EBNF Grammar of DEMO Action Rules Specification and Process and Fact Models Representations for a Municipal Hearings Process

ANNEX TO PAPER PUBLISHED IN PROCEEDINGS OF KEOD 2024 CONFERENCE

David Aveiro[1,2,3][0000−0001−6453−3648], Vítor Freitas[1,3] [0009−0002−0667−5749],Duarte Pinto[1,2] [0000-0002-8451-5727],Valentim Caires[1,2] [0000−0002−0871−7212],Dulce Pacheco[1,2] [0000−0002−3983−434X]

[1] ARDITI - Regional Agency for the Development of Research, Technology and Innovation, 9020-105 Funchal, Portugal
[2] NOVA-LINCS, Universidade NOVA de Lisboa, Campus da Caparica, 2829-516 Caparica, Portugal
[3] Faculty of Exact Sciences and Engineering, University of Madeira, Caminho da Penteada 9020-105 Funchal, Portugal

daveiro@staff.uma.pt, vitor.freitas@arditi.pt, duarte.nuno@arditi.pt, valentim.caires@arditi.pt, dulce.pacheco@arditi.pt

Introduction

This annex provides additional technical details and diagrams that complement the main body of the paper published in Proceedings of KEOD 2024. It includes a more detailed explanation of the new representations of the Process Model and the Fact Model along with examples, as well as the full Extended Backus-Naur Form (EBNF) grammar for the extended Action Rule Specification discussed in Section 4, reusing the same examples.

New Process Model Representation

The alternative Process Structure Diagram (PSD) representation for DEMO's PM which integrates some contents of the CM, in order to improve the clarity, reduce the redundancies and complexity and increase the transparency in the representations allowing for a more comprehensive representation of the operational flow of the organizational processes. These goals are achieved by: 1) simplifying the nomenclature, to make it more accessible for non-specialists, like for instance renaming “transactions” to “tasks” to align with the common language used within organizations, 2) enhancing the process model diagram by offering an alternative notation with more visual queues and taking advantage of colors and 3) moving the fine details of the process model to a transaction description table removing the clutter from the overall picture in the diagram representation but allowing it all to still be easily accessible and manageable when the need arises, using the process model diagram. In Figure 1 we can find the PSD for the MHP.

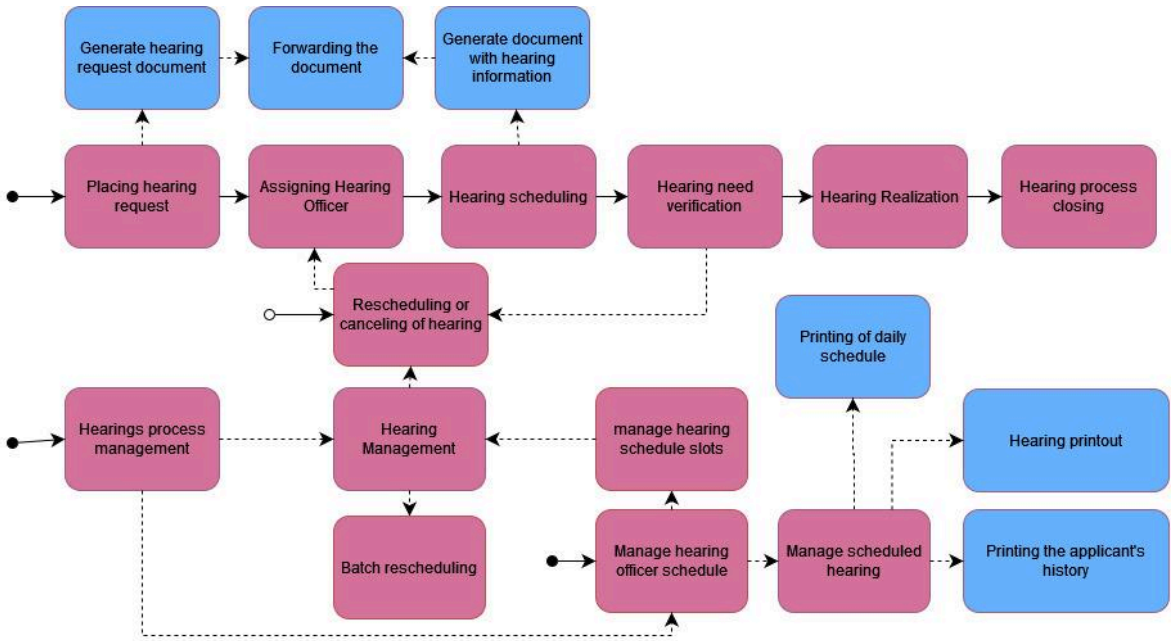

Figure 1. Process Model Diagram

New Fact Model Representation

In the older version of the official DEMO's FM some issues were identified, namely the complexity and poor usability caused by the over-cluttering of shapes that are hard to understand by those not specialized in DEMO, the lack of flexibility to accommodate changes and updates, and the inadequate visual representations with multiple symbols that are not intuitive and can lead to misinterpretation and errors. To address these problems, an alternative notation has been proposed, with what are considered simpler and more intuitive diagrams and tables, readable by any stakeholders that have no knowledge of DEMO, just business know-how. These new representations also address dynamic changes to the models, thus increasing their flexibility and allow for a more detailed representation of facts including their origins and relationships that help to more accurately document and track the process being modeled.

