to the *Find product* state is navigated. Once in that state, the user can ask for additional product details or go back to the main state. The same logic is applied to the rest of the states.

Clearly, not all intents of this ecommerce bot should be available to all users, as the bot is serving both internal and external users of the shop and, for instance, as can be seen in Fig. 2, we do not want external users to be able to modify the shop catalogue (or they could lower the price of a product before buying it).

Therefore, we should add proper access control to manage who is entitled to do what on which resource and protect the others. To this end, in this example, we differentiate between three roles (*anonymous*, *registered*, and *employee*), each of them able to interact with different resources according to the permissions detailed in Table 2. Note that the permissions do not only restrict what intents can be matched but also the possible navigational transitions and reachable states for a matched intent depending on the role.

For instance, according to these permissions, an *anonymous* user can only match two intents (*Find product* and *Get product details*) but, when matching the latter, an anonymous user will only be able to follow the transition leading to the *Get basic product details* as only registered users can see the full product details. On the opposite side, the *employee*s have full permissions to use all types of the resources, while *registered* users can use all the resources, including buying products, except for the update of the catalogue (reserved to employees only) and the state *Get basic product details*, as registered users access the improved version on this state.

Note that, as can be seen in Table 2, to simplify the assignment of permissions, we can use global permissions that grant access to all bot components to a role including the poten-

tial use of an *except for* clause where we could list the few exceptions on a global role.[6]

4 Modeling access-control policy rules for CUIs

The first part of our framework (see Fig. 1 right) consists of a design time component to enable the specification of the CUI authorization policy. This authorization policy is expressed via a policy language. To this end, in this paper we propose to extend a generic CUI language, based on our previous proposal [41], with new modeling primitives, inspired by other RBAC-like languages and tailored to the CUI domain, to add access-control semantics to CUIs.

As any DSL, this extended *access-control-CUI* DSL is defined through two main components [27]: (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. Both aspects are platform-independent. This enables the analysis of the access-control information disregarding the specificities of the concrete CUIs security features and implementation and its deployment on top of different authorization libraries depending on the needs of the system as explained in Sect. 4.5.

The next subsections describe these elements in more detail, after an initial review of the core CUI modeling language that we are extending.

4.1 CUI metamodel

The CUI-specific metamodel part (colored in gray in Fig. 4) is a simplified version of the metamodel defined by the authors in [40] 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 goal when interacting with the CUI. Intents, which are a specific type of *Event* (as bot 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 from several devices (which could also be restricted as part of the security policy).

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

[6] When listing the exceptions we skip listing all impossible transitions, i.e. transitions between two states where at least one of them is not reachable by the role.

Transitions. The metaclass *Transition* represents the potential bot evolutions 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 fine-grained control over the firing of the *Transition*.

Matchings. This part of the metamodel can be used to track the user requests at runtime and link them with the matched intents and recognized parameters. This is useful for the evaluation of the policies but could also be used for logging purposes. In particular, the metaclass *UserRequest* represents the user utterances, which may match with one or more *Intents*. In order to contextualize the requests and apply access-control we add several parameters such as the location and timestamp of the request. Each recognized intent is represented by the *MatchedIntent* metaclass, which stores the level of recognition confidence provided by an external intent recognition provider. For each *Parameter* of an *Intent*, the corresponding *MatchedIntent*s keep its *value* in the association class *ParameterValue*. Each of these *ParameterValue*s corresponds to a specific *text* fragment of the analogous *UserRequest* which has derived the *MatchedInput*.

4.2 RBAC metamodel

The RBAC metamodel part is an extended version of the metamodel presented by the authors in [41] and extends the standard RBAC concepts mentioned in Sect. 2 to adapt it to CUIs. This is done by weaving the RBAC and CUI concepts through the definition of a set of permissions that 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 (intents, transitions, states, or all of the above). Its main elements are:

Resources. The metaclass *Resource* represents the objects that are part of a CUI and that we may want to protect. In the context of CUIs, we consider that resources are basically the different components of the bot, which can be of three types: *Intent*s, *Transition*s or *State*s. Protecting intents will prevent some roles matching part of the CUI's intents. This may be necessary, for instance, to prevent specific users from asking for some specific functionality (if they do not have permissions to match the intent, their request will not be recognized by the bot). On the other hand, protecting transitions and states will allow, once an intent has been matched, to execute different behaviors depending on the role that triggered the intent. This may be useful, for instance, to provide different answers for an intent depending on the role of the user. We expand on this in the *Action* description below.

Note that, to simplify the definition of policies, we also consider the whole *Bot* as a resource itself. This enables

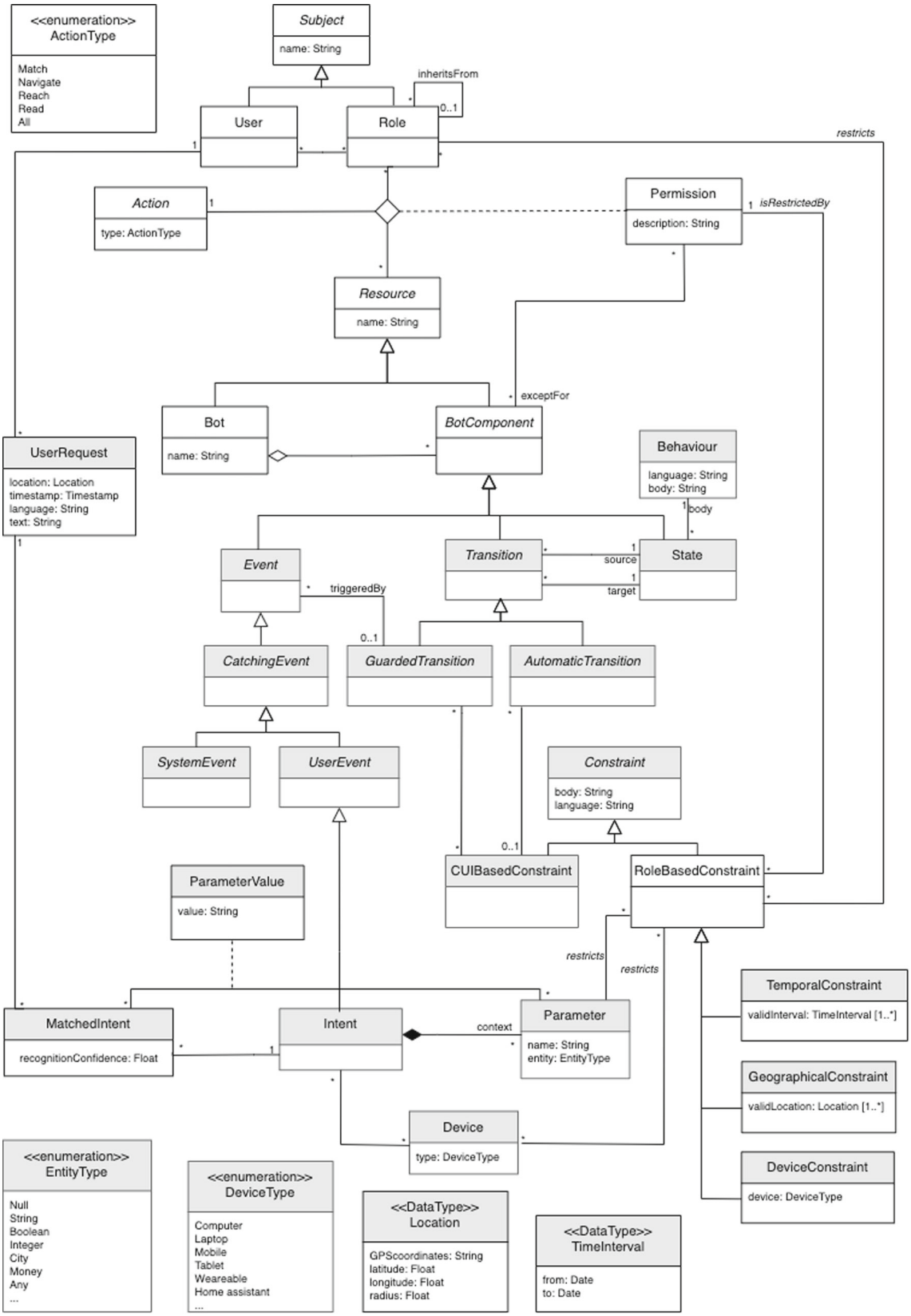

Fig. 4 Access-control CUIs metamodel

granting certain roles permission for the whole bot, especially useful for roles that must be able to perform any action on any bot component, as is the case with the employees in our running example. This kind of GRANT ALL level permission follows the standard semantics of this type of permission as proposed in the database realm. Besides, for roles that should have almost all permissions, we can add an *except for* clause listing those bot components to whom the *grant all* permission does not apply. These global permissions will then be unfolded during the generation process and converted into a set of individual policies over concrete components as the other policies to simplify its treatment and provide a homogeneous environment for the policy enforcement mechanism.