The main artifact of this new Fact Model is the Concepts and Relationships Diagram (CRD), a generic, global, and synthetic view of an entire domain's concepts while abstracting from their attributes. In the CDR, a concept is represented by a collapsible box whose expansion discloses its attributes, one per line. The value type of the attribute is specified to the left of the line, while to the right is the attribute's name. The value type can be any of the following options: category, reference, document, text, doc \& text, number, date, or boolean. Arrows express relationships, which will always consist of an attribute in one concept whose instances will reference instances of the other concept. Cardinalities are represented with arrows pointing to relationships' "one side". A dark-filled circle attached to a concept in one connector means that an instance of this concept, in order to exist, depends on an instance of the concept at the other end of the connector. The specialization/generalization relationship is depicted using a connector with a pointed line. In Figure 2, we can find the CRD of the MHP case study.

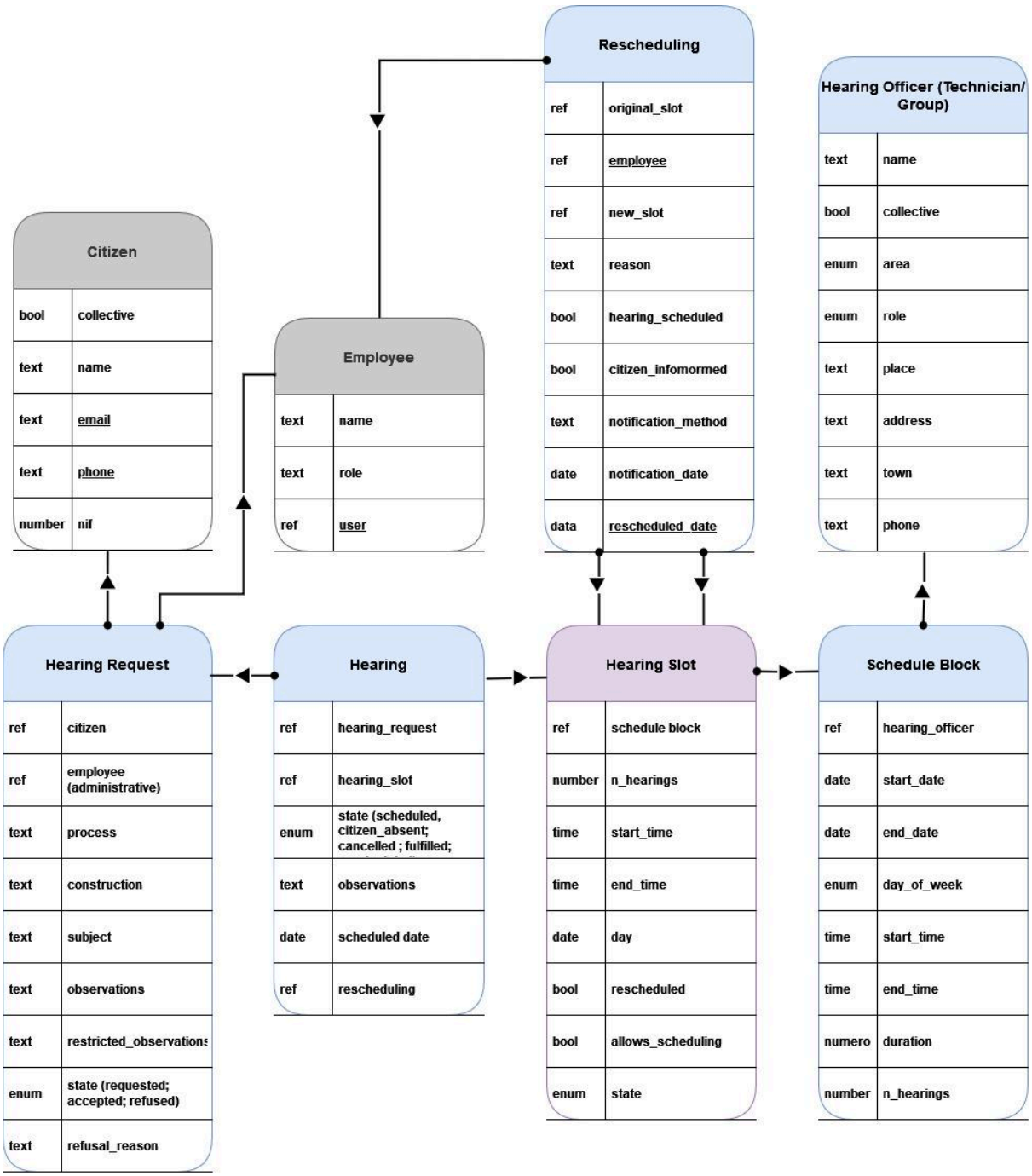

Figure 2. Concepts and Relationships Diagram

A detailed explanation of each concept in the model follows.

The Citizen concept represents the person (or group of people, like, for example, a condominium) who initiates a Hearing Request. An Employee is a worker of the Municipality. Both of these concept's data are managed by specific systems of the municipality and a copy of them is kept in the low-code system, synchronized. There are multiple roles involved in the hearing process, such as municipality clerks, assistants, and the hearing officers of multiple sections of the Municipality. The Hearing Officers are the Municipality councilors/directors or task forces created for specific subjects such as large construction developments who can assist the Citizen(s) through hearings. The Hearing Officers require more attributes than regular employees, and as such warrant a concept of their own.

These attributes include their area of expertise, their role in the section they are part of (that might not be the same as in the municipality Active Directory) and address information (that is also the hearing location).

The Schedule Block concept, which is associated to each of the Hearing Officers and represents a time frame when a Hearing Officer is available to hold Hearings, which are traditionally a few hours at a specific day in a week and apply during a long period of time, allowing the creation of Hearing Slots in batch for each of those days, based on the input information like, start and end time and dates, the recurring day of the week and the duration for each slot. The Hearing Slot concept is an automatically generated time slot based on a Schedule Block that can hold a hearing. These slots are open by default when created and become occupied when a hearing is associated with them. Each one references the day, start time and end time and the order in the hearings of that given day. On events of a rescheduling, a flag is used to mark the necessity to check the rescheduling concept for the new allocated slot for the hearing.

A Hearing Request concept precedes a concrete hearing, citizens can make hearing requests at the municipalities front desk and these are handled by their front desk clerks. They detail the necessary information for the hearing to take place, but also include pertinent observations as well as hidden observations that may influence the outcome of the request. For example, if a citizen already had a hearing for the same subject a short time ago, a new hearing traditionally won't be scheduled as it is unlikely to have been any relevant progress or that the outcome from the previous hearing would change. The Hearing concept represents the hearing itself and contains all the information related to it, the scheduled date, the observations, and outcomes of the hearing. As mentioned before, each hearing derives from a hearing request, although not all requests end up in a hearing, and all hearings take place in a hearing slot from the hearing officer that is selected as the most qualified to deal with the subject at hand.

Finally, the rescheduling concept happens when a hearing that was scheduled to happen in a certain Hearing Slot, but has to be rescheduled and set to another Hearing Slot. This can happen for multiple impediments, both by the citizen or the hearing officer. The concept itself saves the information related to that change, referencing the original (old) and the new Hearing Slot, the employee that made that change, the reason(s) to that change, the new date for the hearing as well as some information about notifying the citizen(s) of this change when the change was generated internally (by or on behalf of the hearing officer).

EBNF Grammar

Our EBNF grammar establishes the syntax of our action rules using specific conventions.
Parentheses "( )" means grouping, square brackets "[ ]" indicate optional elements, and curly braces "{}" denote optional elements that can occur zero or more times. The vertical bar "|" separates alternatives, offering different options for a syntax element. A hyphen "-" marks a syntactic exception, distinguishing a rule that must be used (on the left) from one describing what is not allowed to be used (on the right). If there's nothing on the right, it indicates that the element before it must be used at least one time. Terminal symbols, represented in bold, possess specific behaviours within the system. Each is accompanied by a short explanatory sentence.

Furthermore, additional explanations and examples follow each entry in our grammar to clarify usage, when necessary.

when = **transaction_type is** | **has_been** transaction_state { action }-

This is the root of the syntax. For every action rule defined one must specify which actions should be executed in the context of a transaction type, among those specified in the system, in the activation of a particular transaction state (out of the 21 possible states as per DEMO's PSI-theory). At least one action must be specified. e.g.: **when** *Create Hearing Officer* **is executed** perform the action **instance creation** for a hearing officer entity (present to the system user a form to input hearing officer information).

transaction_type = **"**a transaction_type specified in the system"

Corresponds to the DEMO concept of transaction.

Ex.: Placing Hearing Request

is | ***has_been*** = for each transaction_state, one can have two action rules: one for when the respective act **is** being performed and another when it **has been** performed, effectively decomposing a "normal" action rule into two action rules for each transaction step, one regarding the act itself and another for the respective fact created.

It is also worth noting that the responsible role for the *has_been* action rule is the opposite to the one responsible for the *is* action rule while in the same transaction state, that is, if it is the initiating role of the transaction that is responsible for the *is* action rule, it will be the executing role that will be responsible for the *has_been* action rule of the same transaction state, and vice versa. This allows an even more clear separation of responsibilities in DEMO models.

transaction_state = **requested** | **promised** | **executed** | **declared** | **accepted** | **declined** | **rejected** | **revoke_request_requested** | **revoke_request_allowed** | **revoke_request_refused** | **revoke_promise_requested** | **revoke_promise_allowed** | **revoke_promise_refused** | **revoke_declare_requested** | **revoke_declare_allowed** | **revoke_declare_refused** | **revoke_acceptance_requested** | **revoke_acceptance_allowed** | **revoke_acceptance_refused**

All states according to DEMO's PSI-theory.

action = if | assign_expression | create_instance | update_instance | user_output | produce_doc | create_schedule_slots| for_each | for_n | causal_link

The different action kinds on our syntax. More on each action type can be seen in their definition.

if = condition
then { action } -
[**else** { action } -]

Enables the controlling of the execution flow of action rules based on conditional statements. If the condition is evaluated to being *true*, the actions specified in the *then* part will be executed. In case the evaluation returns a negative value, the actions in the *else* part will be executed, if specified. If there are no actions specified in the optional field *else*, the execution engine will simply pass over the *if* action without executing any further actions inside its definition.