Subjects. The metaclass *Subject* represents the active actors which interact with the CUI. Following an RBAC approach, we define two kinds of subjects: *User*s and *Role*s, where users get roles assigned. Role inheritance is supported. To stick to a pure RBAC approach, permissions cannot be assigned to individual users. Nevertheless, it is always possible to create a role that only such user would have and assign permissions to that role if needed.

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 as main possible actions performed by subjects are *Matching* an intent, *Reaching* of a state and *Navigating* (i.e., traversal) a transition. We would like to remark that, while the above list of actions are the most common ones, we could define additional ones. For instance, a *Read* action for intents. If we grant a role a *Read* action permission over an intent, the users with that role will be able to see the intent exists (e.g., as part of help functions) but will not be able to match it (similar to the concept of gray/disabled buttons or options in a Graphical User Interface). This could be used to push the user to perform the actions required to acquire more permissions (e.g., register in the web application).

Permissions. The metaclass *Permission* represents the right to perform a given *Action* (matching an intent, reaching a state or navigating a transition) on a given *Resource* (an intent, state or transition) granted to a specific *Role* (corresponding to a CUI user). Intent permissions enable users to trigger a bot behavior for certain intentions. State permissions control whether a certain state can be reached or not (even when an intention could lead the user to such state, if that is the case and the intent is matched, the user will be redirected to a different state also linked to the same intent or it will just stay in the same state if no alternative is available). Finally, transition permissions enable a more fine-grained control of the potential user interactions when needed; indeed, even after the match, we can restrict to which state that match should move the user to (e.g. even when a role has permissions to reach a certain state we may want to control the path a user with a given role has to follow to reach that state preventing the transition of navigations that could reach that same state but via other paths).

In order to easily define large permissions, our metamodel allows defining an *All* special permission at the bot level, implying that the role with such global permission will be able to perform all types of actions on the bot components. This can be expressed by relating the permission directly with the *Bot* metaclass, which includes all its components. Besides, our metamodel allows limiting these global permissions through the relationship *exceptFor*, which allows defining which resources of the bot will not be part of the global permission assigned to a specific role.

As we will see later, we can add a WFR to ensure that the bot components of an *exceptFor* relationship are part of the bot to which the permission that holds the exception is related to.

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 to restrict the permissions. Our metamodel explicitly includes some predefined types of constraints regarding the geographical location, the temporal periods when the user can express a certain request, the allowed devices for doing so or even the possible set of values for the parameters to be matched during the Intent recognition phase. But many other ad-hoc constraints could be defined as well using a generic constraint language such as OCL.

4.3 Concrete syntax

As mentioned before, domain-specific languages are usually equipped with a concrete syntax that enables users to create instances (i.e. models conforming to the DSL) without dealing directly with the abstract syntax.

In this case, we decided to provide our language with a textual syntax instead of a graphical or form-based one, as we believe it results in more compact policy specifications that are easier to create and understand. Nevertheless, we could have several alternative concrete syntaxes for the same metamodel, so it would be equally possible to define a graphical syntax for our language or even a mix of the two.

Textual syntaxes are defined via a grammar. As explained in more detail in Sect. 4.5, our grammar is created with Xtext,[7] a state-of-the-art language workbench. Thanks to the Xtext support we can provide advanced editing facilities to the bot designers interested in using our access-control language for their CUIs, e.g. to help them in the creation of *valid* instances.

[7] https://www.eclipse.org/Xtext/.

As we will see in more detail in Sect. 4.5, Listings 3 and 4 show, respectively, the grammar for our language and a partial representation of the policy rules for our running example written in this grammar.

4.4 Policy validation and correctness

The correctness of a policy can be defined and checked at different levels.

The first level is to verify a policy is well-formed, i.e., to check that it is properly aligned with its metamodel and can be expressed as an instance of such metamodel. This is guaranteed by our implementation (see below). This would cover basic structural validations like the fact that permissions cannot be directly linked to users as there is no relationship between the *Permission* and *User* classes in the metamodel.

Additional well-formedness rules can be added by attaching to the metamodel OCL constraints (called, in this context, *well-formedness rules*, WFR). As will be seen in the next section, these WFRs are not purely decorative, instead, they are checked automatically by our editor as Eclipse also includes an OCL engine able to check OCL-based WFRs.[8] As an example, the OCL rule of Listing 1 states that when granting global permission on a bot, the components that are listed as an exception must be components belonging to that same bot.

Listing 1 Consistency rule.

```
1  context Permission inv:
2      self.resource.oclIsTypeOf(Bot) implies
3          self.exceptFor->forAll(c| c.bot = self. resource)
```

Beyond well-formedness we may also want to check that the policies *make sense*. Note that our permissions are always positive and additive in our model.[9] However, several undesirable situations, such as redundancy between permissions could appear, for instance, if the policy includes a permission directly granted to an intent for a role that has also global access to all bot components (and therefore already has the permission to match the intent through this *All* access). Listing 2 shows an example OCL constraint that could prevent this by checking that a role has no global permissions for a role that already has individual ones. As this is a redundancy and not an error, we could decide to just show a warning instead of an error in this case.

Listing 2 No redundancy rule.

```
1  context Permission inv:
2      let allPerm: Set(Permission) =
3      Permission->select(e | e.role = self.role) in
4      self.resource.oclIsTypeOf(BotComponent) implies
5      allPerm->excludes(self.bot)
```

Despite the addition of WFR to check the correctness of the permission rules, the absence of conflicts does not guarantee the policies to be correct, as other anomalies may exist. For instance, the existence of empty roles (roles that have zero permissions) or isolated resources (resources where nobody can perform any action upon). These scenarios can also be checked by evaluating the policy once their definition has been completed. Similar to the WFRs above we could write an OCL query to return the potentially problematic roles and components.

Finally, we may also want to make sure that permissions are consistent with respect to arbitrary constraints attached to them. In the general case, i.e., when the constraints are arbitrary constraints written in a highly expressive language like OCL (that includes iterator expressions, null values, tuple types,...), the use of formal verification methods is required, each one with its own set of trade-offs [17]. Specific adaptations of such methods for RBAC-specific checking do exist as well, e.g. [29, 39, 43, 47].

4.5 Modeling editor for RBAC policies

To prove the feasibility of our approach, we describe in this section the tool support we provide to facilitate such policy modeling.

As mentioned before, we use the Xtext language workbench in order to provide our access-control language with a concrete textual syntax and an associated editing tool.

The central artifact of Xtext is the grammar, which provides the syntactic rules valid *textual* instances must follow. Our grammar, shown in Listing 3, defines a textual language in which RBAC policies are composed by first a number of declarations (lines 1 to 15), and second a number of rules (lines 18 to 42) which refer to these declarations.

[8] https://projects.eclipse.org/projects/modeling.mdt.ocl.

[9] Keep in mind that the global permissions with *except for* restrictions are unfolded into a set of positive permissions on the individual components so the *except for* clause is not a negative condition, just restricts the number of positive permissions that are generated during the unfolding.