An example of the usage of this element, using our visual programming component, can be seen in Figure 3. Here, we define an *if* flow that in its *condition* has a *user evaluated expression*, so the user can evaluate the "Go to Selection of Officer for Hearing" expression. In case the expression is evaluated positively, the *causal link action* in the *then* input will be run, otherwise the *causal link action* in the *else* input, specified in this case, will be the one who will be run at run-time.

condition = (**is_true** | **not**
evaluated_expression | condition) |
(**and** | **or**
{ evaluated_expression | condition }-)

Conditions are evaluated at run-time and define the execution flow of an action at a specific moment in the execution of an action rule. If the condition is of type *is true*, the execution engine will evaluate the expression specified. If the condition is of type *not*, the execution engine will evaluate either the expression or sub-condition. If the condition is of type *and* or *or,* it will accordingly evaluate the specified expressions/sub-conditions.

evaluated_expression = comp_evaluated_expression | **user_evaluated_expression**

The expressions to be evaluated at run-time inserted into a parent condition, which can be evaluated by the system automatically or by the user manually. More on each type on their respective definition.

user_evaluated_expression = "informal expression"

The execution engine presents the user with a modal containing this "informal expression" that has to be evaluated by the user, who will decide on an evaluation result of true or false.

comp_evaluated_expression = term logical_operator (term | **property_value**)

The execution engine will evaluate automatically the constructed expression from the specification made in the action rule, resulting in an evaluation result of true or false.

logical_operator = "<" | ">" | "==" | "!=" | ">=" | "<="

For now simple mathematical operators, more to be included in the future, such as logical operators for strings, dates, etc.

property_value = possible value of an "enum" or "prop_ref" property

Must be a possible value of a property with two cases: 1) value type is enum and one can select one of the allowed values for the selected property; 2) value type is prop_ref where possible values to select will be the values associated with the property's fk_property.
Ex: property "Employee" of entity type "Hearing Request" has many values for the different employees registered in the system.

An example of the usage of this element, using our visual programming component, can be seen in Figure 6, on the value assigning of the third *additional fact* "State".

term = constant | **value** | **property** | query | compute_expression | **context_variable** | **executing_role** | **executing_user**

The different *term types* on our syntax. By allowing this grouping of different concepts under the *term* concept, it allows us to have a greater reusability and componentization of the different elements of the syntax.

constant = value_type **constant_name constant_value**

It can be a reference to a *constant* defined on the system previously or can be the specification of a new *constant*; in the latter case, the *value type*, *name*, and *value* of the *constant* to be created must be specified.

constant_name = simply, the name of the constant in the system.

constant_value = concrete value assigned to the constant in the system, according to the constant's value type.

value = simply, the value that will be inputted to the system.

Free value inserted when creating/editing an action rule. The value type is automatically determined by the value type of the element that will be on its left, and the system will restrict the kind of values that can be specified according to that.

An example of the usage of this element, using our visual programming component, can be seen in Figure 6, on the value assigning of the two first *additional facts*, namely the "Rescheduled" and "Allows Rescheduling".

property = has to be an existent property specified in the system

Ex: "Citizen" from the entity type "Hearing Request"

query = **query_name** { term }

Has to be an existing *query* specified in the system's query definition component. It will be followed by as many *terms* as there are *query parameters* (from the query filters) and the *value type* of the *parameters* of the query must be compatible with the *value type* of the terms.

An example of the usage of this element, using our visual programming component, can be seen in Figure 3, on the *query* that provides the options for the "Citizen" *form fact* from the "Hearing Request" *entity type*.

query_name = simply, the name of the query in the system.

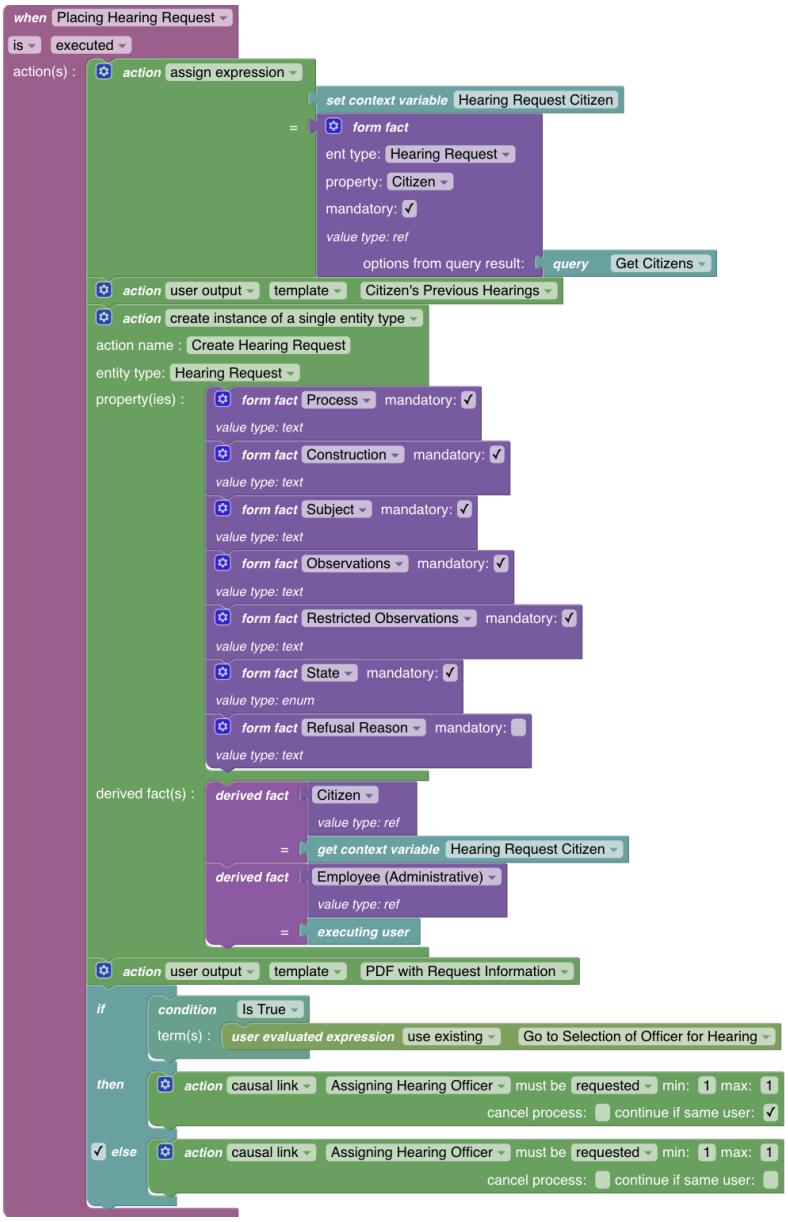

Figure 3. Action Rule for the transaction step "Placing Hearing Request is executed"