Listing 3 CUI-RBAC Xtext grammar.

```
Policy:
 'Sec_Policy' name = ID
 'Declarations' '{'
    'Roles:'
    subjects+=Role (',' subjects+=Role)*
 '}'

 'Rules:' '{'
    permissions+=Permission
    (permissions+=Permission)*
 '}'

 ('Constraints:' '{'
    constraints+=Constraint
 (constraints+=Constraint)*
 '}')?
;


Role returns Role:
    name = ID
    ('inheritingFrom' inheritFrom=[Role])?
;

enum AAction returns AccessAction:
    Match='Match' | Navigate='Navigate'
    | Reach='Reach' |
    Read='Read' |  All='All'
;

Constraint returns RoleBasedConstraint:
    'Constraint' name = ID ':' '[' 'using'
     language=ID ']' body=STRING
;

Permission:
    'GRANT'  action=AAction
    'to' role+=[Role]
    'on' resource=[def::Bot|QualifiedName]
    ('exceptFor' exceptFor+=
    [def::BotComponent
    |QualifiedName]
    (',' exceptFor+=[def::BotComponent
    |QualifiedName])*)?
    ('(' 'withConstraint:' constraints+=
     [RoleBasedConstraint
    (constraints+=[RoleBasedConstraint]
    )* ')')?
    ';'
;

QualifiedName: ID ('.' ID)*;
```

Note that the definition of the bot itself (i.e., the state machine expressing the bot behavior with all the intents, transitions,...) is not part of this grammar. Here we just reference those elements, defined in a separate model with the corresponding CUI modeling language of choice. This way we achieve separation of concerns to facilitate the collaboration between CUI and security experts. Nevertheless, we believe our RBAC language is comprehensible enough to empower CUI designers without a deep security knowledge to still define the key security policies for their bots.

As an example, Listing 4 shows an instantiation of the above grammar to represent the access-control policy of our ecommerce bot policy (i.e., the rules showed in Table 2).

Listing 4 CUI-RBAC policy rules of our running example.

```
Sec_Policy e_commerceBot_policy

Declarations{
    Roles: registered, employee, anonymous
}

Rules: {
    GRANT Match to anonymous on eCommerceBot.
    _FindProduct;
    GRANT Match to anonymous on eCommerceBot.I
    _GetProductDetails;
    GRANT Reach to anonymous on eCommerceBot.S
    _Greetuser;
    GRANT Reach to anonymous on eCommerceBot.S
    _ShowMainMenu;
    GRANT Reach to anonymous on eCommerceBot.S
    _FindProduct;
    GRANT Reach to anonymous on eCommerceBot.S
    _GetBasicProductDetails;
    GRANT Navigate to anonymous on
    eCommerceBot.T1;
     ...
     ...
    GRANT All to registered on
    eCommerceBot except
    For
    eCommerceBot.S_GetBasicProductDetails,
    eCommerceBot.I_UpdateShopCatalogue;
    GRANT All to employee on eCommerceBot;
}
```

Xtext generates an Eclipse-based IDE which includes a textual editor of the grammar with auto-completion, syntax highlighting, and error detection with respect to the metamodel and its WFRs. Note that Xtext uses our metamodel as the abstract syntax for our language and thus, correct textual policies are internally represented as models conforming to it. As an example, Fig. 5 shows a screenshot of our textual editor, where two constraint violations have been introduced. First, the rule `Grant All to registered on eCommerceBot exceptFor...` contains an intent in the *except for* list (the intent *Get Monthly Goals*) that is not part of the ecommerce bot. Second, the rule `Grant Match to employee on eCommerceBot.I_BuyProduct` is redundant with respect to the rule `Grant All to`

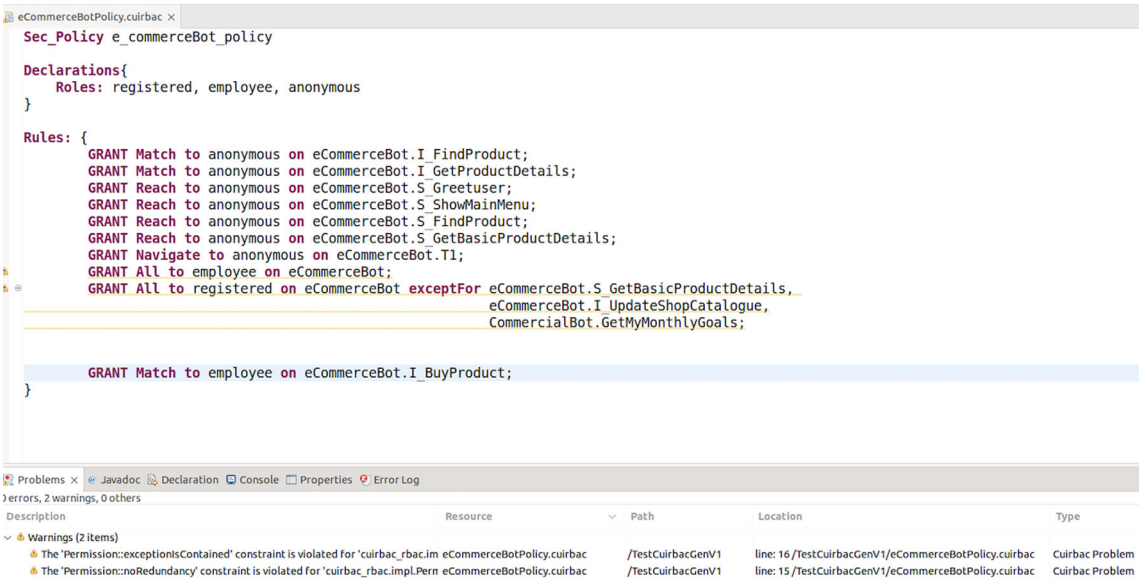

Fig. 5 Screenshot of the Eclipse-based editor of our DSL

`employee on eCommerceBot`, since the last gives permissions to the same role for the entire bot.

5 Evaluating and enforcing policy rules for CUIs

As explained previously, the second part of our framework (see Fig. 1 left) consists of a runtime component in charge of interacting with the user and act accordingly either by allowing or denying the access to the resources depending on the user permissions.

The recommendation in the implementation of modern policy frameworks is to separate the infrastructure logic from the application logic by using a reference monitor architecture [22]. This architecture consists of two basic components: a **Policy Enforcement Point** (PEP) and a **Policy Decision Point** (PDP). As shown in Fig. 1, access requests to the bot resources 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. Note that values for attributes such as location or time (or any other contextual attribute referenced in the access conditions) must be attached to the access request (or directly taken from the runtime environment) in order for the PDP to evaluate the match.

There are several possible strategies to implement this architecture, depending on the level of internal access to the chatbot engine that the chatbot designer has.

If 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 part of 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.

Next subsections discuss both scenarios in more detail, especially the latter one as it will be the most common scenario for chatbot designers, which can not typically modify the chatbot engine themselves. Note that, as a trade-off, in this scenario the expressiveness of the rules we can evaluate may be restricted by the capabilities of the chosen external library (e.g., it may not support temporal or geographical or other types of complex constraints).

5.1 Enforcing RBAC policies via an external library

The first strategy we propose to enforce RBAC policies is by relying on a third-party library able to evaluate an access request against a policy. Then, based on this decision, the chatbot will need to act accordingly.

This enforcement strategy requires then two steps:

1. Translate the modeled policies to the input language used by the external library.
2. Integrate in the bot definition the calls to this external library.

Next, we see how we could implement each step. In particular, we will do it using Xatkit [14] as chatbot framework and Casbin[10] as a RBAC library. Casbin is an open-source access control library that provides support for enforcing authorization based on various access control models. Although Casbin itself supports many programming languages such as Go, Java, Nodejs, PHP, Python, Microsoft.NET, C++, and Rust, a similar approach could be followed to call our access-control rules from other chatbot frameworks and by relying on other external libraries such as Open Policy Agent[11] or Ory.[12]

5.1.1 Step 1: Generating RBAC policies

The policies written with our modeling editor are used at runtime in order to decide upon access requests. This can be done by either developing a brand new runtime component for our language or by re-using some existing authorization framework. In this paper, we opt for this second option as this is a more flexible and simple solution that avoids reinventing the wheel and facilitates the integration of our approach in different technical stacks.

As a proof of concept, we show how to policies written in our access-control for CUIs language will be translated to Casbin policies. A similar approach would be followed to translate them to other authorization systems.

In order to use Casbin with a specific access-control model, the *developer* needs to provide two configuration files. First, a *.conf* file containing an access-control model definition (e.g., ABAC, RBAC, etc). As an example, Listing 5 shows a *.conf* file that corresponds to the default RBAC model as provided by Casbin.

Listing 5 Casbin RBAC predefined configuration.

```
1  [request_definition]
2  r = sub, obj, act
3
4  [policy_definition]
5  p = sub, obj, act
6
7  [role_definition]
8  g = _, _
9
10 [policy_effect]
11 e = some(where (p.eft == allow))
12 [matchers]
13 m = g(r.sub, p.sub) && r.obj == p.obj &&
14  r.act == p.act
```

And second, the *developer* needs to provide a *.csv* file containing the actual access-control policy. As an example, Listing 6 shows the policy, in the form of comma-separated values, that corresponds to the aforementioned Casbin model. Each line corresponds to a policy rule that contains: (1) the identifier of a role; (2) the CUI resource on which the permission is going to be granted (an intent -I-, state -S- or transition -T-); and (3) the action on the resource (matching, reaching or navigating).

Listing 6 Excerpt of Casbin RBAC policy of our running example.

```
1  p anonymous, I_FindProduct, Match
2  p anonymous, I_GetProductDetails, Match
3  p anonymous, S_Greetuser, Reach
4  p anonymous, S_ShowMainMenu, Reach
5  p anonymous, S_FindProduct, Reach
6  p anonymous, S_GetBasicProductDetails, Reach
7  p anonymous, T1, Navigate
8  p employee, I_FindProduct, Match
9  p employee, I_GetProductDetails, Match
10 p employee, I_BuyProduct, Match
11 p employee, T1, Navigate
12 p employee, T2, Navigate
13 ...
14 ...
15 p registered, I_FindProduct, Match
16 p registered, I_GetProductDetails, Match
17 p registered, I_BuyProduct, Match
18 p registered, S_Greetuser, Reach
19 p registered, S_ShowMainMenu, Reach
20 ...
21 ...
22 p registered, T1, Navigate
23 p registered, T2, Navigate
24 ...
25 ...
```

To generate Casbin policies from our policies written in our language (see Listing 4) we employ code generation techniques. The generation focuses on the *.csv* Casbin file. Note that the *.conf* file may be customized (notably to modify the types of elements used in the policy rules, the access requests, and the match process) but this configuration does not depend on the specific access-control policy of a given CUI, and thus, does not need to be (re)generated from it.

Again, we use Xtext facilities to implement our code generation. Xtext integrates code generation triggering options in the generated IDE, so policies are automatically (and transparently) translated to the target language (Casbin) as the policy file is saved. As our model and the Casbin model are very close the translation is straightforward. Listing 7 shows the actual code generator written in the Xtend Java dialect. It starts by retrieving the root of the policy model (line 5), creates a *.csv* file and then fills it by calling the *genPolicy()* method (line 7). This method iterates on the permissions contained in the policy model element and generates for each permission a Casbin RBAC permission rule (p, sub-

[10] www.casbin.org.

[11] https://www.openpolicyagent.org/.

[12] https://github.com/ory.

ject, object, action). Finally, note that our code generator performs the flattening of permissions (lines 18 to 24).

Listing 7 Excerpt of Casbin policy generator.

```
1  override void doGenerate(Resource resource,
2     FileSystemAccess2 fsa,
3     IGeneratorContext context) {
4     for (e : resource.allContents.toIterable.
5     filter(Policy)) {
6        fsa.generateFile(
7           e.name + ".csv",
8           e.genPolicy)
9     }
10 }
11
12 def genPolicy(Policy p)'''
13    «FOR r : p.permissions»
14       «val exceptions = r.exceptFor»
15       «IF r.resource instanceof Component»
16          p «FOR role : r.role «role.name»«
17          ENDFOR»
18          «r.resource.name», «r.action»
19       «ELSE»
20          «val botResource = r.resource as
21          Composite
22          «FOR cp : botResource.component»
23             «IF !exceptions.contains(cp)»
24                p «FOR role : r.role» «
25                role.name»«ENDFOR»,
26                «cp.name», «cp.printAction»
27             «ENDIF»
28          «ENDFOR»
29       «ENDIF»
30    «ENDFOR»
31 '''
32
33 def String printAction(cuirbac_rbac.Resource
34    cp){
35    switch cp{
36    case cp instanceof Intent : "Match"
37    case cp instanceof State : "Reach"
38    case cp instanceof Transition :
39     "Navigate"
40    default : "NoAction"
41    }
42 }
```

5.1.2 Enforcing RBAC policies

Following this strategy, the security checks are explicitly added to each transition of the state machine of the running bot.[13]

As an example, Listing 8 shows how the transition from the *Show Main Menu* state to the *Update Shop Catalogue* state has been modified to add a new condition that checks the user's role is allowed to match the *Update Shop Catalogue* intent and only proceeds to the corresponding state when this condition is true (lines 3 to 5). Otherwise, as seen in Fig. 2 the bot inform the user she has not permissions to perform that action (lines 6-8). The complete implementation for our running example bot is available.[14]

Listing 8 Policy Enforcement Point (PEP) implementation.

```
1  showMainMenuState
2    .next()
3    .when(intentIs(updateShopCatalogueIntent).and
4    (c -> enforcer.enforce
5      (role,"UpdateShopCatalogueIntent",match)))
6      .moveTo(updateShopCatalogueState)
7   .when(intentIs(updateShopCatalogueIntent).and
8     (c -> !enforcer.enforce
9     (role,"UpdateShopCatalogueIntent",match)))
10    .moveTo(informAboutPermissions);
```

Note that, even if access-control evaluation and enforcement are now explicit, they could still be automatically added to the concerned transitions. Given a security policy and a plain chatbot definition, we could automatically instrument all relevant transitions with the proper access-control checks. For Xatkit bots, and as Xatkit is a Java-based engine, a library such as JavaParser[15] could be used to traverse the AST of the bot definition and modify it to add the Casbin calls on each transition.

5.2 Enforcing RBAC policies using a security-aware chatbot engine

A radically different strategy to enforce RBAC policies is to rely on a chatbot engine that already offers RBAC primitives as part of its core engine execution. This would be ideal as our work would consist in translating our modeled policies into the chatbot engine definition language (same as we would need to do for the other chatbot components such as the intents, states,...) and we are done. The engine would take care of internally analyzing and evaluating the policies every time it is needed.

Unfortunately, there is none at the moment but if the engine is open source, you could implement this by yourself. Obviously, the way to modify the engine with *access-control semantics* depends on the engine. Here we briefly comment on how we are implementing this on an experimental branch

[13] Note that constraints on intents and states are at this point transformed into constraint on the transitions related to those intents and states as, without modifying the chatbot engine, the transition is the only point where we can influence the bot behavior. Nevertheless, this kind of *unfolding* can be automated and does not require additional manual effort from the user.

[14] https://github.com/elenaplanas/xatkit-RBACBot/tree/jCasbin.

[15] https://javaparser.org/.

of Xatkit as this is the chatbot engine we created ourselves and therefore we one we are more familiar with.

The first step is to extend Xatkit's FluentAPI to enable the definition of security policies as part of the bot definition. Calls to this extended API would just store in the internal bot model of the chatbot engine the security of details of the chatbot execution same as it stores all the info on the intents, states and transitions to be able to execute the bot and evolve it from one state to the other in response to all types of events.

Listing 9 shows a small example of how the *Update Shop Catalogue* intent of our running example would be defined using this extended FluentAPI. Note how now the definition of the security policies is fully integrated with the own chatbot definition, feeling much more natural and easy to use. We first create the new role/s and then we just add the roles with permissions granted as part of the definition of the bot components, the update shop intent in the example. Moreover, now we do not need to manually modify the state machine part to include any external call, once the policy is defined, all the rest is taken care internally.