compute_expression = term {**compute_operator** term}-

Used when we want to compute the value of a mathematical expression that may contain the different term types in our syntax, including sub-compute expressions themselves as its variables. Its main objective is to calculate a value based on a mathematical evaluation of these options' value at run time. During runtime, the construction of these expressions takes place, taking into account their terms' value at that specific moment in the process.

An example of the usage of this element, using our visual programming component, can be seen in Figure 4, on the value assigning to the *derived fact* “No. of Hearing”. Here, a dividing *compute expression* is used to divide a sub-*compute expression*, that in its turn subtracts the “Start Time” from the “End Time” from the *entity type* “Schedule Block” and returns its value in minutes, and the “Duration” of the same *entity type*.

compute_operator = “+” | “ -” | “*” | “/” | “^”

For now simple mathematical operators, more to be included in the future.

context_variable = the name of a context variable which the system will create and store the respective data.

Serves a similar function as local variables in a function in a programming language. In a certain action rule with several actions, one might need to keep information that might be needed in a future action. The type of the variable will be automatically defined by the element on the right side of an assign expression, and can be, e.g., a value of some property specified in the system, a set of values of this same property, or a set of tuples resulting from a query. One needs to specify a name for the variable, e.g., "requesting citizen" - in this case, this variable will have as value a reference to the citizen selected by the admin staff to have their hearing request processed.

Examples of the usage of this element, using our visual programming component, can be seen in Figure 3, where we first define a new *context variable* “Hearing Request Citizen”, and then use it to assign its value to the *derived fact* “Citizen” from the “Hearing Request” *entity type*.

executing_role = the responsible role of the user executing the current action rule.

In some situations one will want to save in a certain generated instance, a reference to the role who executed a certain action/created a certain instance, e.g., the role of admin staff processing the hearing request.

executing_user = the user executing the current action rule.

In some situations one will want to save in a certain generated instance, a reference to the person who executed a certain action/created a certain instance, e.g., the concrete admin staff processing the hearing request

assign_expression = (**property** | **context_variable**) “=” (term | form_fact | **property_value)**

Assigning a specific value to a *property*/*context variable*. In the case where we assign a *term* or *property value*, the attribution is made automatically without the need for user intervention. In the case where we assign a *form property*, a form will be shown for the user with a single field for the definition of that *form property*’s value.

Ex: **allows_scheduling** “=” **1** - Assign the property value 1 (true) to the allows_scheduling boolean property on a Hearing Slot entity upon their creation.

Another example of the usage of this element, using our visual programming component, can be seen in Figure 3, where we assign the value of a *form fact*, that obtains its possible values from a *query*, to a new *context variable* “Hearing Request Citizen”.

create_instance = **entity_type** {form_fact | derived_fact}-

Renamed the "user_input" element to "*create_instance*" because a created instance of a certain entity type might obtain values from things other than inputs from a form. In this action the system will prompt the user for input through a form, that is, for the user to input some data, if there are specified *form facts* for this action. The specified derived facts will have a value automatically assigned. These implement DEMO’s notion of derived fact specifications. If there are no *form facts* specified, the action will run automatically by the system's engine without user intervention.

Ex: Create Instance Hearing Officer Agenda Block prompting the user to fill a form with multiple properties of the entity type Schedule Block (start_date, end_date, day_of_week, etc). This example of the full definition of this action type, using our visual programming component, can be seen in Figure 4.

entity_type = has to be an existent entity_type specified in table ent_type

Ex: “Hearing Request”