Listing 9 Defining security policies as part of the bot definition in Xatkit.

```
val employeeRole = role("Employee")
    .constraint("A first constraint
     for the role");

val updateShopCatalogue = intent
     ("UpdateShopCatalogue")
    .trainingSentence
    ("I want to update the
      shop catalogue")
    .trainingSentence
     ("I want to update the
     price of a product")
    .trainingSentence
     ("There are some
     products to change");
    .permission()
    .role(employeeRole);
```

The next step is to dynamically filter the set of possible intents to match at any given moment depending on the user permissions. The concrete strategy depends on the features offered by the NLP Engine employed by the chatbot engine (Xatkit can be plugged to several NLP Engines, including DialogFlow, nlp.js or our own engine).[16]

If the NLP Engine allows turning off and on the set of available intents automatically, we can then disable those intents that the current user should not be able to match. This is what DialogFlow supports so in this case we send to DialogFlow the set of Intents (the *context* in Xatkit terminology) that it should consider before every intent matching execution as shown in Listing 10. Starting from all intents that are linked to output transitions from the current state (as these are the only ones that could be potentially matched at this point) we filter out those for which the user has no permission. If the NLP Engine only allows an initial bot deployment that cannot dynamically evolve, we need to let the user match any possible intent and, *a posteriori* remove from the list of matches those that correspond to intents that should not be available based on the user's role.

Listing 10 Dynamic enabling of intents before calling the DialogFlow NLP engine based on the state and user permissions.

```
public @NonNull Iterable<Context>
createOutContextsForState(@NonNull
DialogFlowStateContext context) {
    List<Context> result = new ArrayList<>();
    State state = context.getState();
    Iterable<IntentDefinition>
     accessedIntents =
    state.getAllAccessedIntents();
    accessedIntents.forEach(intent -> {
        if hasPermission(intent,
        context.getsession().getUser().
         getrole()) {
            Context.Builder builder =
             Context.newBuilder().
            setName(ContextName.of(this.
             configuration.getProjectId(),
            context.getSessionName().
             getSession(), "Enable" +
            intent.getName()).toString())...);
            result.add(builder.build());
        }
    });
    return result;
}
```

6 Comparative analysis with the related work

Modeling of security concerns has been a topic largely studied in the modeling community. In the following subsections we review a variety of access control proposals for different domains. First, we explore several general approaches (see Sect. 6.1), which aim to define access control over any general element. Then, we explore in more detail other approaches focused on two specific domains closer to the topic of this paper: Web-based UIs (see Sect. 6.2) and, obviously, Conversational-based UIs (see Sect. 6.3).

In order to provide a theoretical evaluation of our proposal, we also analyze the expressiveness of our language to check whether it is powerful to express complex access control policies by comparing it with the related solutions.

[16] https://github.com/xatkit-bot-platform/xatkit/wiki/Intent-Recognition-Providers.

6.1 General approaches

Two general well-known frameworks to model security concerns are UMLsec [23] and SecureUML [31], which extend UML with RBAC primitives. In particular, UMLsec extends UML so that it supports, among other security concerns, role-based access control. The extension is provided in the form of a UML profile and a corresponding verification tool. In the following, we show how UMLsec can be used to describe a RBAC policy on the state machine in Fig. 3. The policy will give employees the right to update the catalogue and registered users the right to see full product details and buy products. First, we need to apply the stereotype ≪*rbac*≫ to the package containing the state machine. This enables three tags: (1) *protected actions*, which must contain the list of activities to be controlled; (2) *role*, which may have as its value a list of pairs *(actor,role)* where actor is an actor in the activity diagram; and (3) *right*, which value should be a list of pairs *(role,right)* where role represents any role previously declared in the *role* tag and right, a protected activity listed in the *protected actions* tag (meaning the role gets the permission to execute the activity). Supposing we have buyer and worker actors declared in the activity diagram corresponding to the state machine in Fig. 3, the final policy is shown in Listing 11. Fine-grained access-control (e.g., for the verification of conditions) may be achieved by the use of *guarded objects* [24], but this requires detailed design models.

Listing 11 UMLsec RBAC policy

```
{protected actions:
  {Update shop catalogue,
   Get full product details,
    Buy product}
}
{role:
  {(buyer, registered),(worker, employee)}
}
{right:
  {(registered, Get full product details),
   (registered, Buy Product),
   (employee, Update shop catalogue)}
}
```

On the other hand, SecureUML also includes RBAC concepts in UML with stereotypes, including built-in authorization constraints (e.g., separation of duty) and custom constraints expressed in OCL. However, it is somehow more invasive as security concepts such as permissions, roles, etc., need to be modeled along the application logic.

Besides UMLsec and SecureUML, other existing profiles and DSLs focus on access control modeling, including [5, 6, 26] among others (see also [4, 25, 32] for a more general review of the field). Most of these proposals remain at the modeling level and do not cover the generation of an enforcement architecture. Even those that aim at providing an end-to-end solution target either the networking or distributed systems domain and therefore their solution is not directly applicable to the domain of CUIs even if, obviously, all these languages share a core RBAC representation that we also employ in our own DSL.

The need for a specific DSL to express access control rules in new types of systems or components can be observed in many other domains where specific solutions have also been proposed, e.g., web services [2], XML documents [13] or Internet of Things [36]. These extensions provide: (1) At the modeling level, a set of primitives adapted to key concepts in the domain (e.g., Intents in our case), which facilitates the writing of the access control rules by the experts in that domain; and (2) At the implementation level, better integration with the underlying technology to generate a more efficient policy enforcement and decision mechanism. Our proposal follows the same approach but targets a different domain, the domain of CUIs which we believe is an important enough domain in software development and one that is quickly growing in importance in many verticals.

6.2 RBAC for Web-based UIs

In the web interfaces domain, there are a variety of access control proposals for web applications, such as [8, 9, 11, 18, 19, 33]. We consider these proposals are the closest to our contribution, since most bots are displayed as embedded widgets in a web application.

All the above existing approaches support the concept of *Role* (often called *UserGroup*) but the only resource that can be protected are web pages or components that show content within those pages. This would be equivalent to our *states* (see Fig. 6).

Besides, in the web domain, access controls are typically evaluated when clicking a link on the current web page (see Fig. 7). Then, based on the role, the user can be redirected either to the desired page or to an error page (which would be equivalent to our transition permissions). Alternatively, like in the case of the Interaction Flow Modeling Language (IFML) [10], some languages also allow a finer-grained level of control, allowing to specify dedicated rules for visualizing or hiding specific content chunks inside a web page. For instance, Fig. 8 shows a IFML model representing the login page of a site. Upon submission of her credentials, the user is assigned a role (*role1*) and is redirected to a private zone, where some content (*Role1OnlyContent*) is shown only to users with the correct role. This is defined through a dedicated *ActivationExpression* directly in the web navigation model.

The expressiveness of our language is comparable to the ones discussed above. In fact, we consider ours as more expressive, as we can associate complex constraints to the permissions (the only one close to this is [19] that includes

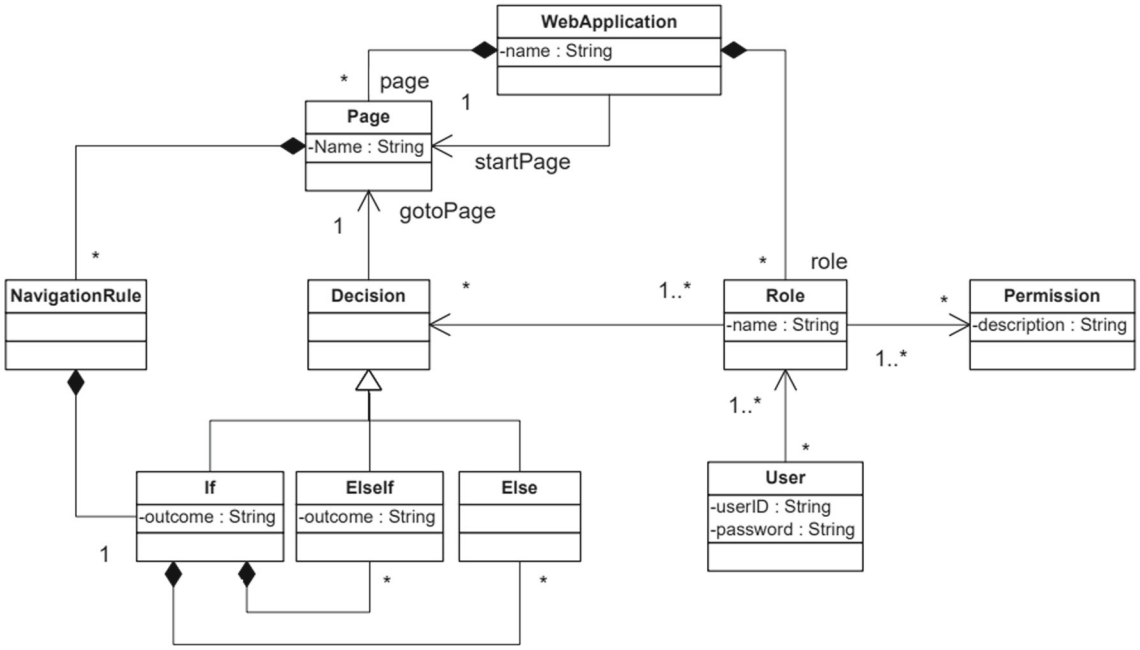

Fig. 6 Example of metamodel excerpt to define roles and their permissions to navigate to web pages, taken from [33]

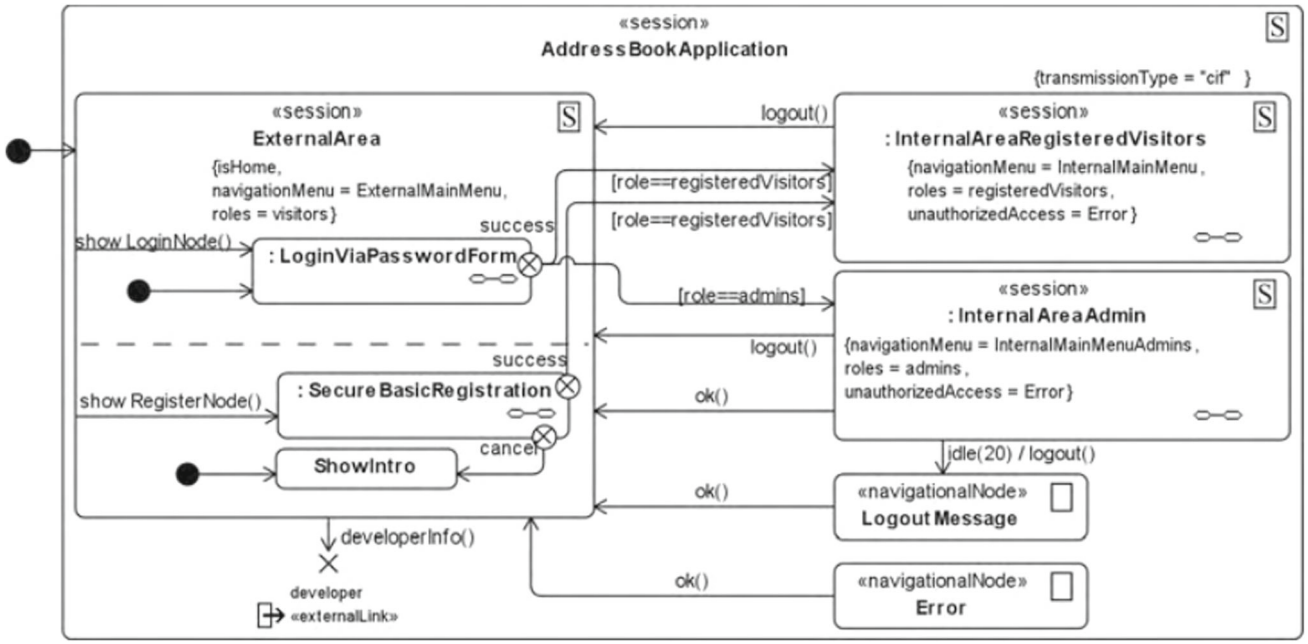

Fig. 7 Example of navigation model showing how roles can be used as conditional terms in the transition from one page to another, taken from [11]

a textual language to express role rules) and offer a larger variety of permissions and resources to constraint.

Our approach also enhances usability through the utilization of this same language richness. For instance, our proposal can prevent a role from reaching a certain state (web page in the web application equivalent) no matter where the user is trying to reach that state from. Instead, in RBAC approaches for web applications, access control is linked to links (more similar to our Intent matching permissions), and therefore, to limit access to a page to a specific role, the corresponding evaluation condition must be added to each and every one of the incoming links to that page or on every component of the page. Note that this is much more time-consuming and error-prone as it requires web application

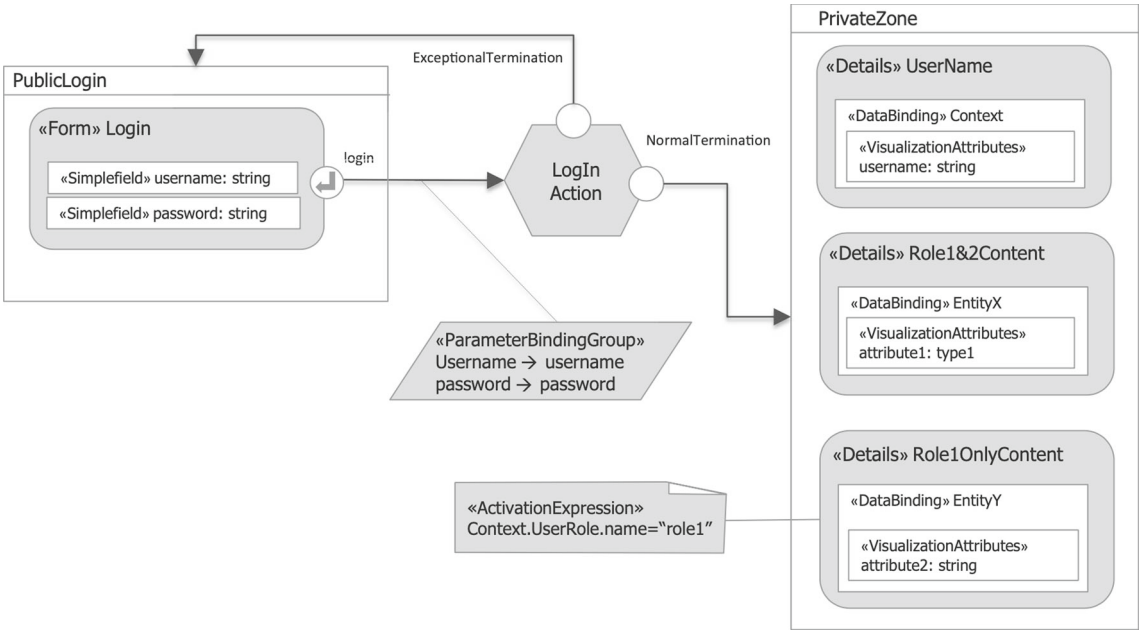

Fig. 8 Example of IFML modeling of user-based and role-based access to resources, taken from [10]: once the user logs into the system, she is assigned a role, and some contents are shown only based on that role