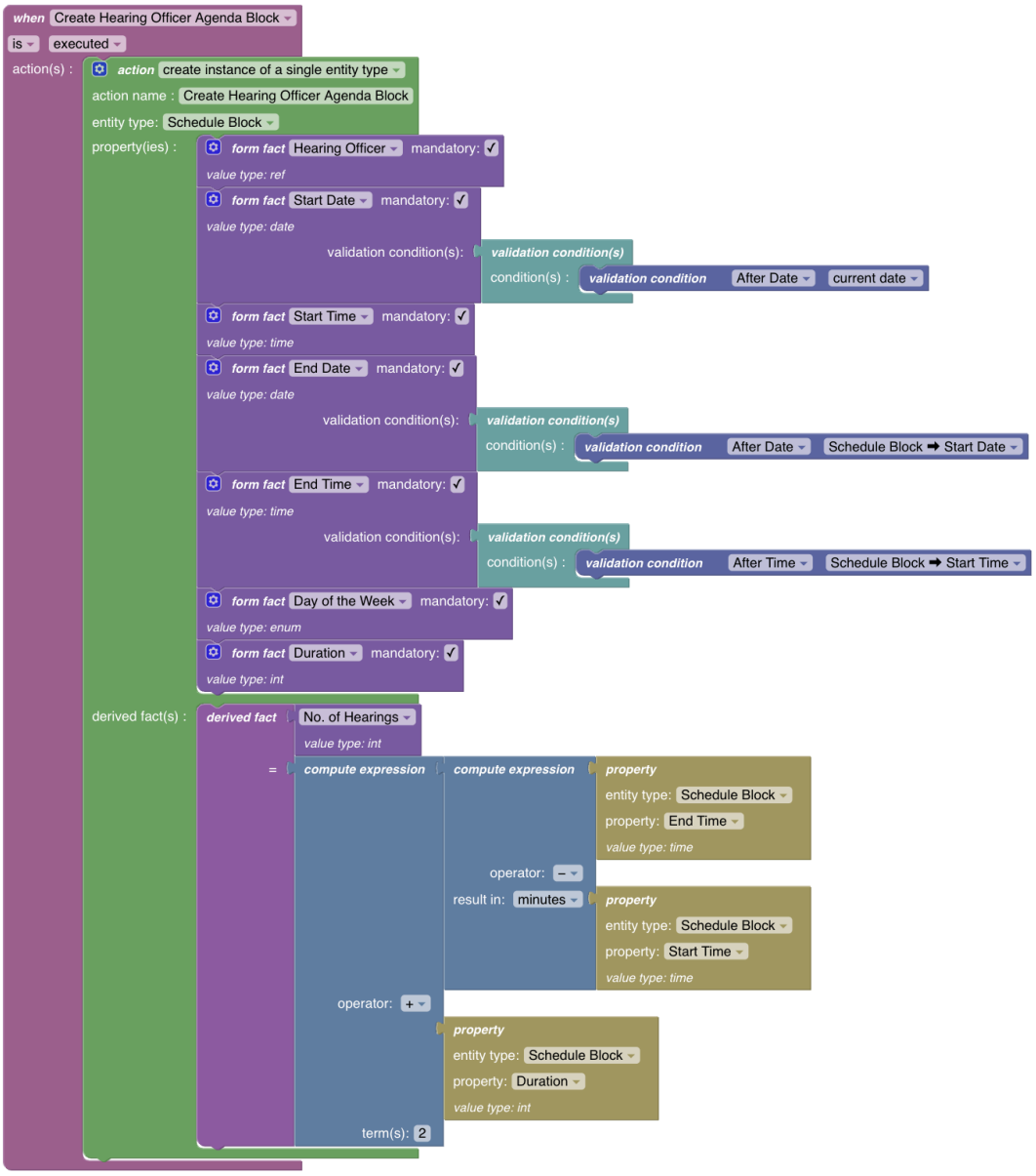

Figure 4. Action Rule for the transaction step “Create Hearing Officer Agenda Block is executed”

update_instance = **entity_type** entity_details { form_fact | derived_fact}-

Renamed the "edit_entity_instance" element to "*update_instance*" so that the syntax is more coherent. This action allows the update of an entity instance specified in the system. All properties inserted here must have the flag ‘editable’ as true. Will prompt the user for input through a form, that is, for the user to input some data, if there are specified *form facts* for the respective action. These form fields will also come pre-filled with the property’s formerly defined value. The specified derived facts will be assigned a value automatically without the need for user intervention. If there are no *form facts* specified, the action will run automatically. At run-time, in the initialization of the action, a select box will be shown to the user to select the exact entity that should be updated.

An example of the full definition of this action type, using our visual programming component, can be seen in Figure 5. Note that the defined entity details are the “Name”, “Role” and “Address”

properties. This means that when the action begins its execution, the user will be presented with a modal containing a select box so that they can choose which instance will be updated. In this select box, containing every instance of the “Hearing Officer” entity type in the system, every option will have these three properties displayed, so that the user can easily identify/distinguish the system’s instances, in order to select the intended one

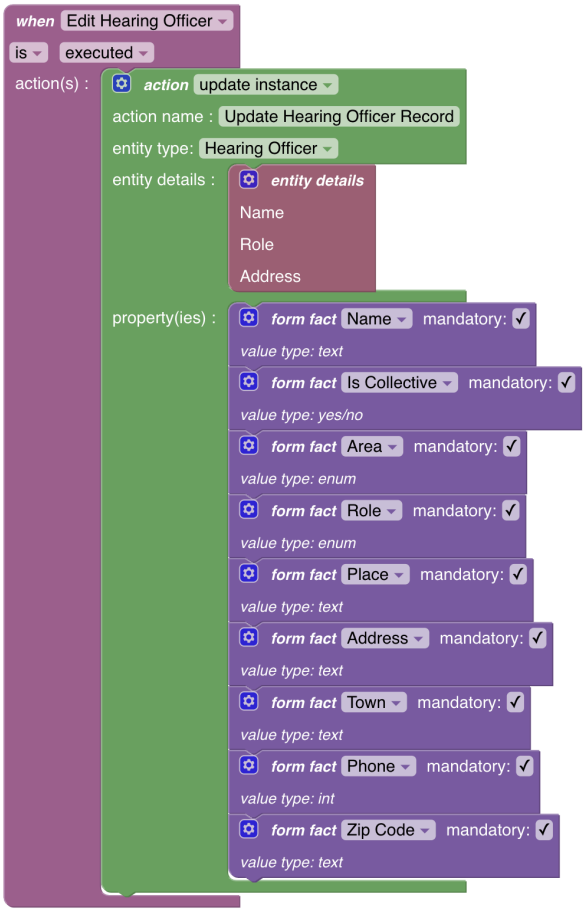

Figure 5. Action Rule for the transaction step “Edit Hearing Officer is executed”

entity_details = {property}-

Values of the respective properties specified here are to be shown in the entity selection modal’s select box in 'update instance' actions. The function is to provide enough information, so a user can easily identify/distinguish the system’s instances, in order to select the intended one.