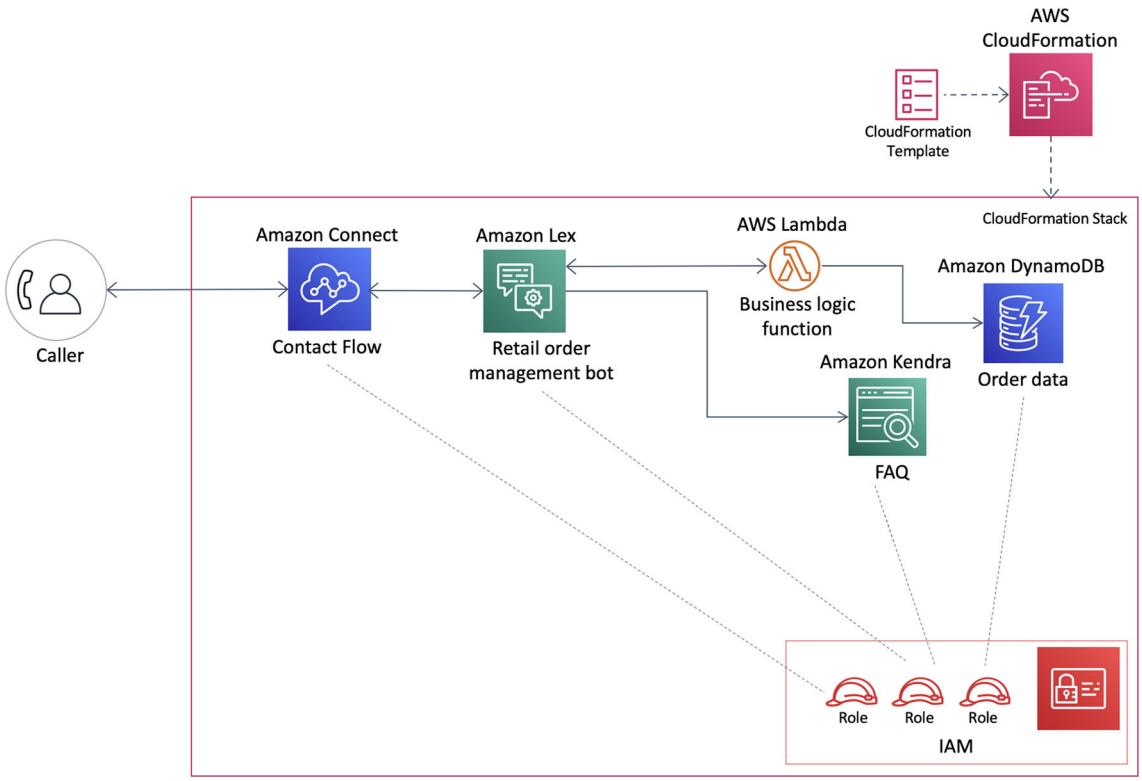

Fig. 9 Example of Identity and Access Management (IAM) roles taken from Amazon Lex (https://aws.amazon.com/es/blogs/machine-learning/build-conversational-experiences-for-retail-order-management-using-amazon-lex)

Table 3 Approaches comparison based on coverage of the control requirements support on resources and events

	Approaches			
	Our proposal	General approaches	Web-based UIs	Conversational UIs
Resources				
Control on Event	✓	×	✓	×
Control on Transition	✓	×	✓	×
Control on State	✓	✓	×	×
Control on Graphical Artifact	×	×	✓	×
Events				
Control on System Event	✓	×	✓	×
Control on GUI Events	×	×	✓	×
Control on Conversational User Events	✓	×	×	✓
Control on Abstract Intents	✓	×	×	×

designers to make sure they do not forget to add the condition to all the relevant links and components. Forgetting to add just one of those rules would create a security threat, enabling unauthorized access to specific contents or whole pages or parts of the site.

6.3 RBAC for CUIs

In the context of CUIs, several authors have expressed the need to secure chatbots, especially in critical domains. For instance, the work of [42] is an example of a concrete chatbot in the e-health domain where the authors emphasize the need for security, in terms of privacy (the data exchange between the bot and the back-end microservices is encrypted) and access control (users must authenticate to be able to use the bot). Similarly, [30] emphasizes the need to put in place a Chatbot Security Control Procedure to address security concerns in another important application domain like banking. Another example of a critical domain is security monitoring where chatbots themselves are used to help in securing other system components [15, 37]. In this case, securing the own chatbots is obviously even more critical for the overall security infrastructure.

Given the importance of securing chatbots, [16] even proposes chatbot providers to attach a Service-Level Agreement (SLA) to their chatbots that covers security aspects. Clearly, chatbots are concerned by (and should be tested against) a number of security concerns, including access control but also privacy [42], GDPR [46], cross-site scripting and injection attacks [7], man-in-the-middle and DDoS attacks [48], social aspects [20] and language-related vulnerabilities [12].

While the above works highlight the need to integrate security aspects, they mostly fail to propose concrete and actionable solutions. This is especially true for access control, the topic of this work. The situation is not better when looking at professional tools where access control is focused on the management of the permissions to enable those who can collaborate in the bot definition. In this context, access control is typically called IAM (Identity and Access Management), for instance in DialogFlow,[17] Amazon Lex[18] or IBM Watson Assistant.[19] At most, as part of an IAM policy (see Fig. 9), you can define whether users must be authenticated to use the deployed bot without any fine-grained role definition, not individual permissions on what each role could do. Moreover, chatbot definition languages, such as [14, 38, 40] do not include modeling primitives to define access control policies.

6.4 Summary

To sum up this section, Table 3 summarizes the functionalities of the most relevant approaches and compares them with ours regarding the type of *resources* and *events* they can control.

Note that general approaches focus only on controlling access to states but do not explicitly support restricting the actions users can perform or a more fine-grained control on how to reach a certain state. On the contrary, access-control proposals for web-based or graphical UIs focus only on restricting what GUI elements a user can interact with but not on the internal logic. This is the only type of user event they can restrict, but cannot deal with conversational events as we do. As we just discussed in the previous section, current chatbot frameworks, even if they obviously have the notion of conversational event, do not offer any possibility to restrict such events at runtime depending on the user. If needed this has to be manually hard-coded in the bot. This is

[17] https://cloud.google.com/dialogflow/cx/docs/concept/access-control.

[18] https://docs.aws.amazon.com/lex/latest/dg/security_iam_service-with-iam.html.

[19] https://www.ibm.com/verify.

precisely what our approach aims to improve by integrating access control as first-class citizens in CUIs as part of a bot definition.

7 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) and proposed a concrete textual syntax to express and implement such access control policy rules. We also show the feasibility of our approach and applicability by showing how it can be implemented on top of an existing chatbot framework using two different strategies.

As further work, we plan to cover additional types of security concerns beyond access-control such as *Confidentiality*, *Integrity*, *Availability*, *Non-re-pudiation*, and many others. The requirements for each of these properties for a given bot should be modeled together with the bot definition, as we have done for access-control policies, via new extensions of our DSL. Then, the concrete implementation strategy will largely depend on the security concern. Some concerns can be delegated to the chatbot engine itself, e.g., protection against DDoS attacks could be taken care of by the engine embedding a rule to automatically disconnect clients sending too many requests, where the threshold is defined when modeling the chatbot. Others to the different clients and connectors to deploy the bots, e.g., encryption requirements could be implemented as the configuration of the communication libraries part of the chatbot widget embedded in webpages in charge of sending user requests to the chatbot server. Finally, GDPR (General Data Protection Regulation) compliance could be facilitated by providing standard conversations (e.g., to consent to the recording of the interaction) to be automatically added to any bot. Our code generation process should be extended for each of these situations accordingly.

Finally, we also plan to enrich the framework by supporting more expressive access control models and languages such as XACML[20] and alternative implementation strategies that facilitate the adoption of our language to deploy secure bots in other environments. We will also start exploring the extension of testing, verification and validation techniques for secured CUIs, expanding on the initial discussion presented in this work.

[20] http://sunxacml.sourceforge.net.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

References

1. 5200.28-STD, D.: Trusted Computer System Evaluation Criteria. Dod Computer Security Center (1985)
2. Attributed based access control (abac) for web services. In: IEEE International Conference on Web Services (ICWS'05). IEEE (2005)
3. Amato, F., Marrone, S., Moscato, V., Piantadosi, G., Picariello, A., Sansone, C.: Chatbots meet ehealth: automatizing healthcare. In: Workshop on Artificial Intelligence with Application in Health, vol. 1982 (2017)
4. Basin, D., Clavel, M., Egea, M.: A decade of model-driven security. In: Proceedings of the 16th ACM Symposium on Access Control Models and Technologies, pp. 1–10 (2011)
5. Basin, D., Doser, J., Lodderstedt, T.: Model driven security: From uml models to access control infrastructures. ACM Trans. Softw. Eng. Methodol. **15**(1), 39–91 (2006)
6. Ben Fadhel, A., Bianculli, D., Briand, L.: Gemrbac-dsl: a high-level specification language for role-based access control policies. In: Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies, pp. 179–190 (2016)
7. Bozic, J., Wotawa, F.: Security testing for chatbots. In: Testing Software and Systems (2018)
8. Bozzon, A., Iofciu, T., Nejdl, W., Taddeo, A.V., Tönnies, S.: Role based access control for the interaction with search engines. In: Ceri, S., Nejdl, W., van Bruggen, J., Assche, F.V. (Eds.) Proceedings of the 1st International Workshop on Collaborative Open Environments for Project-Centered Learning, COOPER-2007, Sissi, Lassithi—Crete Greece, 17 September, 2007, CEUR Workshop Proceedings, vol. 309. CEUR-WS.org (2007). https://ceur-ws.org/Vol-309/paper03.pdf
9. Brambilla, M., Ceri, S., Fraternali, P., Manolescu, I.: Process modeling in web applications. ACM Trans. Softw. Eng. Methodol. **15**(4), 360–409 (2006)
10. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language: Model-Driven UI Engineering of Web and Mobile Apps with IFML. Morgan Kaufmann (2014)
11. Busch, M., Knapp, A., Koch, N.: Modeling secure navigation in web information systems. In: Perspectives in Business Informatics Research: 10th International Conference, BIR 2011, Riga, Latvia, October 6–8, 2011. Proceedings 10, pp. 239–253. Springer, Berlin (2011)