derived_fact = **property** “=” term

The derived fact specified here, to be attributed automatically to the stated property, has to be of a single value kind, that is, if it's a query, it must be a query whose result can be only a single value (e.g. a count). The values to be assigned to the properties of a created instance might come from any term, which can be a context variable, the result of a query, etc. e.g., a hearing request instance creation will have the following *derived facts*: “Citizen” and “Employee”, with its values coming from the *context variable* - requesting citizen - and *executing user*, respectively.

form_fact = property ([enable_condition] validation_condition [**mandatory**]) | [form_calculation] | [query]

Properties whose value will be inserted/chosen by the user through the filling of a form. This element has been changed, now the enable condition, validation condition and mandatory elements are part of a group (inside parenthesis). If it's some kind of user input, these can apply as a set; then, as an alternative, the value to assign to the property might be a calculation based on some other values of the same form; and, as a final alternative and the new element in this paper (the previous changes are slight corrections), the possible values to be selected to be assigned to the property might be the result of a query, and the system must verify that the type of result of this query is a set of values of which their value type is the same as the value type of the property.

Examples of the usage of this element, using our visual programming component, can be seen in Figures 3, 4 and 5.

form_calculation = compute_expression

Enables the definition of computations regarding data in the current form for a specific field, with that property being filled automatically based on the given expression instead of a manual fill by the user.

enable_condition = condition

Used when we want that a property is hidden/disabled from the form unless the specified condition is true, which in that case the property will be shown.

validation_condition = [**not**] validation_condition_type [**extra_field_1** [**extra_field_2**]] [user_output]

Validation conditions have to be satisfied/validated so that the user can submit the form data, being that if the condition is not satisfied, an error message is presented back to him. The *not*, *extra_field_1* and *extra_field_2* fields will appear depending on the validation_condition_type chosen; e.g., *extra_field_1* and *extra_field_2* can be the min and max fields if the validation_condition_type is "*belongs range*" (see below).

An example of the usage of this element, using our visual programming component, can be seen in Figure 4, on three *form facts*, namely an "after date" *validation condition* on *form fact* "Start Date", a "before date" *validation condition* on *form fact* "End Date", and an "after time" *validation condition* on the "End Time" *form fact*.

validation_condition_type = **equal_to** | **max_word_length** | **less_equal** | **higher_equal** | **higher_than** | **less_than** | **min_length** | **belongs_range** | **max_length** | **min_word_length** | **has_character** | **reg_expression** | **has_word** | **is_email** | **is_url** | **after_date** | **before_date** | **after_time** | **before_time** | **custom_validation**

The different *validation condition types* on our syntax for *form properties*. Depending on the current *form property*'s value type, the presented validation conditions will vary. For example, if the current

form property is "*No. of Hearing*", of type "*int*", from the "*Hearing Slot*" *entity type*, the presented validation condition types will be "*belongs range*", "*equal to*", "*higher equal*", "*higher than*", "*less equal*", "*less than*", "*custom validation*" and "*regular expression*".

mandatory = checkbox in the block that specifies if the filling of the property should be mandatory in its respective form

user_output = **template**

Showing of information to the user. We retrieve the content defined in the template to output a custom notification or dialogue box directly to the user when the action rule is run, or to produce and show a document.

Ex: Show previous hearing requests from a citizen when requesting a new one. The usage of this element for this example, using our visual programming component, can be seen in Figure 3, in the first action of this type to appear. Later, in the same Figure 3, it can also be seen another usage of this element, where we produce and then present a document to the user based on the "PDF with Request Information" *template*.

produce_doc = property "=" **template**

Generation of a document and consequent storing on DISME's file system, following the template definition, without showing it directly to the user. The generated file will then be stored as the value of the selected property, which must be of value type "doc".

Ex: Create a PDF with the information of a hearing to deliver / send to a citizen.

template = "HTML"

Special HTML code defined in the template manager component with the help of a form, for specifying the template's name and other variables, and a WYSIWYG (What You See Is What You Get) HTML editor. In this editor, it is possible to insert dynamic values, which will be evaluated at run-time and inserted into the template.

create_schedule_slots = scheduling_record slot_records

This *action* will serve to automatically create time slots based on a *scheduling record entity type*. This *entity type* must be specified in the system and will be used here to set which of its properties holds the information about specific scheduling properties that are needed to create time slots. The *slot records entity type* must also be specified in the system and will be used to set in which properties the information about the specific time slots created should be stored. As scheduling functionalities are crucial in many business types, this kind of action and the elements following below were devised to facilitate the specification of scheduling properties and execution of scheduling mechanisms. Several of these properties were inspired by the proficient online scheduling service Appointlet[1].

[1] https://www.appointlet.com

An example of the full definition of this action type, using our visual programming component, can be seen in Figure 6.

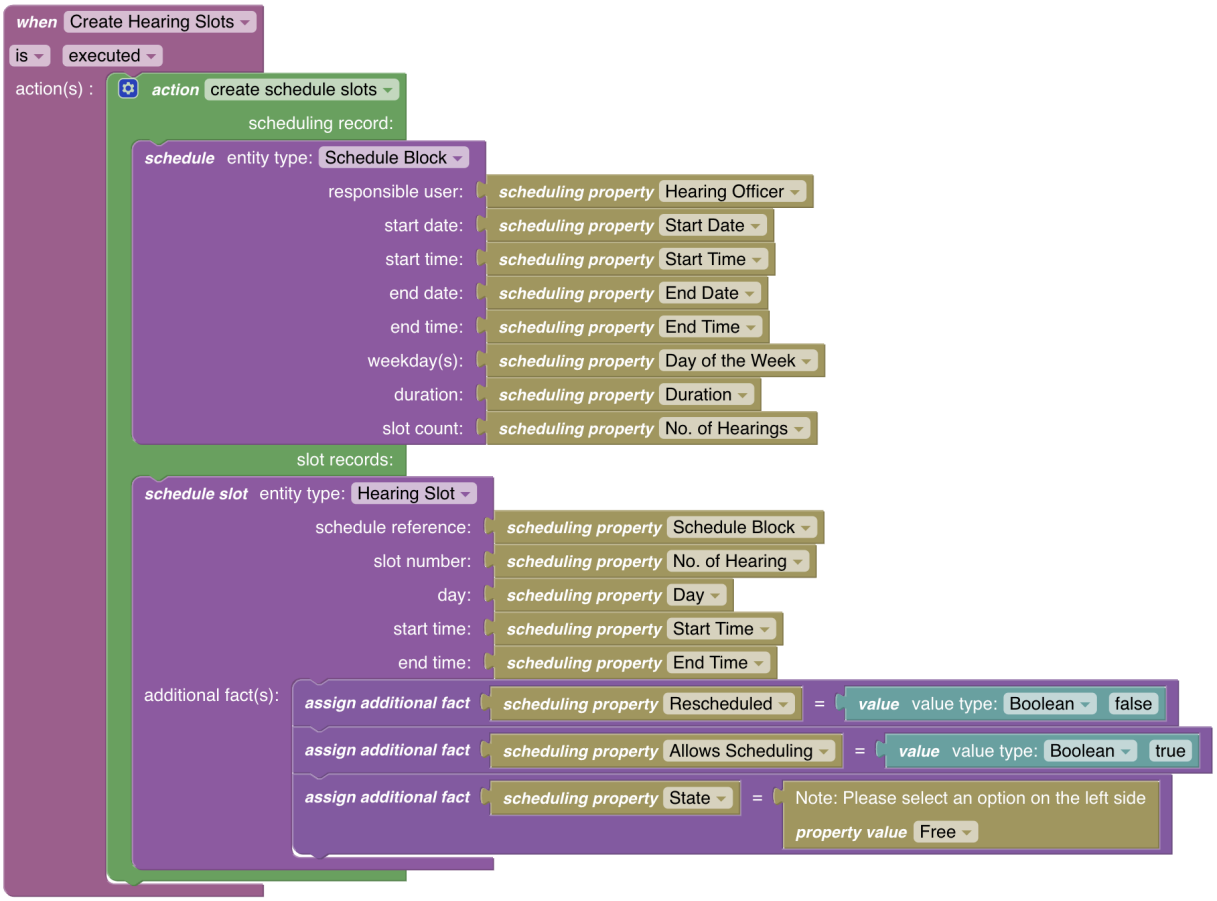

Figure 6. Action Rule for the transaction step "Create Hearing Slots is executed"

scheduling_record = **entity_type responsible_user start_date start_time end_date end_time weekday duration slot_count**

Except for the entity_type element, all other elements of this entry must be matched to properties of the entity type specified as *scheduling record entity type*.

slot_records = **entity_type schedule_reference slot_number day start_time end_time** {additional_fact}

Except for the entity_type element, all other elements of this entry must be matched to properties of the entity type specified as *slot records entity type*.

additional_fact = **property** "=" term | **property_value**

In case we have other properties belonging to the specified *slot records entity type*, which aren't specific to the *slot records* fundamental elements, that we want to specify a value for. For example, the "Rescheduled", "Allows Rescheduling" and "State" of the slot records entity type "Hearing Slot".

for_each = set { action }-

Action that enables us to iterate over each element in a *set*, executing an *action* or set of *actions* for each element. An example of its usage can be seen in an action where we have a rescheduling of X hearing slots from a specific officer, with the set being determined by a context variable that had its value determined by a query that fetched all the hearing slots for a determined time interval in which the officer was in vacations.

set = **context_variable**

Here, the *context variable* that has its value equalled to the set must refer to the result of a *'query'* which returns a set of results.

for_n = **context_variable** { action }-

Action that enables us to iterate over a range of numeric values, executing an *action* or set of *actions* for each value of n within that range. Here, the *context variable* must be of type *integer* and larger than zero, as it will serve as the 'n' in this loop.

causal_link = transaction_type transaction_state [**min** [**max**]] [**cancel_proc**] [**continue_if_same_user**]

Generating a new transaction instance or changing the transaction state of the current transaction. *min* and *max* are optional and by default come with 1 as pre-filled; if *min* doesn't exist, by default=1. If *max* doesn't exist, by default = *min*.

Ex: accepting the Placing Hearing Request generates a causal link to request Assigning Hearing Officer with the default cardinality of 1 without cancelling the process and without continuing if it's the same user as this task takes place independently, usually at a later time.

Other examples of the usage of this element, using our visual programming component, can be seen in Figure 3, on the *then* and *else* inputs of the specified *if* action.

min = integer

max = integer | *

cancel_proc = refers to whether the causal_link cancels the current process.

Refer to whether the causal link action cancels the current process, for example on the passage of a transaction to the "quit" state.

continue_if_same_user = marks whether the execution engine should take the user directly to the execution of the linked transaction when it reaches this action, in case the user is an allowed executor of that transaction.

Whether the Execution Engine should take the user directly to the execution of the transaction step specified in the causal link, when it reaches this action, in case the current user in the execution engine's task is also responsible for that step.