12. Cabot, J., Burgueño, L., Clarisó, R., Daniel, G., Perianez-Pascual, J., Rodríguez-Echeverría, R.: Testing challenges for nlp-intensive bots. In: 3rd IEEE/ACM International Workshop on Bots in Software Engineering. IEEE (2021)
13. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: A fine-grained access control system for XML documents. ACM Trans. Inf. Syst. Secur. **5**(2), 169–202 (2002). https://doi.org/10.1145/505586.505590
14. Daniel, G., Cabot, J., Deruelle, L., Derras, M.: Xatkit: a multimodal low-code chatbot development framework. IEEE Access **8**, 66 (2020)
15. Fiore, D., Baldauf, M., Thiel, C.: "Forgot your password again?" Acceptance and user experience of a chatbot for in-company it support. In: Proceedings of the 18th International Conference on Mobile and Ubiquitous Multimedia, pp. 1–11 (2019)
16. Gondaliya, K., Butakov, S., Zavarsky, P.: SLA as a mechanism to manage risks related to chatbot services. In: 2020 IEEE 6th International Conference on Big Data Security on Cloud (BigDataSecurity) (2020)
17. González, C.A., Cabot, J.: Formal verification of static software models in MDE: a systematic review. Inf. Softw. Technol. **56**(8), 821–838 (2014). https://doi.org/10.1016/j.infsof.2014.03.003
18. González, M., Cernuzzi, L., Pastor, O.: A navigational role-centric model oriented web approach—Moweba. Int. J. Web Eng. Technol. **11**(1), 29–67 (2016). https://doi.org/10.1504/IJWET.2016.075963
19. Groenewegen, D., Visser, E.: Declarative access control for webdsl: combining language integration and separation of concerns. In: 2008 Eighth International Conference on Web Engineering, pp. 175–188. IEEE (2008)
20. Hasal, M., Nowaková, J., Ahmed Saghair, K., Abdulla, H., Snášel, V., Ogiela, L.: Chatbots: security, privacy, data protection, and social aspects. Concurr. Comput. Pract. Exp. **33**(19), 566 (2021)
21. 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 Spec. Publ. **800**(162), 66 (2013)
22. Information technology—Open Systems Interconnection—Security frameworks for open systems: Access control framework (ISO-10181-3/X.812) (1996)
23. Jürjens, J.: Umlsec: extending uml for secure systems development. In: UML 2002-The Unified Modeling Language: Model Engineering, Concepts, and Tools 5th International Conference Dresden, Germany, September 30–October 4, 2002 Proceedings, pp. 412–425. Springer, Berlin (2002)
24. Jürjens, J.: Model-based run-time checking of security permissions using guarded objects. In: International Workshop on Runtime Verification, pp. 36–50. Springer, Berlin (2008)
25. Kashmar, N., Adda, M., Atieh, M., Ibrahim, H.: A review of access control metamodels. Procedia Comput. Sci. **184**, 445–452 (2021)
26. Kim, D.K., Ray, I., France, R., Li, N.: Modeling role-based access control using parameterized uml models. In: Fundamental Approaches to Software Engineering: 7th International Conference, FASE 2004. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29–April 2, 2004. Proceedings 7, pp. 180–193. Springer, Berlin (2004)
27. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Pearson Education (2008)
28. 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)
29. Kotenko, I., Polubelova, O.: Verification of security policy filtering rules by model checking. In: Proceedings of the 6th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems, vol. 2, pp. 706–710. IEEE (2011)
30. 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)
31. 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, Berlin (2002)
32. Nguyen, P.H., Kramer, M., Klein, J., Le Traon, Y.: An extensive systematic review on the model-driven development of secure systems. Inf. Softw. Technol. **68**, 62–81 (2015)
33. Oberortner, E., Vasko, M., Dustdar, S.: Towards modeling role-based pageflow definitions within web applications. In: Koch, N., Houben, G., Vallecillo, A. (Eds.) Proceedings of the 4th International Workshop on Model-Driven Web Engineering, MDWE@MoDELS 2008, Tolouse, France, September 30 , 2008, CEUR Workshop Proceedings, vol. 389. CEUR-WS.org (2008). https://ceur-ws.org/Vol-389/paper01.pdf
34. Oguntosin, V.W., Olomo, A.: Development of an e-commerce chatbot for a university shopping mall. Appl. Comput. Intell. Soft Comput. **2021**, 66 (2021)
35. OMG: Unified Modeling Language (UML) specification. Version 2.5.1 (2017). https://www.omg.org/spec/UML/About-UML/
36. Ouaddah, A., Mousannif, H., Kalam, A.A.E., Ouahman, A.A.: Access control in the internet of things: big challenges and new opportunities. Comput. Netw. **112**, 237–262 (2017). https://doi.org/10.1016/j.comnet.2016.11.007
37. Perera, V.H., Senarathne, A.N., Rupasinghe, L.: Intelligent soc chatbot for security operation center. In: 2019 International Conference on Advancements in Computing (ICAC), pp. 340–345. IEEE (2019)
38. Pérez-Soler, S., Guerra, E., de Lara, J.: Model-driven chatbot development. In: Conceptual Modeling (2020)
39. Pistoia, M., Fink, S.J., Flynn, R.J., Yahav, E.: When role models have flaws: static validation of enterprise security policies. In: 29th International Conference on Software Engineering (ICSE'07), pp. 478–488. IEEE (2007)
40. Planas, E., Daniel, G., Brambilla, M., Cabot, J.: Towards a model-driven approach for multiexperience AI-based user interfaces. Soft. Syst. Model. **20**(4), 66 (2021)
41. Planas, E., Perez, S.M., Brambilla, M., Cabot, J.: Towards access control models for conversational user interfaces. In: Enterprise, Business-Process and Information Systems Modeling - 23rd International Conference, BPMDS 2022 and 27th International Conference, EMMSAD 2022, Held at CAiSE 2022, Leuven, Belgium, June 6–7, 2022, Proceedings, *Lecture Notes in Business Information Processing*, vol. 450, pp. 310–317. Springer, Berlin (2022)
42. Roca, S., Sancho, J., García, J., Alesanco, Á.: Microservice chatbot architecture for chronic patient support. J. Biomed. Inform. **102**, 66 (2020)
43. Salnitri, M., Dalpiaz, F., Giorgini, P.: Modeling and verifying security policies in business processes. In: Enterprise, Business-Process and Information Systems Modeling: 15th International Conference, BPMDS 2014, 19th International Conference, EMMSAD 2014, Held at CAiSE 2014, Thessaloniki, Greece, June 16–17, 2014. Proceedings, pp. 200–214. Springer, Berlin (2014)
44. Sandhu, R., Ferraiolo, D., Kuhn, R.: The NIST model for role-based access control: towards a unified standard. In: RBAC'00. ACM (2000)
45. Sandhu, R.S., Samarati, P.: Access control: principle and practice. IEEE Commun. Mag. **32**(9), 66 (1994)
46. Sağlam, R.B., Nurse, J.R.C.: Is your chatbot GDPR compliant? Open issues in agent design. In: Proceedings of the 2nd Conference on Conversational User Interfaces (CUI'20). Association for Computing Machinery (2020)

47. Song, E., Reddy, R., France, R., Ray, I., Georg, G., Alexander, R.: Verifiable composition of access control and application features. In: Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies, pp. 120–129 (2005)
48. Ye, W., Li, Q.: Chatbot security and privacy in the age of personal assistants. In: 2020 IEEE/ACM Symposium on Edge Computing (SEC) (2020)